



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



1

Experiment 1: MPLAB and Instruction Set Analysis 1



Objectives

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

- ❖ The MOV instructions
- ❖ Writing simple codes, compiling the project and Code simulation
- ❖ The concept of bank switching
- ❖ The MPASM directives
- ❖ Microcontroller Flags
- ❖ Arithmetic and logical operations

Pre-lab requirements

Before starting this experiment, you should have already acquired the MPLAB software and the related PIC datasheets from drive D on any of the lab PC's. You are encouraged to install the latest version of MPLAB (provided in the lab) especially if you have Windows Vista

Starting up with instructions

Movement instructions

You should know by now that most PIC instructions (logical and arithmetic) work through the working register “W”, that is one of their operands must always be the working register “W”, the other operand might be either a constant or a memory location. Many operations store their result in the working register; therefore we can conclude that we need the following movement operations:

1. Moving constants to the working register (Loading)
2. Moving values from the data memory to the working register (Loading)
3. Moving values from the working register to the data memory (Storing)

INSTRUCTIONS ARE CASE INSENSITIVE: YOU CAN WRITE IN EITHER SMALL OR CAPITAL LETTERS

- ❖ **MOVLW**: moves a literal (constant) to the working register (final destination). The constant is specified by the instruction. You can directly load constants as decimal, binary, hexadecimal, octal and ASCII. The following examples illustrate:

DEFAULT INPUT IS HEXADECIMAL

1. **MOVLW 05** : moves the constant 5 to the working register
2. **MOVLW 10** : moves the constant **16** to the working register.
3. **MOVLW 0xAB** : moves the constant **AB_h** to the working register
4. **MOVLW H'7F'** : moves the constant **7F_h** to the working register
5. **MOVLW CD** : **WRONG**, if a hexadecimal number starts with a character, you should write it as **0CD** or **0xCD** or **H'CD'**
6. **MOVLW d'10'** : moves the **decimal** value 10 to the working register.
7. **MOVLW .10** : moves the **decimal** value 10 to the working register.
8. **MOVLW b'10011110'** : moves the **binary** value 10011110 to the working register.
9. **MOVLW O'76'** : moves the **octal** value 76 to the working register.
10. **MOVLW A'g'** : moves the **ASCII** value **g** to the working register.

-
- ❖ **MOVWF**: **COPIES** the value found in the working register into the data memory, **but to which location?** The location is specified along with the instruction and according to the memory map.

So what is the memory map?

A memory map shows all available registers (in data memory) of a certain PIC along with their addresses, it is organized as a table format and has two parts:

1. **Upper part:** which lists all the Special Function Registers (SFR) in a PIC, these registers normally have specific functions and are used to control the PIC operation
2. **Lower part:** which shows the General Purpose Registers (GPR) in a PIC; GPRs are data memory locations that the user is free to use as he wishes.

Memory Maps of different PICs are different. Refer to the datasheets for the appropriate data map

Examples:

1. MOVWF 01 : COPIES the value found in W to TMR0
2. MOVWF 05 : COPIES the value found in W to PORTA
3. MOVWF 0C : COPIES the value found in W to a GPR (location 0C)
4. MOVWF 32 : COPIES the value found in W to a GPR (location 32)
5. MOVWF 52 : **WRONG**, out of data memory range of the PIC 16F84a (GPR range is from 0C-4F and 8C to CF)

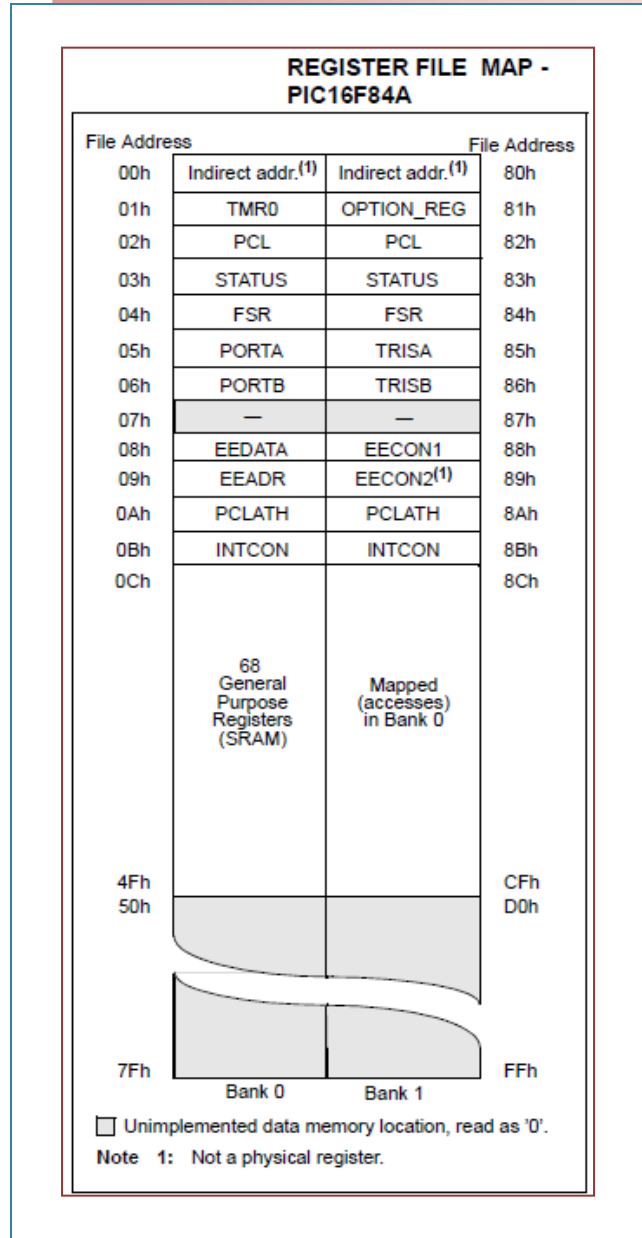
❖ **MOVF:** COPIES a value found in the data memory to the **working register OR to itself**.

Therefore we expect a second operand to specify whether the destination is the working register or the register itself.

For now: a 0 means the W, a 1 means the register itself.

Examples:

1. MOVF 05, 0 : **copies** the content of PORTA to the working register
2. MOVF 2D, 0 : **copies** the content of the GPR 2D to the working register
3. MOVF 05, 1 : **copies** the content of PORTA to itself
4. MOVF 2D, 1 : **copies** the content of the GPR 2D to itself

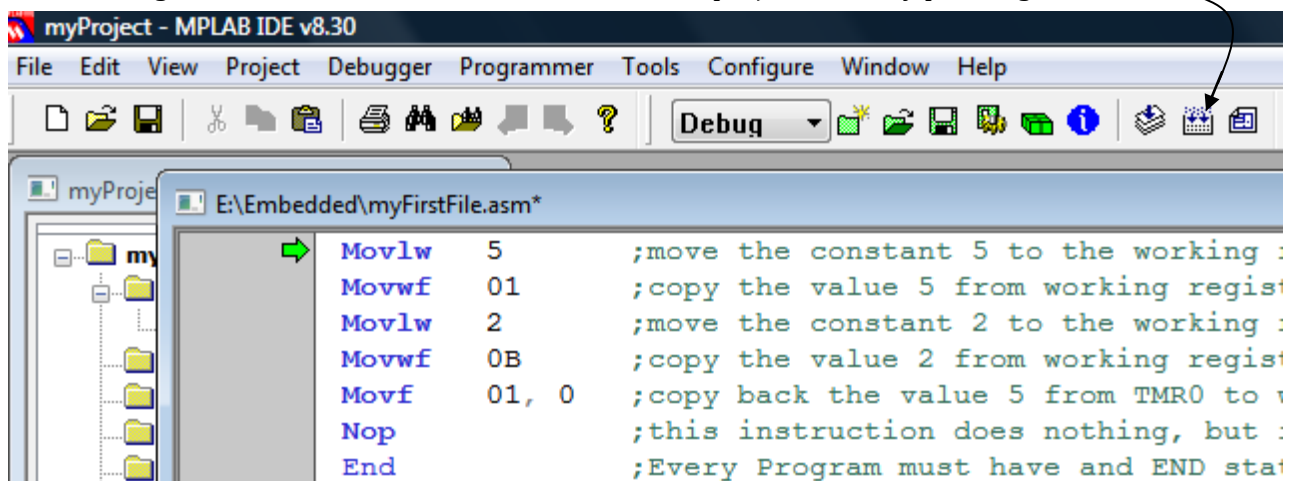


Now we will simulate a program in MPLAB and check the results

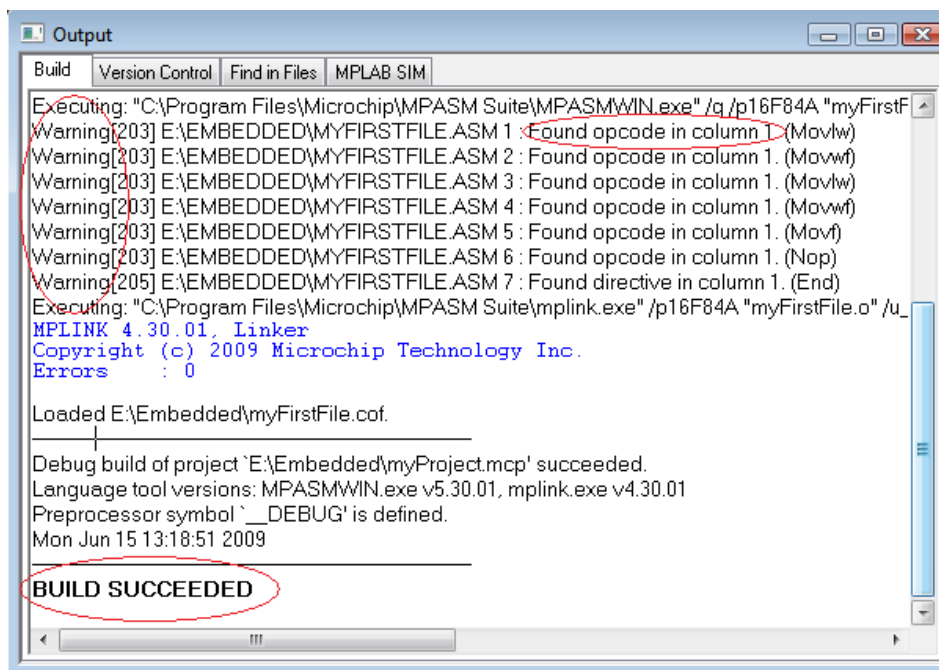
In MPLAB write the following program:

```
Movlw    5      ; move the constant 5 to the working register
Movwf    01     ; copy the value 5 from working register to TMR0 (address 01)
Movlw    2      ; move the constant 2 to the working register
Movwf    0B     ; copy the value 2 from working register to INTCON (address 0B)
Movf     01, 0  ; copy back the value 5 from TMR0 to working register
Nop      ; this instruction does nothing, but it is important to write for now
End      ; every program must have an END statement
```

After writing the above instructions we should build the project, do so by pressing **build**



An output window should show: **BUILD SUCCEEDED**



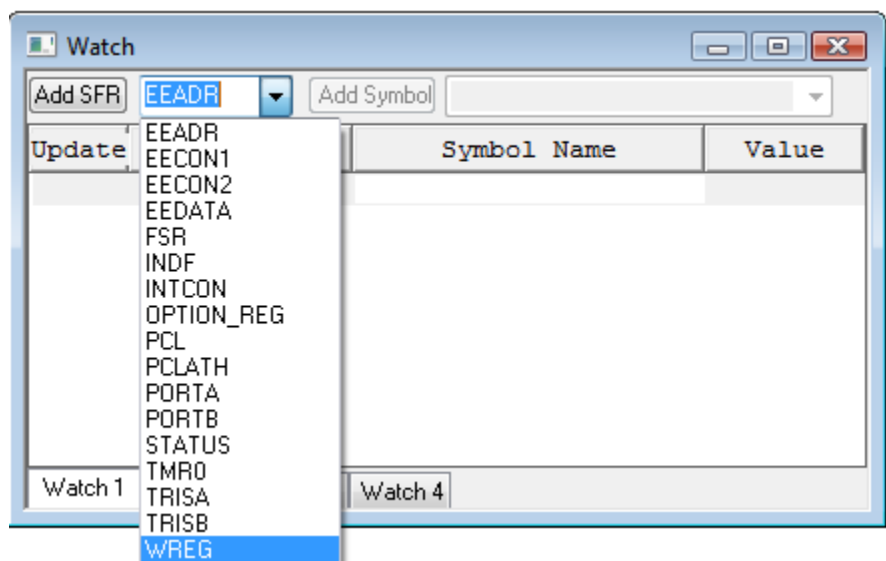
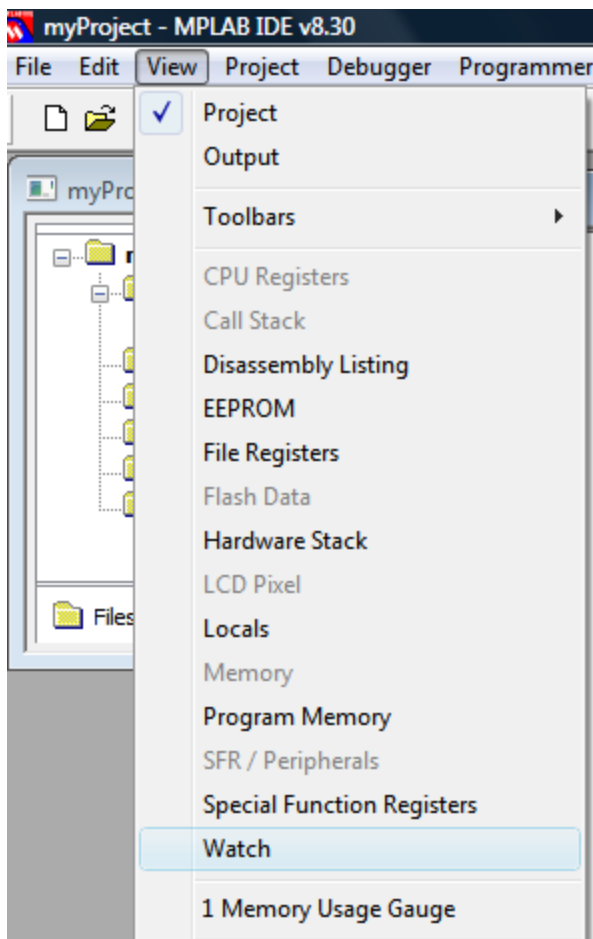
BUILD SUCCEED DOES NOT MEAN THAT YOUR PROGRAM IS CORRECT, IT SIMPLY MEANS THAT THERE ARE NO SYNTAX ERRORS FOUND, SO WATCH OUT FOR ANY LOGICAL ERRORS YOU MIGHT MAKE.

Notice that there are several warnings after building the file, warnings do not affect the execution of the program but they are worth reading. This warning reads: “Found opcode in column 1”, column 1 is reserved for labels; however, we have written instructions (opcode) instead thus the warning.

TO SOLVE THIS WARNING SIMPLY TYPE FEW BLANK SPACES BEFORE EACH INSTRUCTION OR PRESS TAB

Preparing for simulation

Go to View Menu → Watch

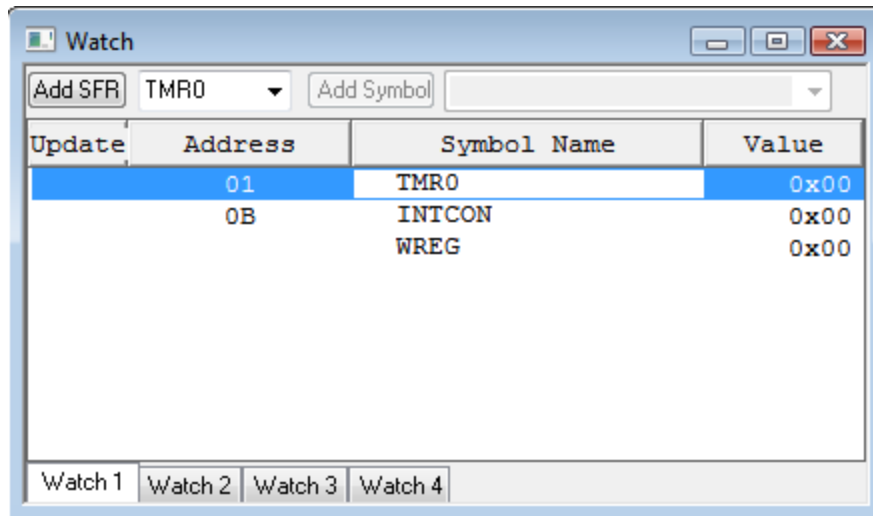


From the drop out menu choose the registers we want to watch during simulation and click ADD SFR for each one

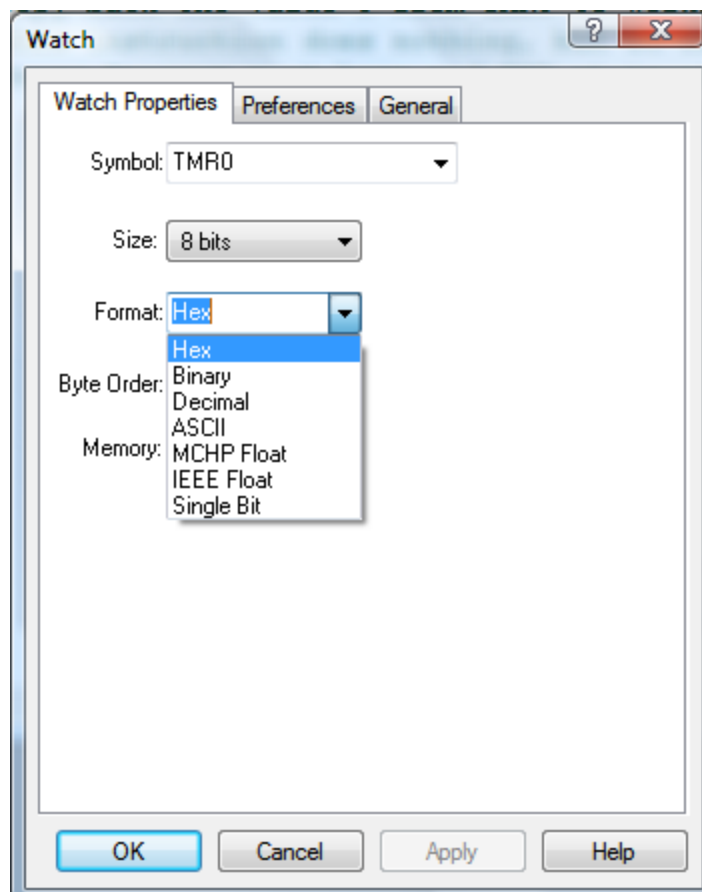
Add the following:

- WREG: working register
- TMR0
- INTCON

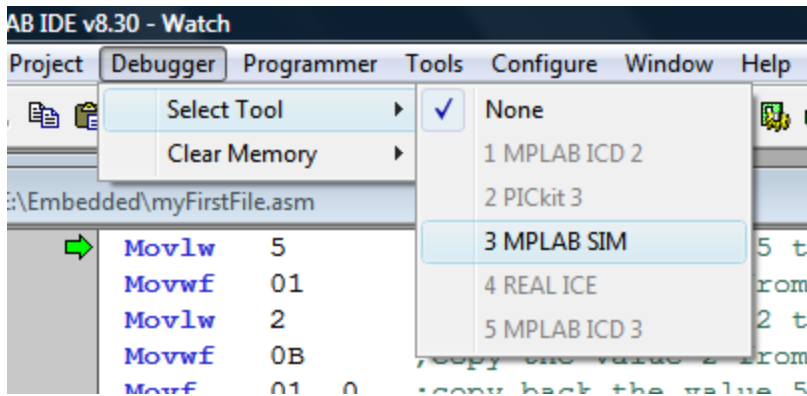
You should have the following:



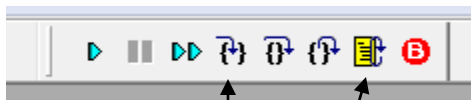
Notice that the default format is in hexadecimal, to change it (if you need to) simply right-click on the row → **Properties** and choose the new format you wish.




From the **Debugger Menu** → choose **Select Tool** → then **MPLAB SIM**



Now new buttons will appear in the toolbar:

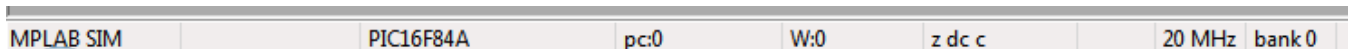


Step Into Reset

1. To begin the simulation, we will start by resetting the PIC; do so by pressing the yellow reset button. A green arrow  will appear next to the first instruction.

The green arrow means that the program counter is pointing to this instruction *which has not been executed yet*.

Notice the status bar below:



Keep an eye on the value of the program counter (pc: initially 0), see how it changes as we simulate the program

2. Press the “Step Into” button one at a time and check the Watch window each time an instruction executes; keep pressing “Step Into” until you reach the **NOP instruction** then **STOP**.

Compare the results as seen in the Watch window with those expected.

Directives

Directives are not instructions. They are **assembler commands** that appear in the source code but are not usually translated directly into opcodes. They are used to control the **assembler**: its input, output, and data allocation. They are not converted to machine code (.hex file) and therefore not downloaded to the PIC.

The “END” directive

If you refer to the Appendix at the end of this experiment, you will notice that there is no end instruction among the PIC 16 series instructions, so what is “END”?

The “END” command is a directive which tells the MPLAB IDE that we have finished our program. It has nothing to do with neither the actual program nor the PIC.

The END should always be the last statement in your program

Anything which is written after the end command will not be executed and any variable names will be undefined.

Making your program easier to understand: the “equ” and “include” directives

As you have just noticed, it is difficult to write, read, debug or understand programs while dealing with memory addresses as numbers. Therefore, we will learn to use new directives to facilitate program reading.

The “EQU” directive

The equate directive is used to **assign** labels to numeric values. They are used to *DEFINE CONSTANTS* or to *ASSIGN NAMES TO MEMORY ADDRESSES OR INDIVIDUAL BITS IN A REGISTER* and then use the name instead of the numeric address.

```
Timer0    equ 01
Intcon     equ 0B
Workrg     equ 0
Movlw     5                ; move the constant 5 to the working register
Movwf     Timer0          ; copy the value 5 from working register to TMR0 (address 01)
Movlw     2                ; move the constant 2 to the working register
Movwf     Intcon           ; copy the value 2 from working register to INTCON (address 0B)
Movf      Timer0, Workrg   ; copy back the value 5 from TMR0 to working register
Nop                          ; this instruction does nothing, but it is important to write it for now
End
```

Notice how it is easier now to read and understand the program, you can directly know the actions executed by the program without referring back to the memory map by simply giving each address a name at the beginning of your program.

DIRECTIVES THEMSELVES ARE NOT CASE-SENSITIVE BUT THE LABELS YOU DEFINE ARE. SO YOU MUST USE THE NAME AS YOU HAVE DEFINED IT SINCE IT IS CASE-SENSITIVE.

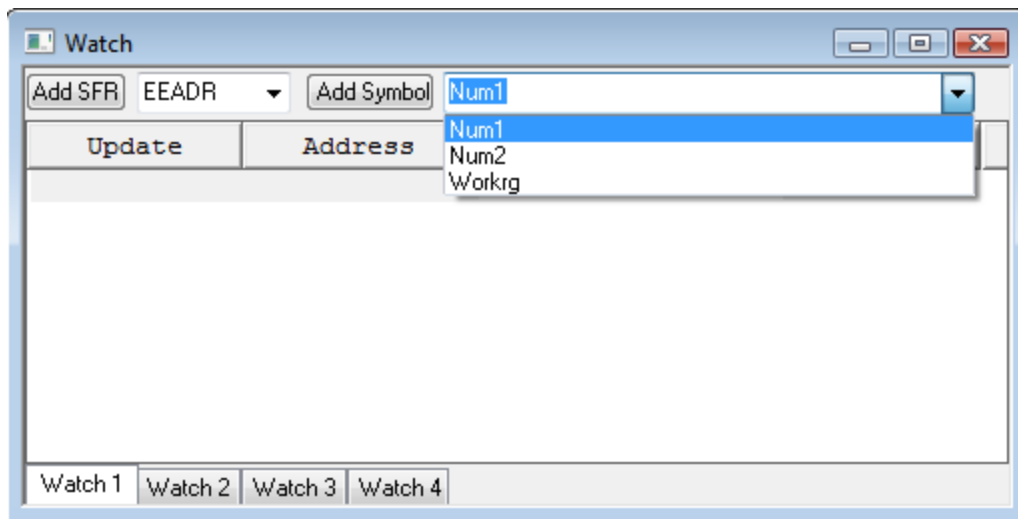
As you have already seen, the GPRs in a memory map (lower part) do not have names as the SFRs (Upper part), so it would be difficult to use their addresses each time we want to use them. Here, the “*equate*” statement proves helpful.

```

Num1      equ 20           ;GPR @ location 20
Num2      equ 40           ;GPR @ location 40
Workrg    equ 0
Movlw      5                   ; move the constant 5 to the working register
Movwf      Num1                ; copy the value 5 from working register to Num1 (address 20)
Movlw      2                   ; move the constant 2 to the working register
Movwf      Num2                ; copy the value 2 from working register to Num2 (address 40)
Movf       Num1, Workrg        ; copy back the value 5 from Num1 to working register
Nop                            ; this instruction does nothing, but it is important to write it for now
End

```

When simulating the above code, you need to add Num1, Num2 to the watch window, however, since Num1 and Num2 are not SFRs but GPRs, you will not find them in the drop out menu of the “Add SFR”, instead you will find them in the drop out menu of the “Add symbol”.



The “INCLUDE” directive

Suppose we are to write a huge program that uses all registers. It will be a tiresome task to define all Special Function Registers (SFR) and bit names using “*equate*” statements. Therefore we use the include directive. The include directive calls a file which has all the *equate* statements defined for you and ready to use, its syntax is

```
#include "PXXXXXXX.inc"    where XXXXXX is the PIC part number
```

↓
Older version of include without #, still supported.

Example: **#include “P16F84A.inc”**

The only **condition** when using the include directive is to use the names as Microchip defined them which are **ALL CAPITAL LETTERS** and **AS WRITTEN IN THE DATA SHEET**. If you don't do so, the MPLAB will tell you that the variable is undefined!

#include "P16F84A.inc"

```
Movlw    5                ; move the constant 5 to the working register
Movwf    TMR0             ; copy the value 5 from working register to TMR0 (address 01)
Movlw    2                ; move the constant 2 to the working register
Movwf    INTCON           ; copy the value 2 from working register to INTCON (address 0B)
Movf     TMR0, W          ; copy back the value 5 from TMR0 to working register
Nop      ; this instruction does nothing, but it is important to write it for now
End
```

The "Cblock" directive

You have learnt that you can assign GPR locations names using the equate statements to facilitate dealing with them. Though this is correct, it is not recommended by Microchip as a good programming practice. Instead you are instructed to use cblocks when defining and declaring GPRs. So then, what is the use of the "equ" directive?

From now on, follow these two simple programming rules:

1. The **"EQU"** directive is used to define **constants**
2. The **"cblock"** is used to define **variables** in the data memory.

The cblock defines variables in sequential locations, see the following declaration

```
Cblock 0x35
```

```
    VarX
```

```
    VarY
```

```
    VarZ
```

```
endc
```

Here, VarX has the starting address of the cblock, which is 0x35, VarY has the sequential address 0x36 and VarZ the address of 0x37

What if I want to define variable at locations which are not sequential, that is some addresses are at 0x25 others at 0x40?

Simply use another cblock statement, you can add as many cblock statements as you need

The Origin "org" directive

The origin directive is used to place the instruction **which exactly comes after it** at the location it specifies.

Examples:

```
Org    0x00
Movlw  05      ;This instruction has address 0 in program memory
Addwf  TMR0    ;This instruction has address 1 in program memory
Org    0x04    ;Program memory locations 2 and 3 are empty, skip to address 4 where it contains
Addlw  08      ;this instruction
```

```
Org    0x13    ;WRONG, org only takes even addresses
```

In This Course, Never Use Any Origin Directives Except For Org 0x00 And 0x04, Changing Instructions' Locations In The Program Memory Can Lead To Numerous Errors.

The Concept of Bank Switching

Write, build and simulate the following program in your MPLAB editor. This program is very similar to the ones discussed above but with a change of memory locations.

```
#include "P16F84A.inc"
```

```
Movlw    5      ; move the constant 5 to the working register
Movwf    TRISA  ; copy the value 5 from working register to TRISA (address 85)
Movlw    2      ; move the constant 2 to the working register
Movwf    OPTION_REG ; copy 2 from working register to OPTION_REG (address 81)
Movf     TRISA, W ; copy back the value 5 from TRISA to working register
Nop      ; this instruction does nothing, but it is important to write it for now
End
```

After simulation, you will notice that both TRISA and OPTION_REG were not filled with the values 5 and 2 respectively! But why?

Notice that the memory map is divided into two columns, each column is called a bank, here we have two banks: bank 0 and bank 1. In order to access bank 1, we have to switch to that bank first and same for bank 0. But how do we make the switch?

Look at the details of the STATUS register in the figure below, there are two bits RP0 and RP1, these bits control which bank we are in:

- ❖ If RP0 is 0 then we are in bank 0
- ❖ If RP0 is 1 then we are in bank 1

We can change RP0 by using the bcf/bsf instructions

- ❖ BCF STATUS, RP0 → RP0 in STATUS is 0 → switch to bank 0
- ❖ BSF STATUS, RP0 → RP0 in STATUS is 1 → switch to bank 1

BCF: *Bit Clear File Register (makes a specified bit in a specified file register a 0)*

BSF: *Bit Set File Register (makes a specified bit in a specified file register a 1)*

Try the program again with the following change and check the results:

```
#include "P16F84A.inc"
```

```
BSF      STATUS, RP0
Movlw     5           ; move the constant 5 to the working register
Movwf    TRISA       ; copy the value 5 from working register to TRISA (address 85)
Movlw     2           ; move the constant 2 to the working register
Movwf    OPTION_REG  ; copy 2 from working register to OPTION_REG (address 81)
Movf     TRISA, W    ; copy back the value 5 from TRISA to working register
BCF      STATUS, RP0
Nop
End
```

The "Banksel" directive

When using medium-range and high-end microcontrollers, it will be a hard task to check the memory map for each register we will use. Therefore the **BANKSEL** directive is used instead to simplify this issue. This directive is a command to the assembler and linker to generate bank selecting code to set the bank to the bank containing the designated *label*

Example:

BANKSEL TRISA will be replaced by the assembler, which will automatically know which bank the register is in and generate the appropriate bank selection instructions:

```
Bsf STATUS, RP0
Bcf STATUS, RP1
```

In the PIC16F877A, there are four banks; therefore you need two bits to make the switch between any of them. An additional bit in the STATUS register is RP1, which is used to make the change between the additional two banks.

One drawback of using **BANKSEL** is that it always generates two instructions even when the switch is between bank0 and bank1 which only requires changing RP0. We could write the code above in the same manner using **Banksel**

```
#include "P16F84A.inc"
```

```
Banksel   TRISA
Movlw     5           ; move the constant 5 to the working register
Movwf    TRISA       ; copy the value 5 from working register to TRISA (address 85)
Movlw     2           ; move the constant 2 to the working register
Movwf    OPTION_REG  ; copy 2 from working register to OPTION_REG (address 81)
Movf     TRISA, W    ; copy back the value 5 from TRISA to working register
Banksel   PORTA
Nop
End
```

Check the program memory window to see how **BANKSEL is replaced in the above code and the difference in between the two codes in this page.**

FLAGS

The PIC 16 series has three indicator flags found in the STATUS register; they are the C, DC, and Z flags. See the description below. Not all instructions affect the flags; some instructions affect some of the flags while others affect all the flags. Refer to the Appendix at the end of this experiment and review which instructions affect which flags.

The **MOVLW** and **MOVWF** do not affect any of the flags while the **MOVF** instruction affects the zero flag. Copying the register to itself does make sense now because if the file has the value of zero the zero flag will be one. Therefore the MOVF instruction is used to affect the zero flag and consequently know if a register has the value of 0. (Suppose you are having a down counter and want to check if the result is zero or not)

STATUS REGISTER

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC ⁽¹⁾	C ⁽¹⁾
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared
		x = Bit is unknown

bit 6-5 **RP<1:0>**: Register Bank Select bits (used for direct addressing)

00 = Bank 0
 01 = Bank 1
 10 = Bank 2
 11 = Bank 3

bit 2 **Z**: Zero bit

1 = The result of an arithmetic or logic operation is zero
 0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit Carry/Borrow bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)⁽¹⁾

1 = A carry-out from the 4th low-order bit of the result occurred
 0 = No carry-out from the 4th low-order bit of the result

bit 0 **C**: Carry/Borrow bit⁽¹⁾ (ADDWF, ADDLW, SUBLW, SUBWF instructions)⁽¹⁾

1 = A carry-out from the Most Significant bit of the result occurred
 0 = No carry-out from the Most Significant bit of the result occurred

Note 1: For Borrow, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high-order or low-order bit of the source register.

Types of Logical and Arithmetic Instructions and Result Destination

The PIC16 series logical and arithmetic instructions are easy to understand by just reading the instruction, for from the name you readily know what this instruction does. There are the ADD, SUB, AND, XOR, IOR (the ordinary **Inclusive OR**). They only differ by their operands and the result destination. The following table illustrates:

	Type I – Literal Type	Type II – File Register Type
<i>Syntax</i>	xxx LW <i>k</i> where <i>k</i> is constant	xxx WF <i>f, d</i> where <i>f</i> is file register and <i>d</i> is the destination (F, W)
<i>Instructions</i>	Addlw, sublw, andlw, iorlw and xorlw	Addwf, subwf, andwf, iorwf, xorwf
<i>Operands</i>	<ol style="list-style-type: none"> 1. A literal (given by the instruction) 2. The working register 	<ol style="list-style-type: none"> 1. A file register in the data memory (either SFR or GPR) 2. The working register
<i>Result destination</i>	The working register only	Two Options: <ol style="list-style-type: none"> 1. W: the Working register 2. F: The same File given in the instruction
<i>How does it work?</i>	W = L operation W	F = F operation W The value of F is overwritten by the result, original value lost W = F operation W The value of F is the original value, result stored in working register instead
The order is important in the subtract operation		
<i>Examples</i> (assuming you are using the include statement and appropriate equ statements for defining GPRs)	xorlw 0BB $W = W \wedge 0BB$ sublw .85 $W = 85_d - W$	Andwf TMR0, W $W = TMR0 \& W$ addwf NUM1, F $NUM1 = NUM1 + W$ Subwf PORTA, F $PORTA = PORTA - W$
Notice that in subtraction, the W has the minus sign		

Many other instructions of the PIC16 series instruction set are of Type II; refer back to the Appendix at the end of this experiment for study.

Starting Up with basic programs



Program One: Fibonacci Series Generator

In mathematics, the Fibonacci numbers are the following sequence of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

The first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two

```
1  include "p16f84a.inc"
2  Fib0  equ 20      ;At the end of the program the Fibonacci series numbers from 0 to 5 will
3  Fib1  equ 21      ;be stored in Fib0:Fib5
4  Fib2  equ 22
5  Fib3  equ 23
6  Fib4  equ 24
7  Fib5  equ 25
8
9  Clrw           ;This instruction clears the working register, W = 0
10 clrf  Fib0     ;The clrf instruction clears a file register specified, here Fib0 = 0
11 movf  Fib0, w  ;initializing Fib1 to the value 1 by adding 1 to Fib0 and storing it in Fib1
12 addlw 1
13 movwf Fib1
14
15 movf  Fib0, W  ; Fib2 = Fib1 + Fib0
16 addwf Fib1, W
17 movwf Fib2
18
19 movf  Fib1, W  ; Fib3 = Fib2 + Fib1
20 addwf Fib2, W
21 movwf Fib3
22
23 movf  Fib2, W  ; Fib4 = Fib3 + Fib2
24 addwf Fib3, W
25 movwf Fib4
26
27 movf  Fib3, W  ; Fib5 = Fib4 + Fib3
28 addwf Fib4, W
29 movwf Fib5
30 nop
31 end
```

1. Start a new MPLAB session, add the file *example1.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. Simulate the program step by step, analyze and study the function of each instruction. **Stop at the “nop” instruction**
6. Study the comments and compare them to the results at each stage and after executing the instructions
7. As you simulate your code, keep an eye on the MPLAB status bar below (the results shown in this status bar are not related to the program, they are for demo purposes only)



The status bar below allows you to instantly check the value of the flags after each instruction is executed. In the figure above, the flags are z, DC, C.

- ❖ A **capital letter** means that the value of the flag is **one**; meanwhile a **small letter** means a value of **zero**. In this case, the result is not zero; however, digit carry and a carry are present.

Another faster method of simulation: Run and break points

Many times you will need to make some changes to your code, additions, omissions and bug fixes. It is not then flexible to step into your code step by step to observe the changes you have made especially when your program is large. It would be a good idea to execute your code **all at once** or **up to a certain point** and then read the results from the watch window.

Now suppose we want to execute the Fibonacci series code at once - to do so, follows these steps:

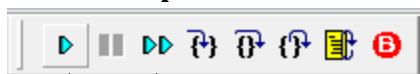
1. Double click on the “nop” instruction (line 30), a red circle with a letter “B” inside is shown to the left of the instruction. This is called a breakpoint. Breakpoints instruct the simulator to stop code execution at this point. *All instructions before the breakpoint are only executed*

```

29  movwf  Fib5
30  nop
31  end

```

2a. Now press the run button

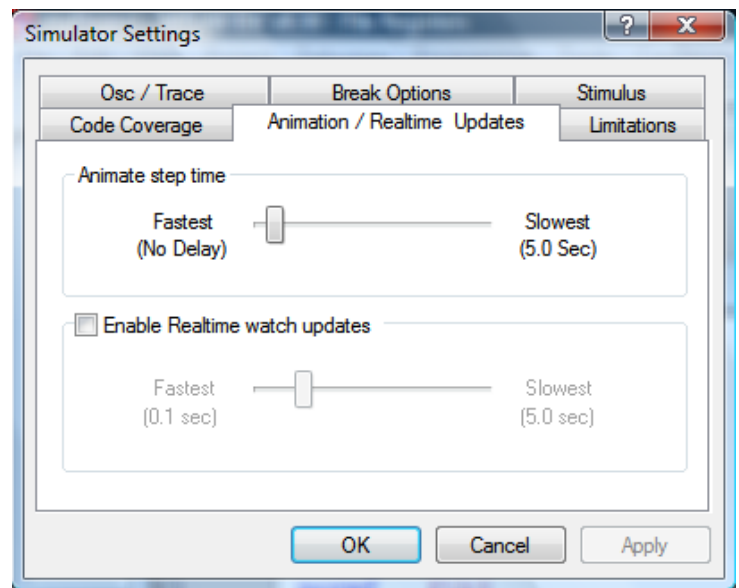


Run ↑ Animate ↑

- 2b. Alternatively, you can instruct the IDE to automatically step into the code an instruction at a time by simply pressing “animate”

You can control the speed of simulation as follows:

1. **Debugger** ↘ **Settings** ↘ **Animation/ Real time Updates**
2. Drag the slider to set the speed of simulation you find convenient



Program Memory Space Usage

Though we have written about 31 lines in the editor, the total number of program memory space occupied is far less, remember that directives are not instructions and that they are not downloaded to the target microcontroller. To get an approximate idea of how much space does the program occupy: Select **View** ↪ **Program Memory** ↪ **Symbolic** tab

Line	Address	Opcode	Label
1	000	0103	CLRW
2	001	01A0	CLRF Fib0
3	002	0820	MOVF Fib0, W
4	003	3E01	ADDLW 0x1
5	004	00A1	MOVWF Fib1
6	005	0820	MOVF Fib0, W
7	006	0721	ADDWF Fib1, W
8	007	00A2	MOVWF Fib2
9	008	0821	MOVF Fib1, W
10	009	0722	ADDWF Fib2, W
11	00A	00A3	MOVWF Fib3
12	00B	0822	MOVF Fib2, W
13	00C	0723	ADDWF Fib3, W
14	00D	00A4	MOVWF Fib4
15	00E	0823	MOVF Fib3, W
16	00F	0724	ADDWF Fib4, W
17	010	00A5	MOVWF Fib5
18	011	0000	NOP
19	012	3FFF	

Note that the last instruction written is “nop” (end is a directive). The total space occupied is only 18 memory locations

The “opcode” field shows the actual machine code of each instruction which is downloaded to the PIC

Program Two: Implementing the function $Result = (X + Y) \oplus Z$

This example is quite an easy one, initially the variable X, Y, Z are loaded with the values which make the truth table

1	include "p16F84A.inc"		
2			
3	cblock 0x25		
4	VarX		
5	VarY		
6	VarZ		
7	Result		
8	endc		
9			
10	org 0x00		
11	Main		;loading the truth table
12	movlw B'01010101'		;ZYX Result
13	movwf VarX	;000 0	(bit7_VarZ, bit7_VarY, bit7_VarX)
14	movlw B'00110011'	;001 1	(bit6_VarZ, bit6_VarY, bit6_VarX)
15	movwf VarY	;010 1	.
16	movlw B'00001111'	;011 1	.
17	movwf VarZ	;100 1	.

18		;101	0	.
19		;110	0	.
20		;111	0	(bit0_VarZ, bit0_VarY, bit0_VarX)
21	movf	VarX, w		
22	iorwf	VarY, w		
23	xorwf	VarZ, w		
24	movwf	Result		
25	nop			
26	end			

1. Start a new MPLAB session, add the file *example2.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. Simulate the program step by step, analyze and study the function of each instruction. **Stop at the “nop” instruction**
6. Study the comments and compare them to the results at each stage and after executing the instructions

Appendix A: Instruction Listing

Mnemonic, Operands	Description	Cycles	14-Bit Opcode				Status Affected	Notes	
			MSb			LSb			
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	1fff	ffff	Z	2
CLRWF	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECWF	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	1fff	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
RRWF	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		
CLRWDTC	-	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	

Appendix B: MPLAB Tools

Another method to view the content of data memory is through the File Registers menu:

❖ Select **View Menu** → **File Registers**

After building the Example1.asm codes, start looking at address 20 (which in our code corresponds to Fib0), to the right you will see the adjacent file registers from 21 to 2F.

Observe that **after code execution**, these memory locations are filed with Fibonacci series value as anticipated.

Address	00	01	02	03	04	05	06	07	08	09	0A	0B
00	--	00	11	18	00	00	00	--	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00
20	00	01	01	02	03	05	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00
50	--	--	--	--	--	--	--	--	--	--	--	--

