University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334

2 Experiment 2: Instruction Set Analysis 2 & Modular Programming Techniques

Objectives

The main objectives of this experiment are to familiarize you with:

- Program flow control instructions
- Conditional and repetition structures
- The concept of modular programming
- Macros and Subroutines

Pre-lab requirements

Before starting this experiment, you should have already familiarized yourself with MPLAB software and how to create, simulate and debug a project.

Introducing conditionals

The PIC 16series instruction set has four instructions which implement a sort of conditional statement: ***btfsc*** , ***btfss, decfsz and incfsz*** instructions.

1. **btfsc** checks for the condition that a bit is clear: 0 (***B***it ***T***est ***F***ile, ***S***kip if ***C***lear)
2. **btfss** checks for the condition that a bit is set one: 1 (***B***it ***T***est ***F***ile, ***S***kip if ***Set***)
3. Review ***decfsz*** *and* ***incfsz*** *functions from the datasheet*

Example 1:

```
movlw 0x09
btfsc PORTA, 0
movwf  Num1
movwf  Num2
```

The above instruction tests bit 0 of PORTA and checks whether it is clear (0) or not

- If it is clear (0), the program will **skip** "movwf Num1" and will only execute "movwf Num2"
  Only Num2 has the value 0x09
- If it is set (1), it will not skip but **execute** "movwf Num1" and then **proceed** to "movwf Num2"
  In the end, both Num1 and Num2 have the value of 0x09

You have seen above that **if the condition fails**, the code will continue normally and both instructions will be executed.

Example 2:

```
movlw 0x09
btfsc PORTA, 0
goto firstcondition
goto secondCondition
Proceed
…….. your remaining code
firstcondition
movwf  Num1
goto Proceed
secondCondition
movwf  Num2
goto Proceed
```

Firstcondition, secondCondition, and Proceed are called Labels, Labels are used to give names for a specific block of instructions and are referred to as shown above to change the program execution order.

Example 2 is basically the same as Example 1 with one main difference:

- If it is clear (0), the program will **skip** "*goto firstcondition*" and will only execute "*goto secondCondition*", the program will then execute "*movwf Num2" and then "gotoProceed"*
  Only Num2 has the value 0x09
- If it is set (1), it will not skip but **execute** "*goto firstcondition*", the program will then execute "*movwf Num1" and then "gotoProceed"*
  Only Num1has the value 0x09

Conditional using Subtraction and how the Carry/$\overline{\text{Borrow}}$ flag is affected?

The Carry concept is easy when dealing with addition operations but it differs in borrow operations according to Microchip implementation.

Carry is a physical flag; you will find it in the STATUS register,
Borrow is not implemented; it is in your mind ☺
In the following examples we will show the status of the Carry/$\overline{\text{Borrow}}$ flag and how it differs between addition and subtraction operations:

Ex1) 99-66 10011001 – 01100110 10011001+ 10011010 2's complement of 66 ^{1}00110011 There is carry (C = 1), since Borrow is the complement of Carry, then Borrow is 0 (No borrow) which is correct	Expect no borrow since 99 > 66
Ex 2) 66 – 99 01100110- 10011001 01100110+ 01100111 ^{0}11001101 There is no carry (C = 0), since Borrow is the complement of Carry, then Borrow is 1 (There is borrow) which is correct	Expect borrow since 66 < 99

Program One: Check if a value is greater or smaller than 10, if greater Result will have the ASCII value G, if smaller, it will have the ASCII value S.

```
1   include "p16F84A.inc"
2   cblock  0x25
3           testNum
4           Result
5   endc
6           org     0x00
7   Main
8           movf    testNum, W
9           sublw   .10             ;10d - testNum
10          btfss   STATUS, C
11          goto    Greater         ;C = 0, that's B = 1, then testNum > 10
12          goto    Smaller         ;C = 1, that's B = 0, then testNum < 10
13  Greater
14          movlw A'G'
15          movwf Result
16           goto   Finish
17  Smaller
18          movlw A'S'
19          movwf Result
20  Finish
21          nop
22          end
```

1. Start a new MPLAB session, add the file ***example3.asm*** to your project
2. Build the project
3. Select **Debugger ↳ Select Tool ↳MPLAB SIM**
4. Add the necessary variables and the working register to the watch window (remember that user defined variables are found under the "**Add Symbol**" list)
5. Enter values into testNum, simulate the program step by step, concentrate on what happens at lines10-12
6. Keep an eye on the Flags at the status bar below while simulating the code
7. Enter other values lesser and greater and observe how the code behaves

❖ What is the value stored in Result when testNum = 10? Is this correct? Can you think of a solution?

Program Two: Counting the Number of Ones in a Register's Lower Nibble
Introducing simple conditional statements

This program will take a hexadecimal number as an input in the lower nibbles (bits 3:0) in a register called testNum.The number will be masked by anding it with 0F, (remember that 0 & Anything = 0, while 1 & anything will remain the same), we used masking because if the user accidentally wrote a number in the higher nibble (bits 3:0), it will be forced to zero. The number in the lower nibble will not be affected (anded with 1). The masked result will be saved in a register called tempNum.
Now tempNum will be rotated to the right, bit0 (least significant bit) will move to the C flag of the STATUS register after rotation. Then it will be tested whether it 0 or 1. If it is 1, the numOfOnes register will be incremented. Else the program proceeds. This operation will continue for 4 times (because the number of bits in the lower nibble is 4)

Byte 8 bits							
7	6	5	4	3	2	1	0
Higher 4 bits				Lower 4 bits			
Upper Nibble				Lower Nibble			

```
1   include "p16f84a.inc"
2
3   cblock 0x20
4           testNum             ;GPR @ location 20
5           tempNum             ;GPR @ location 21
6   endc
7
8   cblock 0x30
9           numOfOnes           ;GPR @ location 30
10  endc
11
12  org 0x00
13  clrf    numOfOnes     ;Initially number of ones is 0
14  movf    testNum, W    ;Since we only need to test the number of ones in the lower nibble, we
15                        ;mask them by 0F (preserve lower nibble and discard higher nibble)
16  andlw   0x0F          ;in case a user enters a number in the upper digit. Save masked result
17  movwf tempNum         ;in tempNum
18  rrf     tempNum, F    ;rotate tempNum to the right through carry, that is the least
19                        ;significant bit of tempNum (bit0) goes into the C flag of the
20                        ;STATUS register, while the old value of C flag goes into bit 7 of
21                        ;tempNum
```

```
22    btfsc   STATUS, C     ;tests the C flag, if it has the value of 1, increment number of ones and
23    incf    numOfOnes, F;proceed, else proceed without incrementing
24    rrf     tempNum, F
25    btfsc   STATUS, C     ;Same as above
26    incf    numOfOnes, F
27    rrf     tempNum, F
28    btfsc   STATUS, C
29    incf    numOfOnes, F
30    rrf     tempNum, F
31    btfsc   STATUS, C
32    incf    numOfOnes, F
33    nop
34    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 18 to 32 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.

Program Three: Counting the Number of Ones in a Register's Lower Nibble Using a Repetition Structure

```
1             include "p16f84a.inc"
2             cblock 0x20
3                     testNum
4                     tempNum
5             endc
6
7             cblock 0x30
8                     numOfOnes
9                     counter         ;since repetition structures require a counter, one is declared
10            endc
11
12            org 0x00
13            clrf    numOfOnes
14            movlw 0x04            ;counter is initialized by 4, the number of the bits to be tested
```

```
15          movwf counter
16          movf   testNum, W
17          andlw  0x0F
18          movwf tempNum
19  Again
20          rrf      tempNum, F
21          btfsc    STATUS, C
22          incf     numOfOnes, F
23          decfsz   counter, F          ; The contents of register counter are decremented then test :
24           goto     Again              ; if the counter reaches 0, it will skip to "nop" and program ends
25          nop                          ; if the counter is > 0, it will repeat "goto Again"
26          end
```

Introducing the Concept of 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 similar to functions are **Macros** and **Subroutines** ***which are used to implement modular programming.***

Subroutines

Subroutines are the closest equivalent to functions

- Subroutines start with a **Label** giving them a name and end with the instruction **return**

Examples:

```
doMath
        Instruction 1
        Instruction 2
              .
              .
        Instruction n
        return
```

```
Process
        Instruction 1
        Instruction 2
              .
              .
Calculate
        Instruction 7
        Instruction 8
        return
```

This is still one subroutine, no matter the number of labels in between

- Subroutines can be written anywhere in the program after the org and before the end directives
- Subroutines are used in the following way: Call subroutineName
- 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
  Example:

```
Main
        Instruction1
        Instruction2
        Call doMath
        Instruction4
        Instruction5
        Nop
        Nop

doMath
        Instruction35
        Instruction36
        Instruction37
        return
```

So what is the stack and how is it used?

Initially the program executes sequentially; instructions 1 then 2 then 3, when the instruction Call doMath is executed, the program will no longer execute sequentially, instead it will start executing Instructions35, then 36 then 37, when it executes **return**, what will happen? Where will it go and what instruction will be executed?

When the Call doMath instruction is executed, the address of the next instruction (which as you should already know id found in the program counter) Instruction4 is saved in a special memory called the stack. When the return instruction is executed, it reads the **last** address saved in the stack, which is the address of Instruction4 and then continues from there.

----Read section 2.4.1 of the P16F84A datasheet for more information regarding the stack----

Macros

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

```
macroName   macro
                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: macroName
- Each time you use a macro, it will be replaced by its body, refer to the example below. Therefore, the program will execute sequentially, the flow of the program will not change. The Stack is not affected

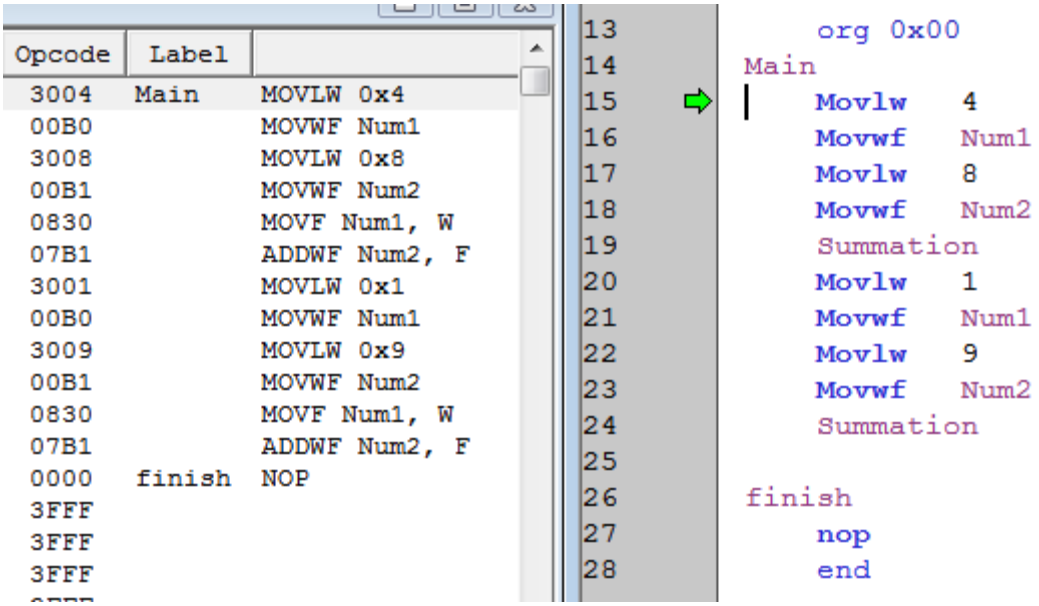
Programs Four and Five

The following simple program demonstrates the differences between using macros and subroutines. They essentially perform the same operation: Num2 = Num1 + Num2

	Example4 using Macro		Example5 using Subroutine
1	include "p16f84a.inc"	1	include "p16f84a.inc"
2		2	
3	cblock 0x30	3	cblock 0x30
4	Num1	4	Num1
5	Num2	5	Num2
6	endc	6	endc
7		7	
8	Summation macro	8	
9	movf Num1, W ;Macro Body	9	
10	addwf Num2, F	10	
11	endm	11	
12		12	
13	org 0x00	13	org 0x00
14	Main	14	Main
15	Movlw 4	15	Movlw 4
16	Movwf Num1	16	Movwf Num1
17	Movlw 8	17	Movlw 8
18	Movwf Num2	18	Movwf Num2
19	Summation	19	Call Summation
20	Movlw 1	20	Movlw 1
21	Movwf Num1	21	Movwf Num1
22	Movlw 9	22	Movlw 9
23	Movwf Num2	23	Movwf Num2
24	Summation	24	Call Summation
25		25	goto finish
26	finish	26	
27	nop	27	Summation
28	end	28	movf Num1, W
		29	addwf Num2, F
		30	return
		31	finish
		32	nop
		33	end

Analyzing the two programs and highlighting the differences

For **both** applications, go to **View → Program Memory**, let's see the differences:

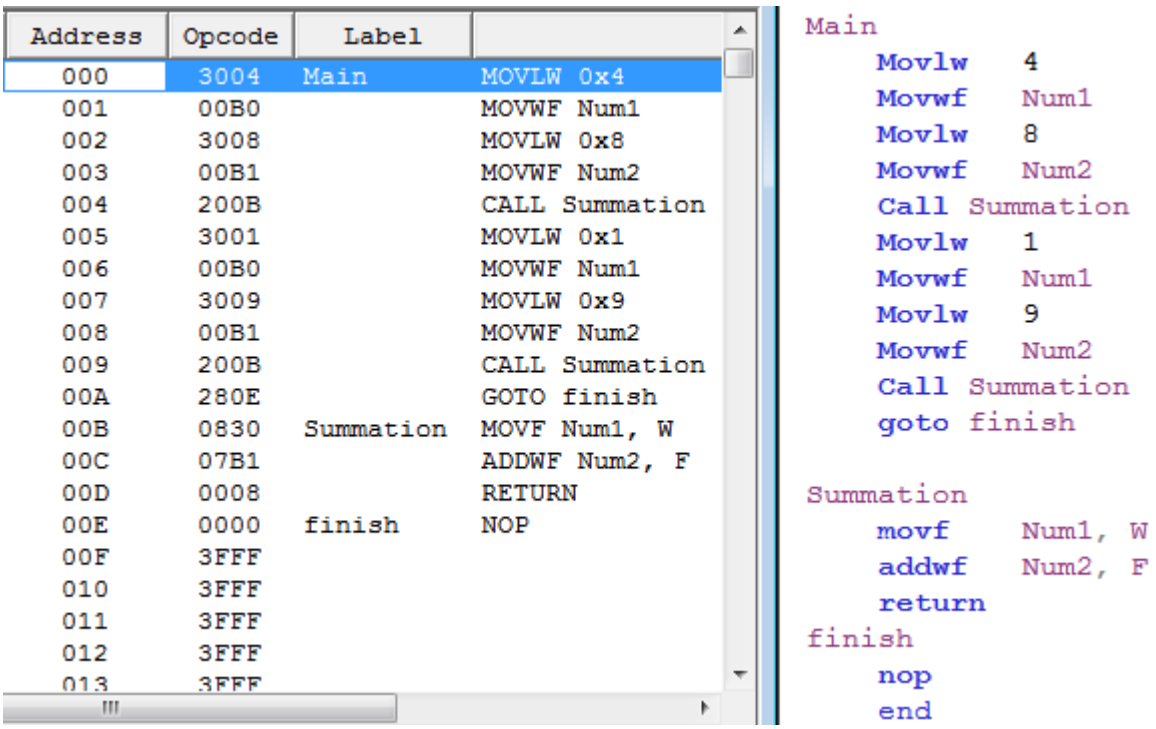


Opcode	Label	
3004	Main	MOVLW 0x4
00B0		MOVWF Num1
3008		MOVLW 0x8
00B1		MOVWF Num2
0830		MOVF Num1, W
07B1		ADDWF Num2, F
3001		MOVLW 0x1
00B0		MOVWF Num1
3009		MOVLW 0x9
00B1		MOVWF Num2
0830		MOVF Num1, W
07B1		ADDWF Num2, F
0000	finish	NOP
3FFF		
3FFF		
3FFF		

```
13          org 0x00
14      Main
15          Movlw   4
16          Movwf   Num1
17          Movlw   8
18          Movwf   Num2
19          Summation
20          Movlw   1
21          Movwf   Num1
22          Movlw   9
23          Movwf   Num2
24          Summation
25
26      finish
27          nop
28          end
```

Figure 1. The example using macros

In the program memory window, notice that the macro name is replaced by its **body**. The instructions movf Num1, W and addwf Num2, F replace the macro name @ lines 19 and 24. Using macros clearly affects the space used by the program as it increases due to code copy.



Address	Opcode	Label	
000	3004	Main	MOVLW 0x4
001	00B0		MOVWF Num1
002	3008		MOVLW 0x8
003	00B1		MOVWF Num2
004	200B		CALL Summation
005	3001		MOVLW 0x1
006	00B0		MOVWF Num1
007	3009		MOVLW 0x9
008	00B1		MOVWF Num2
009	200B		CALL Summation
00A	280E		GOTO finish
00B	0830	Summation	MOVF Num1, W
00C	07B1		ADDWF Num2, F
00D	0008		RETURN
00E	0000	finish	NOP
00F	3FFF		
010	3FFF		
011	3FFF		
012	3FFF		
013	3FFF		

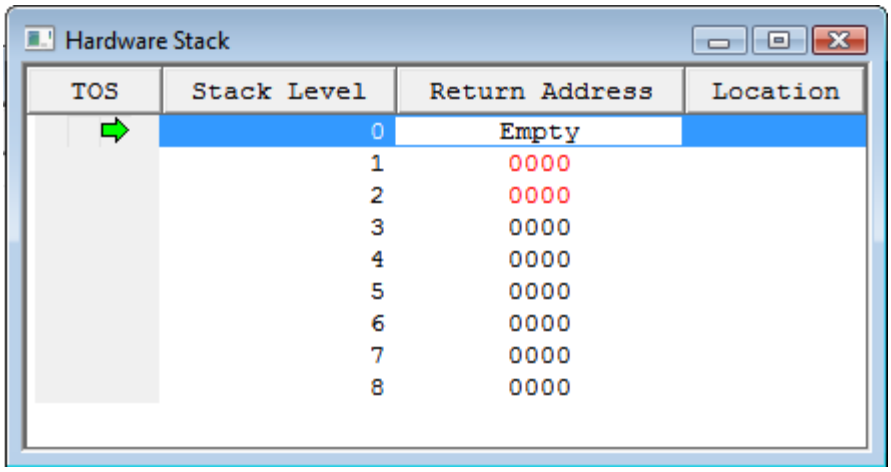
```
Main
    Movlw   4
    Movwf   Num1
    Movlw   8
    Movwf   Num2
    Call Summation
    Movlw   1
    Movwf   Num1
    Movlw   9
    Movwf   Num2
    Call Summation
    goto finish

Summation
    movf    Num1, W
    addwf   Num2, F
    return
finish
    nop
    end
```

Figure 2. The example using subroutines

Now notice that the subroutine is only stored once in the program memory. No code replacement is present.
You can also observe from the program memory that the program utilizing the macro executes sequentially from start to end, while the second program alters the program flow.
For ***Program Two***, do the following:

1. After building the project, go to **View → Hardware Stack**



2. Simulate the program up to the point when the green arrow points to the first Call Summation instruction.

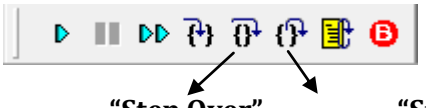
3. Look at the status bar below your MPLAB screen, what is the value of pc (program counter) (Note that the program counter has the address of the next instruction to be executed, that is Call Summation, Remember the instruction the arrow points to is not yet executed)
4. Now execute (use Single step) the Call Summation instruction.
   - ❖ After doing step4, what is the address of PC?
   - ❖ What is now stored at the TOS (Top of Stack)? (Refer to the Hardware Stack window)
   - ❖ How many levels of stack are used?
5. Now, continue simulating the program (subroutine). After executing the return instruction
   - ❖ What is the address of PC?
   - ❖ What is now stored at the TOS?
   - ❖ How many levels of stack are used?
6. Repeat the steps above for the second Call Summation instruction?

   The operation of saving the address on the stack - and any other variables - when calling a subroutine and later retrieving the address – and variables if any - when the subroutine finishes executing is called **context switching.**

Important Notes:

1. Assuming both a macro and a subroutine has the exact same body (same instructions), the execution of the subroutine takes slightly more time due to context switching.
2. You can use macro inside a macro, call a subroutine inside a subroutine, use a macro inside a subroutine and call a subroutine inside a macro

Further Simulation Techniques: Step Over and Step Out



Step Over is used when you want to execute the subroutine as a whole unit without seeing how each individual instruction is executed. It is usually used when you know that that the subroutine executes correctly and you are only interested to see the results.

1. Simulate program two up to the point when the green arrow points to the first Call Summation instruction.
2. Press Step Over, observe how the simulation runs

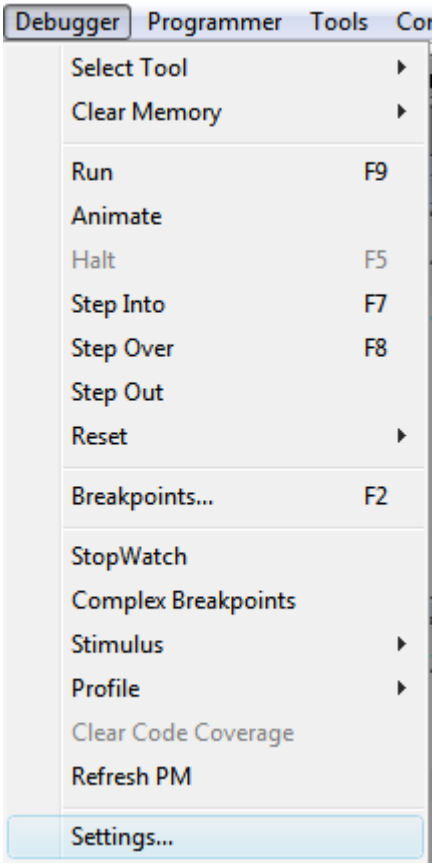
Step Out resembles Step Over, the only difference is that you use it **when you are already inside the subroutine and you want to continue** executing the subroutine as a whole unit without seeing how each **remaining** individual instruction is executed.

1. Simulate the program up to the point when the green arrow points to the first instruction inside the Summation subroutine: movf Num1, W
3. Press Step Out, , observe how the simulation runs

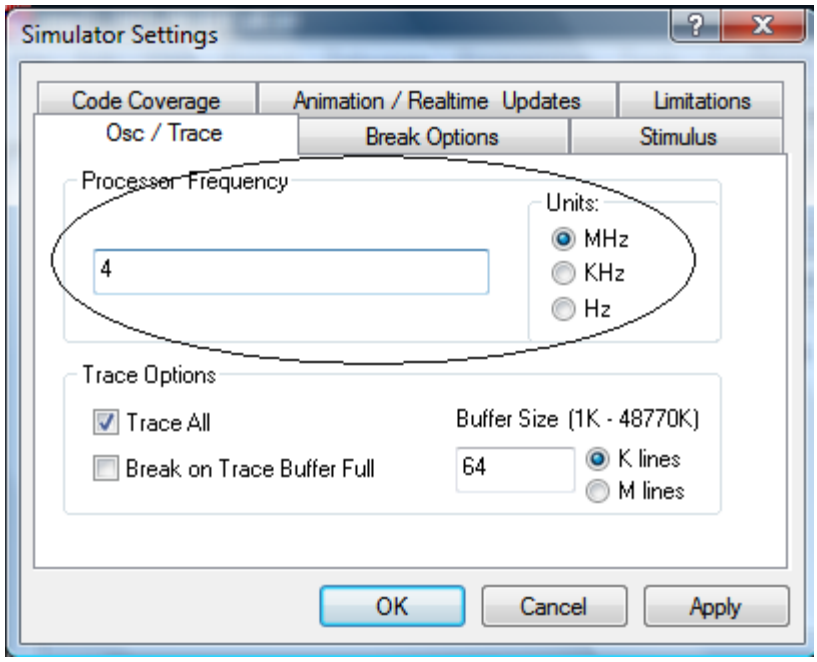
In both cases, the instruction are executed but you only see the end result of the subroutine

Time Calculation

To calculate the total time spent in executing the whole program or a certain subroutine, do the following:



1. Set the oscillator (external clock speed) as follows:

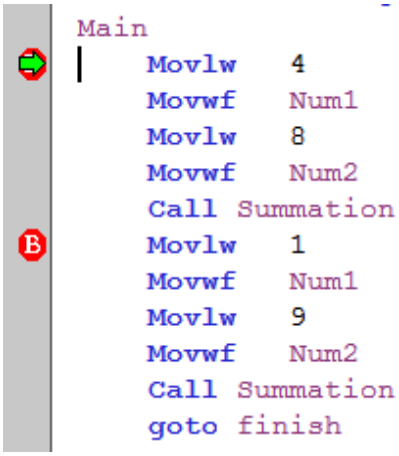


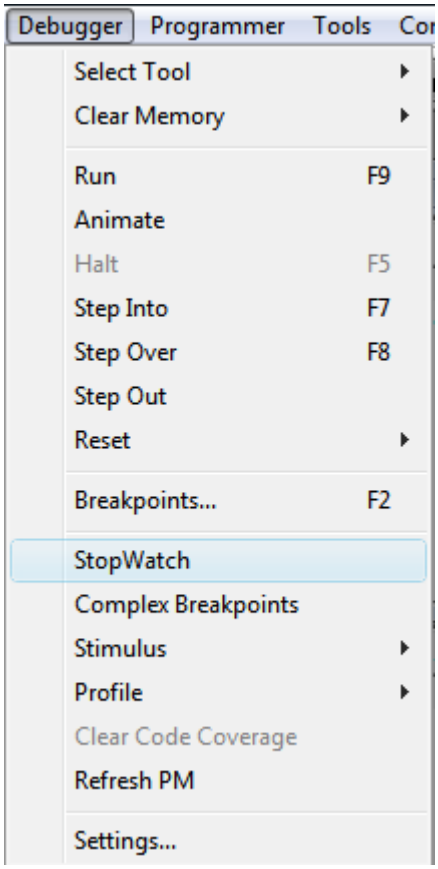
2. Set the processor frequency to 4MHz

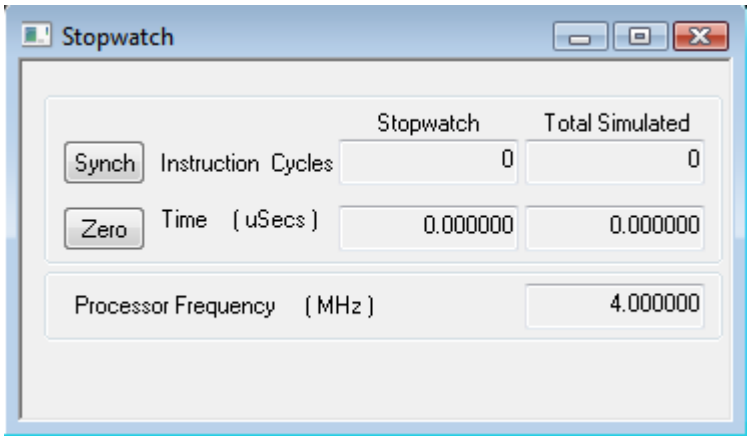
This means that each instruction cycle time is 4MHz/4 = 1MHz and T = 1/f = 1/MHz = 1µs

3. Now set breakpoints at the beginning and end of the code you want to calculate time for

4. Go to **Debugger → Stop Watch**







5. Now run the program, when the pointer stops at the first breakpoint → Press Zero
6. Run the program again. When the pointer reaches the second breakpoint, read the time from the stopwatch. This is the time spent in executing the code between the breakpoints.

Modular Programming

How to think Modular Programming?

Initially, you will have to read and analyze the problem statement carefully, based on this you will have to

1. Divide the problem into several separate tasks,
2. Look for similar required functionality

Non Modular and Modular Programming Approachs: Read the following problem statement

A PIC microcontroller will take as an input two sensor readings and store them in Num1 and Num2, it will then process the values and multiply both by 5 and store them in Num1_5, and Num2_5. At a later stage, the program will multiply Num1 and Num2 by 25 and store them in Num1_25 and Num2_25 respectively.

Analyzing the problem above, it is clear that it has the following functionality:

- Multiply Num1 by 5
- Multiply Num2 by 5
- Multiply Num1 by 25
- Multiply Num2 by 25

As you already know, we do not have a multiply instruction in the PIC 16F84A instruction set, so we do it by addition since:

2 x 3 = 2 + 2 + 2 ; add 2 three times

7 x 9 = 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 ; add 7 nine times

So we write a loop as follows (example 4 x 9, add four nines), initially one nine is placed in W then we construct a loop to add the remaining 8 nines:

```
        movlw .8            ; because we put the first 4 in W, then we add the remaining 8 fours to it
        movwf counter
        movf   temp, w      ; 1st four in W
add
        addwf  temp, w
        decfsz  counter, f  ; decrement counter, if not zero keep adding, else continue
        goto    add
        ; continue with code
```

A Non Modular Programming Approach		**Modular Programming Approach**	
Write multiply code for each operation above		Write one "Multiply by 5" code, use it two times Write one "Multiply by 25" code, use it two times Note that you do not need to write the "Multiply by 25" code from scratch, since 25 is 5x5, you can simply use "Multiply by 5" two times!	
Code lines: 38		Code lines: 27	
get Num1	1	get Num1	1
Write whole code to multiply Num1 by 5	7	call "multiply by 5" function	1
Store in Num1_5	1	Store in Num1_5	1
get Num2	1	get Num2	1
Write whole code to multiply Num2 by 5	7	call "multiply by 5" function	1
Store in Num2_5	1	Store in Num2_5	1
get Num1	1	get Num1	1
Write whole code to multiply Num1 by 25	7	call "multiply by 25" function	1
Store in Num1_25	1	Store in Num1_25	1
get Num2	1	get Num2	1
Write whole code to multiply Num2 by2 5	7	call "multiply by 25" function	1
Store in Num2_25	1	Store in Num2_25	1
goto finish	1	goto finish	1
nop	1	nop	1
		A single Multiply by 5 function	8
		A single Multiply by 5 function	5

```
include "p16f84a.inc"
cblock 0x30
        Num1
        Num2
        Num1_5
        Num2_5
        Num1_25
        Num2_25
        temp
        counter
endc

        org 0x00
Main
        movf    Num1, w    ;Num1 x 5
        movwf temp
        movlw .4
        movwf counter
        movf    temp, w
add1
        addwf  temp, w
        decfsz  counter, f
        goto    add1
        movwf Num1_5


        movf    Num2, w    ;Num2 x 5
        movwf temp
        movlw .4
        movwf counter
        movf    temp, w
add2
        addwf  temp, w
        decfsz  counter, f
        goto    add2
        movwf Num2_5

        movf    Num1, w    ;Num1 x 25
        movwf temp
        movlw .24
        movwf counter
        movf    temp, w
add3
        addwf  temp, w
        decfsz  counter, f
        goto    add3
```

```
include "p16f84a.inc"
cblock 0x30
        Num1
        Num2
        Num1_5
        Num2_5
        Num1_25
        Num2_25
        temp
        counter
endc

        org 0x00
Main
        movf    Num1, w    ;Num1 x 5
        call    Mul5
        movwf Num1_5

        movf    Num2, w    ;Num2 x 5
        call    Mul5
        movwf Num2_5

        movf    Num1, w    ;Num1 x 25
        call    Mul25
        movwf Num1_25

        movf    Num2, w    ;Num2 x 25
        call    Mul25
        movwf Num2_25
        goto    finish

Mul5
        movwf temp
        movlw .4
        movwf counter
        movf    temp, w
add
        addwf  temp, w
        decfsz  counter, f
        goto    add
        return

Mul25
        movwf temp
        call Mul5
        movwf temp
```

```
        movwf Num1_25

        movf    Num2, w    ;Num2 x 25
        movwf temp
        movlw .24
        movwf counter
        movf    temp, w
add4
        addwf  temp, w
        decfsz  counter, f
        goto    add4
        movwf Num2_25
        goto    finish
finish
        nop
        end
```

```
        call Mul5
        return
finish
        nop
        end
```

Notes on passing parameters to subroutines

Subroutines and macros are **general** codes; they work on many variables and generate results. So how do we tell the macro/subroutine that we want to work on this specific variable?

We have two approaches:

Place the input at the working register
Take the output from the working register

Example:

```
Main
   Movlw 03                ;input to W
   Call     MUL_by4
   Movwf Result1           ;output from W
   Movlw 07                ;input to W
   Call     MUL_by4
   Movwf Result2           ;output from W
   Nop
      .
      .
MUL_by4
   Movwf temp
   Rlf      temp,F
   Rlf      temp, F
   Movf   temp, W          ;place result in W
   Return
```

Store the input(s) in external variables
Load the output(s) from external variables

Example:

```
   Movf    Num1, W        ;load Num with Num1
   Movwf Num
   Call     MUL_by4
   Movf     Result, W      ;read the result and store
   Movwf Result1           ;it in Result1
   Movf    Num2, W        ;load Num with Num2
   Movwf Num
   Call     MUL_by4
   Movf    Result, W      ;read the result and store
   Movwf Result2           ;it in Result2

MUL_by4
   Rlf      Num,F
   Rlf      Num, W
   Movwf Result            ;place result in W
   Return
```

In this approach, the MUL_by4 subroutine takes the input from W (movwf), processes it then places the result back in W. Notice that we initially load W by the numbers we work on (here 03 and 07) then we take their values from W and save them in Result1 and Result2 respectively	In this approach the MUL_by4 subroutine expects to find the input in Num and saves the output in Result. Therefore, before calling the subroutine we load Num by the value we want (here Num1) and then take the value from Result and save it in Result1. The same is repeated for Num2
This approach is useful when the subroutine/macro has only one input and one output	This approach is useful when the subroutine/macro takes many inputs and produces multiple outputs

Special types of subroutines: Look up tables

Look up tables are a special type of subroutines which are used to retrieve values depending on the input they receive. They are invoked in the same as any subroutine: Call tableName
They work on the basis that they change the program counter value and therefore alter the flow of instruction execution
The **retlw** instruction is a **return** instruction with *the benefit that it returns a value in W* when it is executed.

Syntax:

```
lookUpTableName
        addwf   PCL, F   ;add the number found in the program counter to PCL (Program counter)
        nop
        retlw   Value            ;if W has 1, execute this
        retlw   Value            ;if W has 2, execute this
        retlw   Value
        …
        retlw   Value
```

Value can be in any format: decimal, hexadecimal, octal, binary and ASCII. It depends on the application you want to use this look-up table in.

Program Six: Displaying the 26 English Alphabets

This program works as follows:

Counter is loaded with the number 1 because we want to get the first letter of the alphabet, when we call the look-up table, it will retrieve the letter 'A'. The counter is incremented by 1 and then checked if we have reached the 26th letter of the alphabet (27 – the initial 1), if not we proceed to display the second letter 'B' and the third 'C' and so on. When we have displayed all the alphabets, counter will have the value 27 after which the program exits.

```
1   include "p16f84a.inc"
2   cblock  0x25
3           counter                 ;holds the number of Alphabet displayed
4           Value                   ;holds the alphabet value
5   endc
6           org 0x00
7   Main
8           movlw 1                 ;Initially no alphabet is displayed
9           movwf counter
10  Loop
11          movf    counter, W
12          call    Alphabet        ;display Alphabet
13          movwf Value
14          incf    counter, F      ;Each time, increment the counter by 1
15          movf    counter, w      ;if counter reaches 27, exit loop else continue
16          sublw   .27
17          btfss   STATUS, Z
18          goto    Loop
19          goto    finish
20  Alphabet
21          addwf  PCL, F
22          nop
23          retlw   'A'
24          retlw   'B'
25          retlw   'C'
26          retlw   'D'
27          retlw   'E'
28                .
29                .
30          retlw   'Z'
31  finish
32          nop
33          end
```

1. Complete the look-up table above with the missing alphabet
2. Add both counter and value to the watch window.
3. Place a breakpoint @ instruction 14: `incf counter, F`
4. Run the program, keep pressing run and observe the values of the variables in the Watch window

Appendix A: Documenting your program

It is a good programming practice to document your program in order to make it easier for you or others to read and understand it. For that reason we use comments. A proper way of documenting your code is to write **a functional comment,** which is a comment that **describes the function** of one or a set of instructions. Comments are defined after a semicolon (;) and are **not** read by MPLAB IDE

```
BSF STATUS, RP0
; Switch to Bank 1                          Good comment                √
; Set the RP0 bit in the Status Register to 1   Bad Comment, no new added info   X
```

How to professionally document your program?

At the beginning of your program, you are encouraged to add the following header which gives an insight to your code, its description, creator, version, date of last revision, etc… Most importantly, it is encouraged to document the necessary connections and classify them as input/output.

```
;*****************************************************************************************
; * Program name: Example Program
; * Program description: This program …….
; *
; * Program version: 1.0
; * Created by Embedded lab engineers
; * Date Created: September 1st, 2008
; * Date Last Revised: September 16th, 2008
;*****************************************************************************************
; * Inputs:
; *       Switch 0 (Emergency) to RB0 as interrupt
; *       Switch 1 (Start Motor) to RB1
; *       Switch 2 (Stop Motor) to RB2
; *       Switch 3 (LCD On) to RB3
; * Outputs:
; *       RB4 to Motor
;*       RB5 to Green LED (Circuit is powered on)
;*****************************************************************************************
```

1. Your code declarations go here: *includes, equates, cblocks, macros, origin, etc…*
2. Your code goes here…
3. When using subroutines/macros, it is advised to add a header like this one before each to properly document and explain the function of the respected subroutine/macro.

```
;*****************************************************************************************
;* Subroutine Name: ExampleSub
;* Function: This subroutine multiplies the value found in the working register by 16
;* Input: Working register
;* Output: Working register * 16
;***************************************
```

Appendix B: Instruction Listing

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	1,2
CLRF	f	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	-	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	1,2
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00 1011 dfff ffff		1,2,3
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00 1111 dfff ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1,2
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	1,2
MOVWF	f	Move W to f	1	00 0000 lfff ffff		
NOP	-	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	-	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	-	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	-	Go into standby mode	1	00 0000 0110 0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	