Hashemite University

Faculty of Engineering and Technology

Computer Engineering Department

Embedded system Internal Lab Manual

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 1
Introduction to PIC Microcontroller & MPLAB IDE

1.1 Objectives:

1. Introduce the PIC16F84A microcontroller.
2. Microchip MPLAB Integrated Development Environment (IDE) and the whole process of building a project, writing simple codes, and compiling the project.
3. Code simulation.

1.2 Pre-lab Preparation:

- Read the experiment thoroughly BEFORE coming to the lab.

1.3 Equipment:

- Personal computer.

1.4 Theoretical background::

First: PIC16F84A Features:

- Only 35 single word instructions
- Operating speed: DC - 20 MHz clock input
- 1024 words of program memory
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
- 10,000 erase/write cycles Enhanced FLASH Program memory typical
- Low power, high speed technology

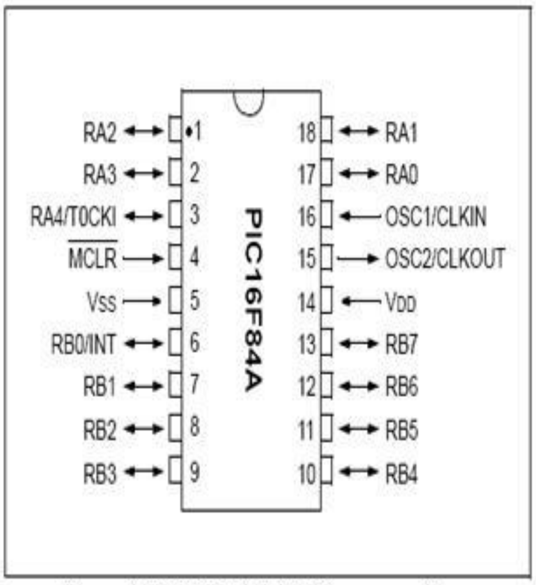
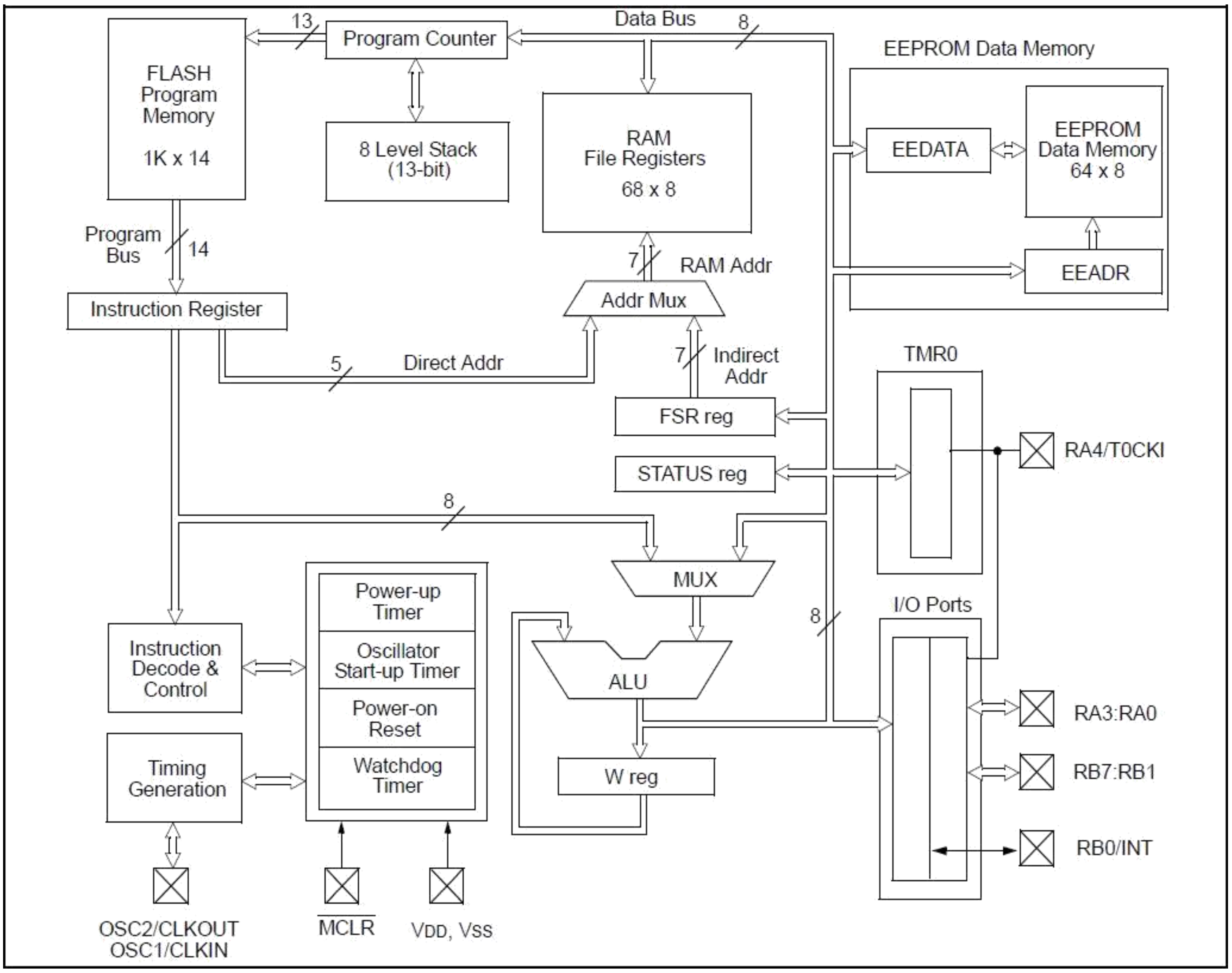


Figure 1: PIC16F84A 8-bit Microcontroller



PIC 16F84A Block diagram

Second: Before starting up with MPLAB

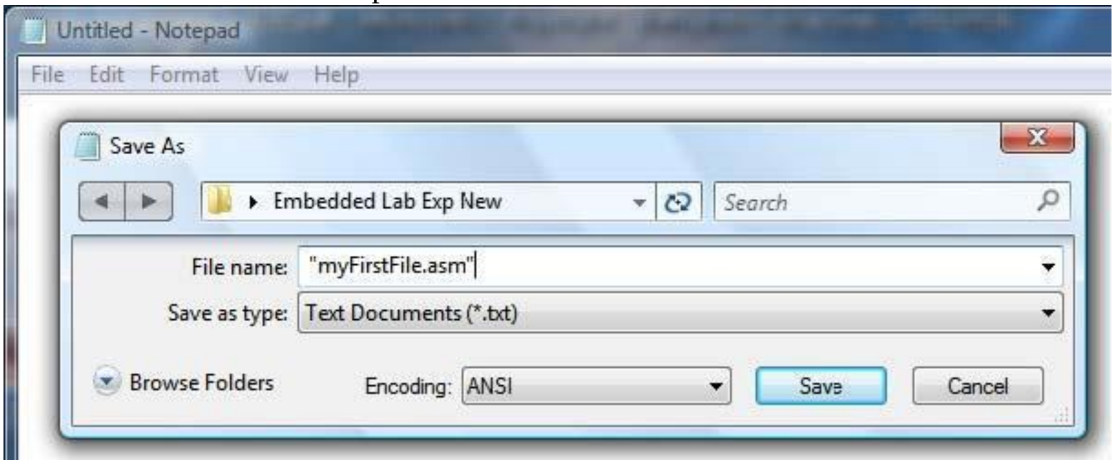
Before using MPLAB you have to make sure of the following:

1. All your files should be stored in a short path:

The total number of characters in a path should not exceed 64	***Char No.***	
C:\ or D:\ or …	3	✓
D:\Interfacing \	11	✓
D:\ Interfacing \Lab	14	✓
D:\Engineer\Year_Three\Summer_Semester\Interfacing_Lab\Experiment_1\MyProgram.asm	76	×
Any file on Desktop		×

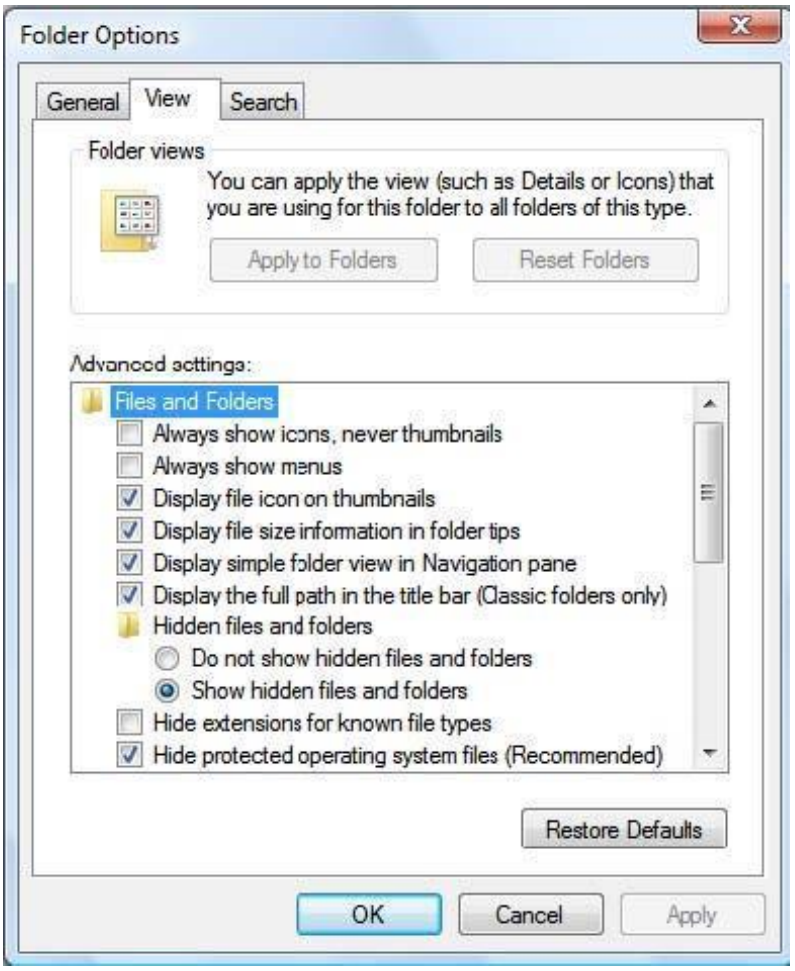
2. All programs written, simulated and debugged in MPLAB should be stored in files with .asm extension. Initially open a notepad file and proceed to change the file extension in **either one** of the following two ways:

   a. Before saving the file in notepad and in the save dialog box, name the file as “Lab1exp.asm” **WITH THE DOUBLE QUATATIONS MARKS**, this will instruct Notepad to save the file in .asm format.



File name: "myFirstFile.asm"
Save as type: Text Documents (*.txt)
Encoding: ANSI

   b. In windows **XP and earlier**, in your current folder, go to Tools → Folder Options → View and then uncheck the option *“hide extension for known file types”*. Then rename your file from xxxxx.txt to myFirstFile.asm



You can apply the view (such as Details or Icons) that you are using for this folder to all folders of this type.

Advanced settings:

Files and Folders
- [ ] Always show icons, never thumbnails
- [ ] Always show menus
- [✓] Display file icon on thumbnails
- [✓] Display file size information in folder tips
- [✓] Display simple folder view in Navigation pane
- [✓] Display the full path in the title bar (Classic folders only)

Hidden files and folders
- ○ Do not show hidden files and folders
- ● Show hidden files and folders
- [ ] Hide extensions for known file types
- [✓] Hide protected operating system files (Recommended)

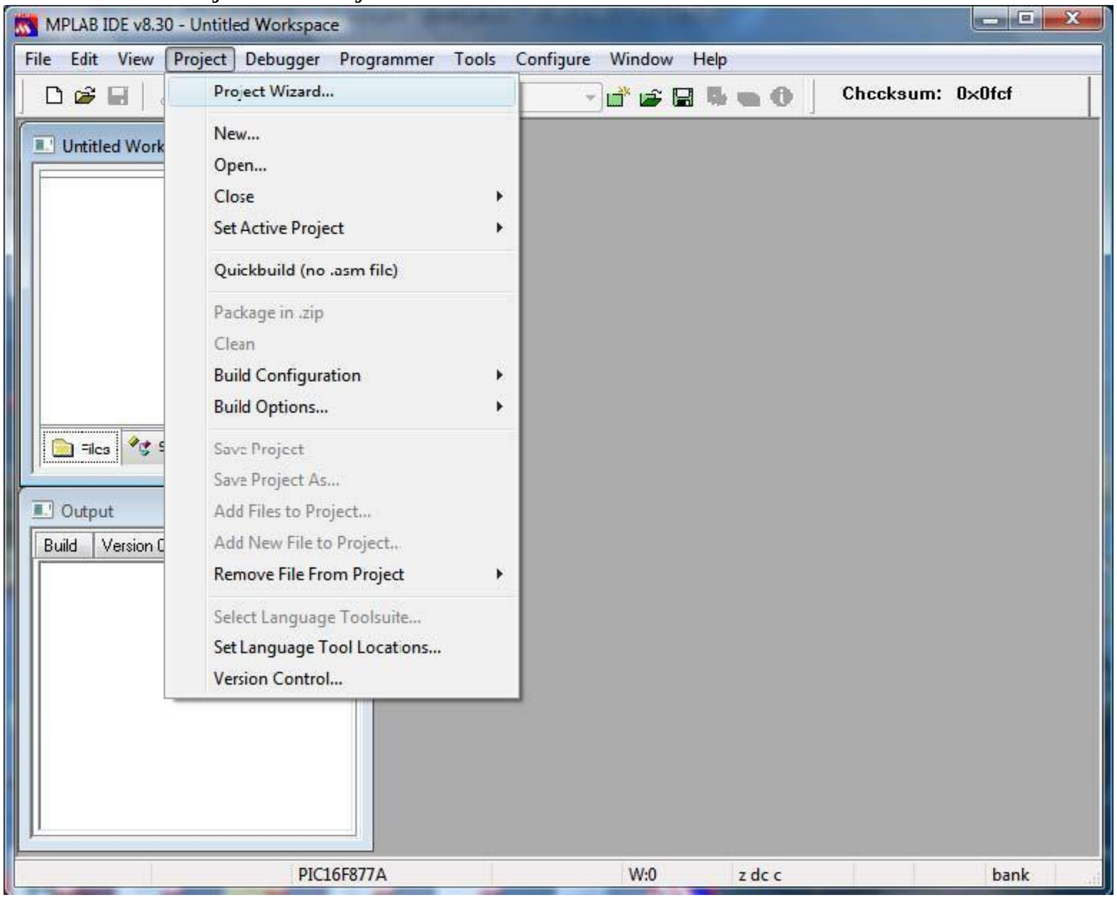
In windows **Vista**, in your current folder, go to Organize → Folder and Search Options →View and then uncheck the option *"hide extension for known file types"*. Then rename your file from xxxxxx.txt to myFirstFile.asm

Third: Starting MPLAB

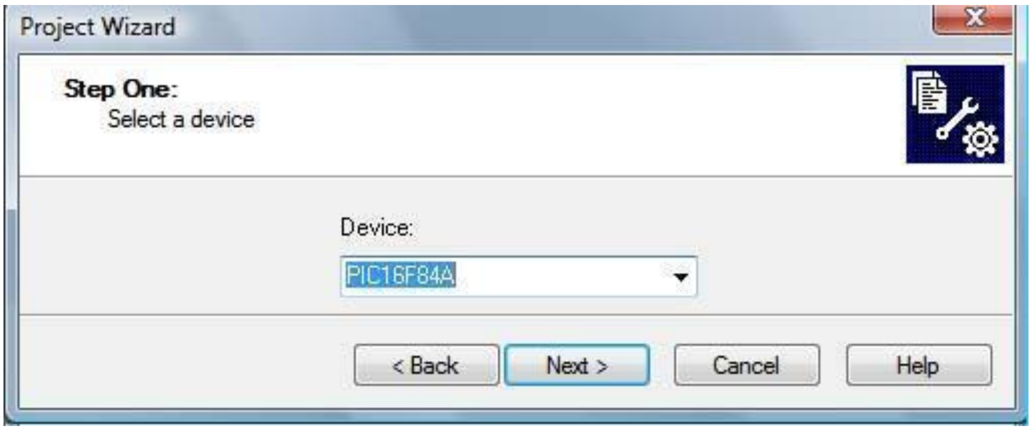
THIS PART ASSUMES YOU HAVE ALREADY SAVED A FILE IN AN ASM EXTENSION

Create a project in MPLAB by following these simple steps:

1. Select the Project → Project Wizard menu item → Next.

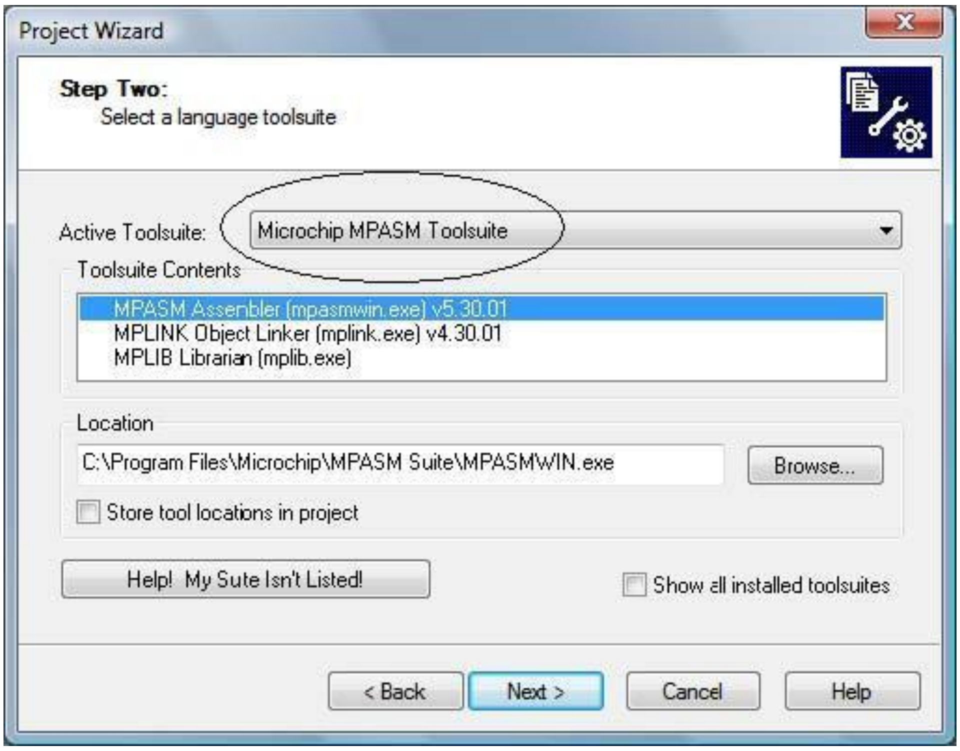


2. In the device selection menu, choose 16F84A (or your target PIC) →Next



3. In the Active Toolsuite, choose Microchip MPASM Toolsuite → Click next.

DO NOT CHANGE ANYTHING IN THIS SCREEN



Project Wizard

Step Two:
Select a language toolsuite

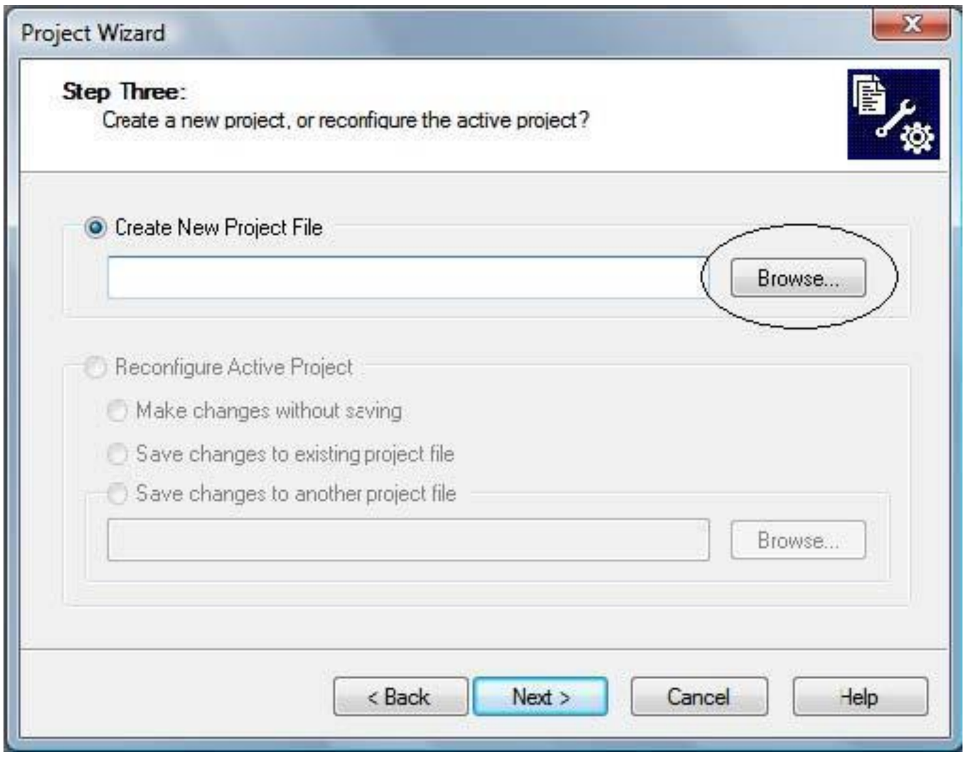
Active Toolsuite: Microchip MPASM Toolsuite

Toolsuite Contents
MPASM Assembler (mpasmwin.exe) v5.30.01
MPLINK Object Linker (mplink.exe) v4.30.01
MPLIB Librarian (mplib.exe)

Location
C:\Program Files\Microchip\MPASM Suite\MPASMWIN.exe

Store tool locations in project

4. Browse to the directory where you saved your ASM file. Give your project a name → Save →Next.



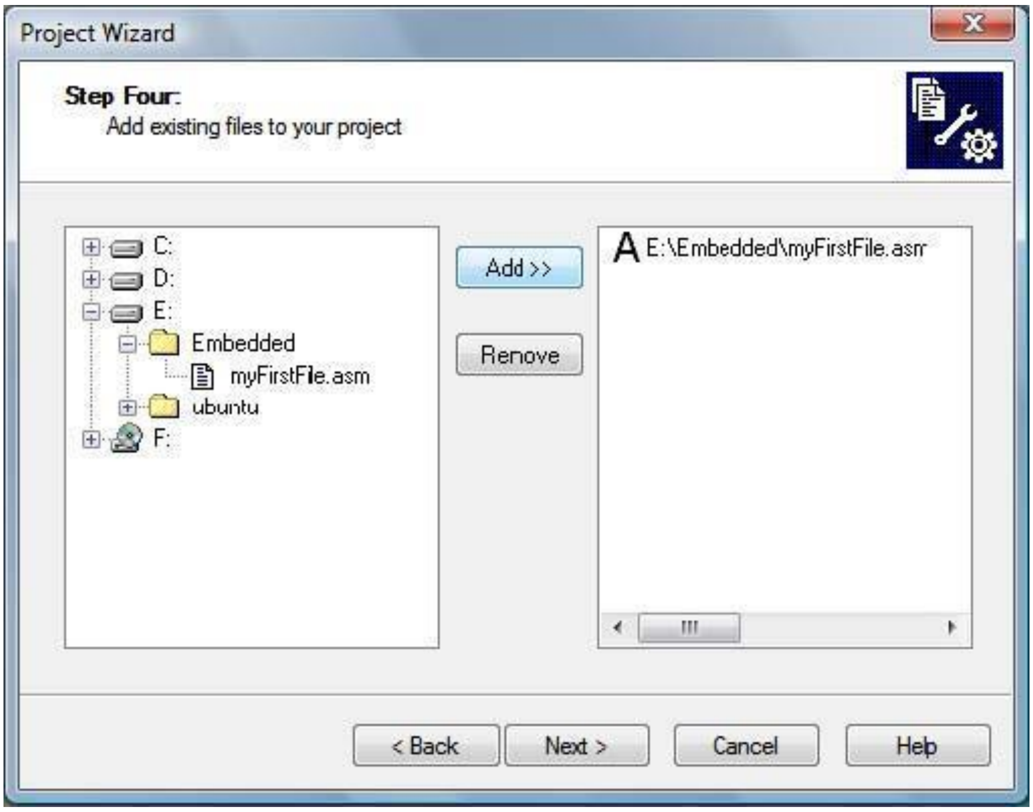
Project Wizard

Step Three:
Create a new project, or reconfigure the active project?

Create New Project File

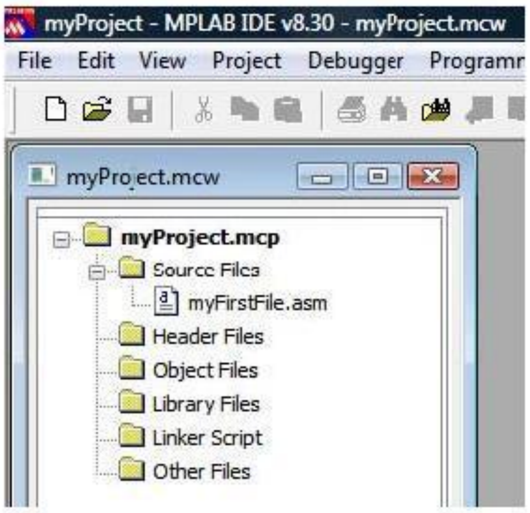
Browse...
Reconfigure Active Project
Make changes without saving
Save changes to existing project file
Save changes to another project file
Browse...
< Back
Next >
Cancel
Help

5. If, in Step 4, you navigated correctly to your file destination you should see it in the left pane otherwise choose back and browse to the correct path. When done Click add your file to the project (here: myFirstFile.asm). Make sure that the letter A is beside your file and not any other letter → Click next → Click Finish.

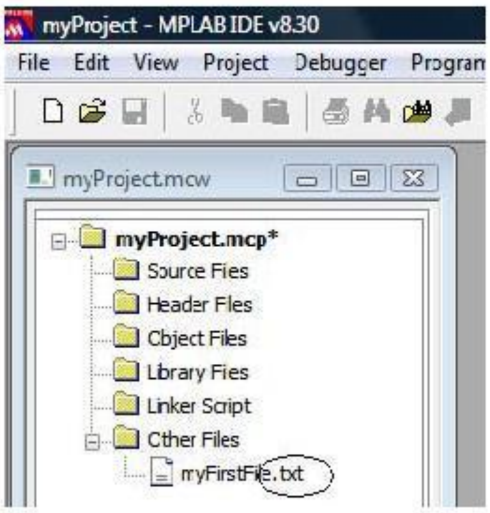


6. You should see your ASM file under *Source file, now you are ready to begin.*

Double click on the myFirstFile.asm file in the project file tree to open. This is where you will write your programs, debug and simulate them.



CORRECT



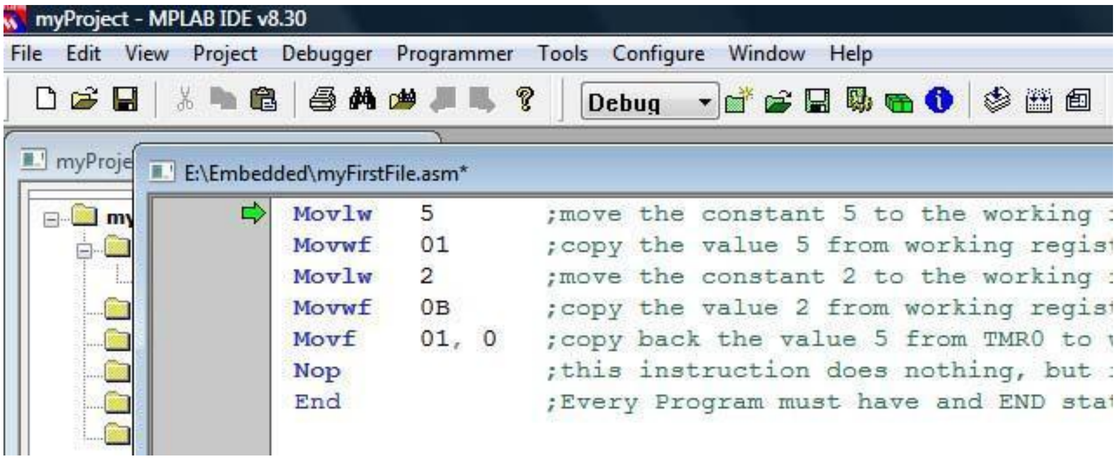
WRONG

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
```

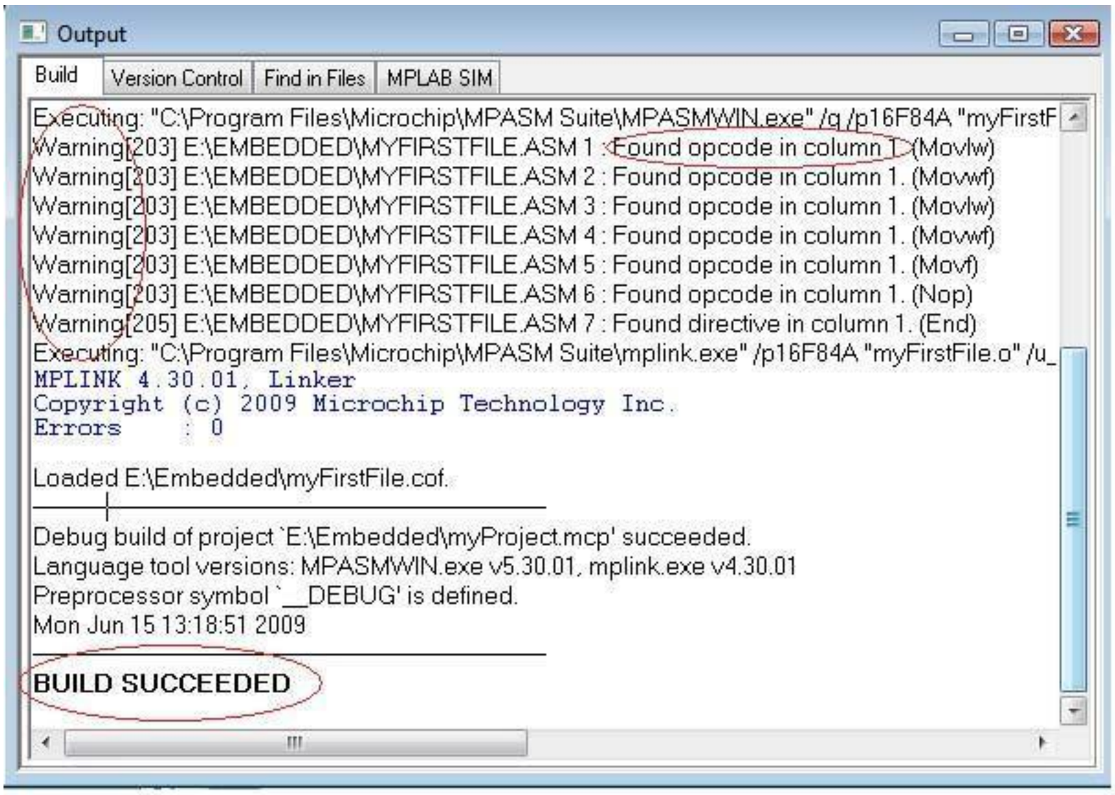
After writing the above instructions we should build the project, do so by pressing ***build***



myProject - MPLAB IDE v8.30
File Edit View Project Debugger Programmer Tools Configure Window Help
Debug
myProje
my
E:\Embedded\myFirstFile.asm*

```
Movlw    5        ;move the constant 5 to the working
Movwf    01       ;copy the value 5 from working regis
Movlw    2        ;move the constant 2 to the working
Movwf    0B       ;copy the value 2 from working regis
Movf     01, 0    ;copy back the value 5 from TMR0 to
Nop               ;this instruction does nothing, but
End               ;Every Program must have and END stat
```

An output window should show: BUILD SUCCEDDED



```
Executing: "C:\Program Files\Microchip\MPASM Suite\MPASMWIN.exe" /q /p16F84A "myFirstF
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 1 : Found opcode in column 1. (Movlw)
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 2 : Found opcode in column 1. (Movwf)
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 3 : Found opcode in column 1. (Movlw)
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 4 : Found opcode in column 1. (Movwf)
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 5 : Found opcode in column 1. (Movf)
Warning[203] E:\EMBEDDED\MYFIRSTFILE.ASM 6 : Found opcode in column 1. (Nop)
Warning[205] E:\EMBEDDED\MYFIRSTFILE.ASM 7 : Found directive in column 1. (End)
Executing: "C:\Program Files\Microchip\MPASM Suite\mplink.exe" /p16F84A "myFirstFile.o" /u_
MPLINK 4.30.01, Linker
Copyright (c) 2009 Microchip Technology Inc.
Errors    : 0

Loaded E:\Embedded\myFirstFile.cof.
----------------------------------------------------------------------
Debug build of project `E:\Embedded\myProject.mcp' succeeded.
Language tool versions: MPASMWIN.exe v5.30.01, mplink.exe v4.30.01
Preprocessor symbol `__DEBUG' is defined.
Mon Jun 15 13:18:51 2009
----------------------------------------------------------------------
BUILD SUCCEEDED
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 2
MPLAP Introduction to PIC Assembly Instruction Set

1.1 Objectives:

1. The MOV instructions
2. Writing simple codes, compiling the project and Code simulation
3. The concept of bank switching
4. The MPASM directives
5. Microcontroller Flags
6. Arithmetic and logical operations

1.2 Pre-lab Preparation:

- Read the experiment thoroughly BEFORE coming to the lab.
- Read the PIC16F84A data sheet chapters 1, 2 especially (2.1, 2.2, 2.3).
- 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.
- Read Appendix C.(*very important*)

1.3 Equipments:

- Personal computer with MPLAB software installed on it.

1.4 Introduction:

First: 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)

➢ MOV**LW**: 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 ABh to the working register
4. MOVLW H'7F'           ; moves the constant 7Fh 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.
```

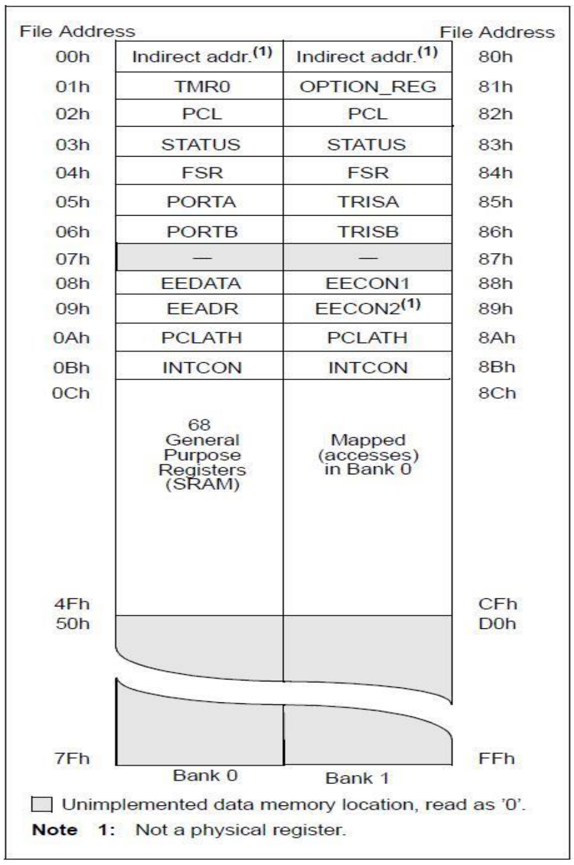
- MOV**WF:** ***COPIES*** the value found in the working register into the data memory, **but to which location?** The location is specified along with the instruction and according to the memory map.

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.



File Address	Bank 0	Bank 1	File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch–4Fh	68 General Purpose Registers (SRAM)	Mapped (accesses) in Bank 0	8Ch–CFh
50h–7Fh	▨	▨	D0h–FFh

▨ Unimplemented data memory location, read as '0'.

Note 1: Not a physical register.

Figure 1: Data memory organization

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 06          ; COPIES the value found in W to PORTB
4. MOVWF 0C          ; COPIES the value found in W to a GPR (location 0C)
5. MOVWF 32          ; COPIES the value found in W to a GPR (location 32)
6. 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 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 fornow
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 SUCCEDDED

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”.

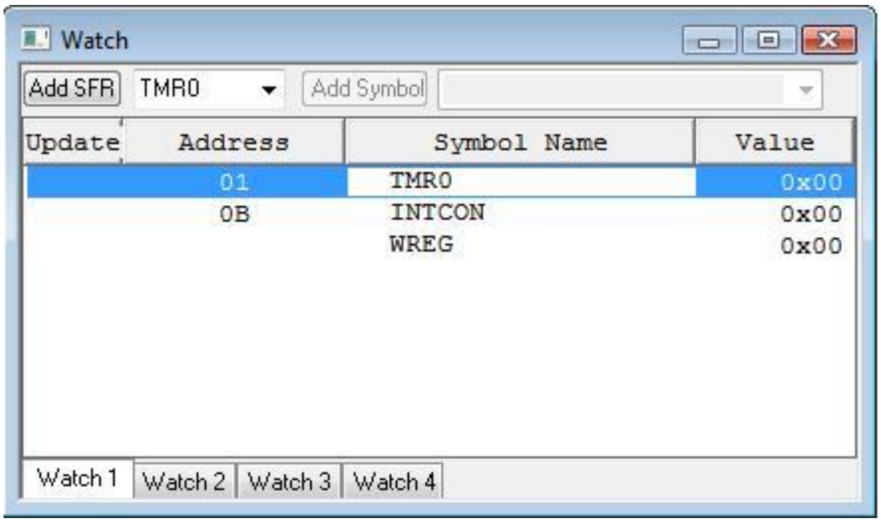
Preparing for simulation

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

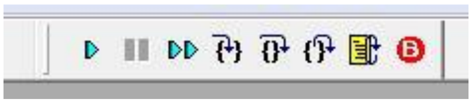
You should have the following:



Update	Address	Symbol Name	Value
	01	TMR0	0x00
	0B	INTCON	0x00
		WREG	0x00

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

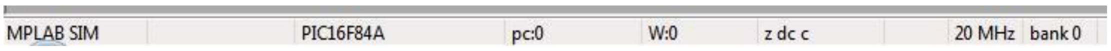
Now new buttons will appear in the toolbar:



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:



MPLAB SIM | PIC16F84A | pc:0 | W:0 | z dc c | 20 MHz | bank 0

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
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,
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
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
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 RP1 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: **B**it **C**lear **F**ile Register (makes a specified bit in a specified file register a 0)*
*BSF: **B**it **S**et **F**ile Register (makes a specified bit in a specified file register a 1)*

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

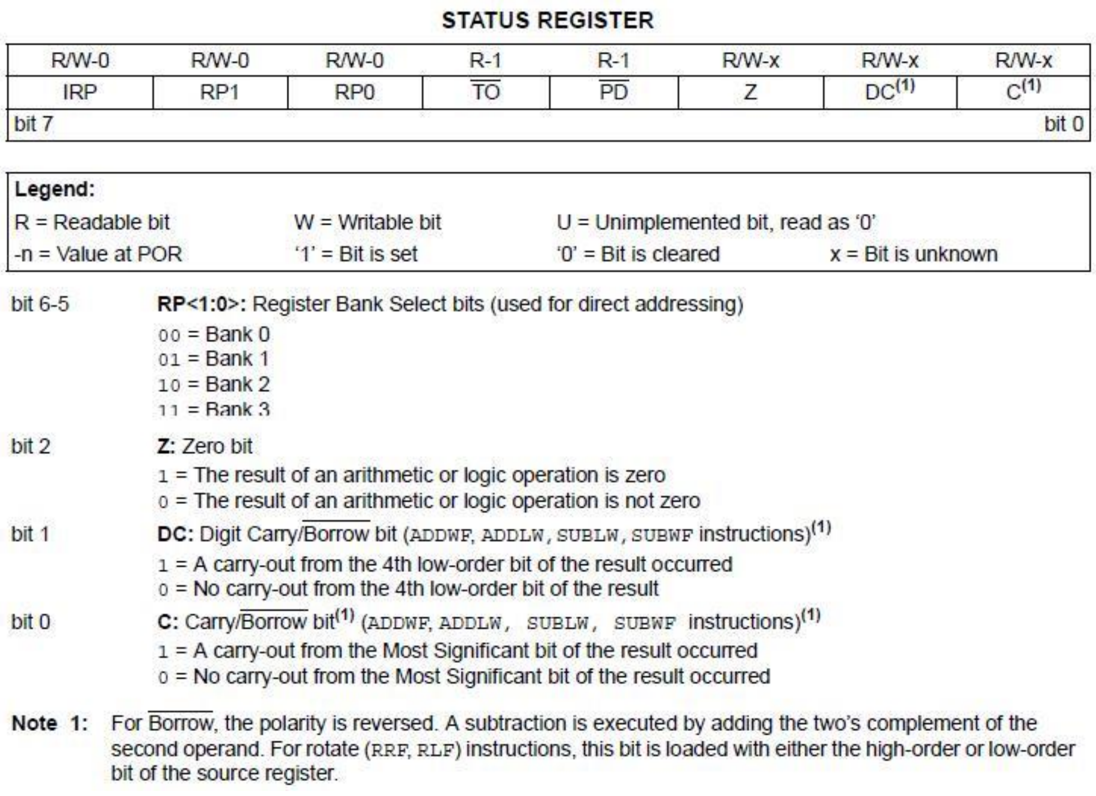
```
#include "P16F84A.inc"
Banksel TRISA
Movlw 5              ; move the constant 5 to the working register
Movwf TRISA          ; copy the value 5 from working register to TRISA (address 85)
Movlw 2              ; move the constant 2 to the working register
Movwf OPTION_REG  ; copy 2 from working register to OPTION_REG (address 81)
Movf TRISA, W        ; copy back the value 5 from TRISA to working register
Banksel PORTA
Nop                  ; this instruction does nothing
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/$\overline{\text{Borrow}}$ 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/$\overline{\text{Borrow}}$ 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 $\overline{\text{Borrow}}$, 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 ***I***nclusive ***OR***). They only differ by their operands and the result destination.

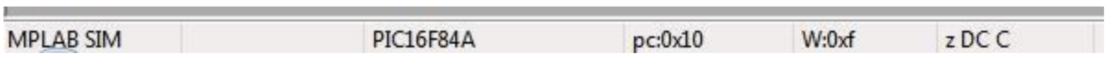
Many other instructions of the PIC16 series instruction set are of Type II; refer back to the Appendix at the end of this experiment for study.

Starting Up with basic program:

Example Program: Implementing the function Result = (X + Y) ⊕ 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. 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)



```
MPLAB SIM          PIC16F84A          pc:0x10          W:0xf          z DC C
```

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.

The code:

```
include "p16F84A.inc"
cblock 0x25
        VarX
        VarY
        VarZ
        Result
endc
        org     0x00
Main                                    ;loading the truth table
        movlw B'01010101'               ;ZYX    Result
        movwf VarX                      ;000    0     (bit7_VarZ, bit7_VarY, bit7_VarX)
        movlw B'00110011'               ;001    1     (bit6_VarZ, bit6_VarY, bit6_VarX)
        movwf VarY                      ;010    1                          .
        movlw B'00001111'               ;011    1                          .
        movwf VarZ                      ;100    1                          .
                                        ;101    0                          .
                                        ;110    0                          .
                                        ;111    0     (bit0_VarZ, bit0_VarY, bit0_VarX)
        movf    VarX, w
        iorwf   VarY, w
        xorwf   VarZ, w
        movwf Result
        nop
        end
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 3
PIC Assembly Instruction Set (2)

1.1 Objectives:

7. Program flow control instructions
8. Conditional and repetition structures
9. The concept of modular programming
10. Macros and Subroutines

1.2 Pre-lab Preparation:

- Read the experiment thoroughly BEFORE coming to the lab.
- Read the PIC16F84A data sheet chapters 1, 2 especially (2.1, 2.2, 2.3).
- Study the two code examples 3.1& 3.2 carefully.

1.3 Equipments:

- Personal computer with MPLAB software installed on it.

1.4 Theoretical backgrounds:

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
```

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** *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*

Coding for efficiency: The repetition structures

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

Program#1:

```
#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
```

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.***

<u>Subroutines;</u>

Subroutines are the closest equivalent to functions

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

  Example:

doMath Instruction 1 Instruction 2 . . Instruction n **return**	**Process** Instruction 1 Instruction 2 . . **Calculate** Instruction 7 Instruction 8 **return** *Note: 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

<u>Macros;</u>

Macros are declared in the following way:

```
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

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

Notes: **on passing parameters to macros and 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:

Approach#1	**Approach#2**
Place the input at the working register Take the output from the working register	Store the input(s) in external variables Load the output(s) from external variables
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
        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.

The General structure of a complete program :(Very Important)

```
;Program Title, description of its function, its inputs and outputs and the programmer name
;****************************************************************
;* FUNCTION:
;* PROGRAMED BY: ENG. EZYA
KHADER ;* INPUTS:
;* OUTPUTS:
;* ***************************************************************
#include "P16F84A.inc"
;------------------------------
; Local equates for your own symbolic designators
;------------------------------
NUM1        equ 20h        ; The NUM1 number is in File 20h

; --------------------------------------------------------
; General Purpose RAM Assignments (Cblock definitions) ; -
--------------------------------------------------------
                              cblock 0x30
                                        MY_VAR           ; MY_VAR is in File 30h

                              endc
; --------------------------------------------------------
; Macro Definitions
; --------------------------------------------------------
MY_MACRO     macro
                              Instruction 1
                              :
                              nop
                    endm
; --------------------------------------------------------
; Vector definition
; --------------------------------------------------------
                              org 0x000
                              nop
                              goto Main

INT_Routine                   org 0x004
                              goto INT_Routine
; --------------------------------------------------------
; The main Program
; --------------------------------------------------------
Main
          Instruction x
          Instruction y
          Call MY_FUN
          :
          Instruction z
          :
          Instruction n
          goto Finish

; --------------------------------------------------------
          Sub Routine Definitions
          --------------------------------------------------------
MY_FUN
          Instruction   w
          Instruction xx
     return
          --------------------------------------------------------
          The main Program end
          --------------------------------------------------------
Finish
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 4
Introduction to Proteus & I/O Interfacing

1.1 Objectives:

1. I/O ports of the Microcontroller.
2. Stressing software and hardware co-design techniques by introducing the *Proteus* IDE package.

1.2 Equipments:

- Personal computer with MPLAB and *Proteus* software installed on it.

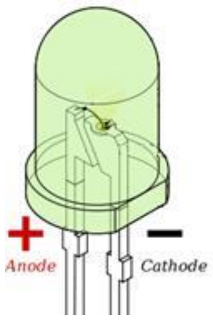
1.3 Theoretical backgrounds:

First: Light Emitting Diodes

Light emitting diodes or LEDs are semiconductor light sources used as indicator lamps in many electronic devices. Modern versions are available across the visible, ultraviolet and infrared wavelengths, with very high brightness and come in a variety of shapes and sizes. The physics behind LED operation is covered in the Electronics I course and will not be offered here.

In order to switch a LED on, forward current must pass from the anode to the cathode, but how to determine which pin of the LED is anode and which is cathode, generally, there are two ways:

1. The longer lead is anode, the shorter is cathode.
2. The cathode has a flat surface as shown in Figure 1.



Resistors with values in between 220Ω to 1kΩ are placed in between the voltage source (often 5 to 9V or even more) and the anode to limit the current entering the LED or else it will burn. The lesser the resistor value, the brighter the LED shines (Ohms Law). In this case these resistors are called current limiting resistors.
Figure 2 shows how to interface a LED to the PORTC pin 1

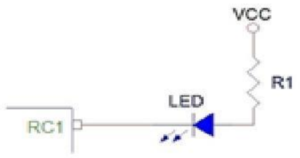
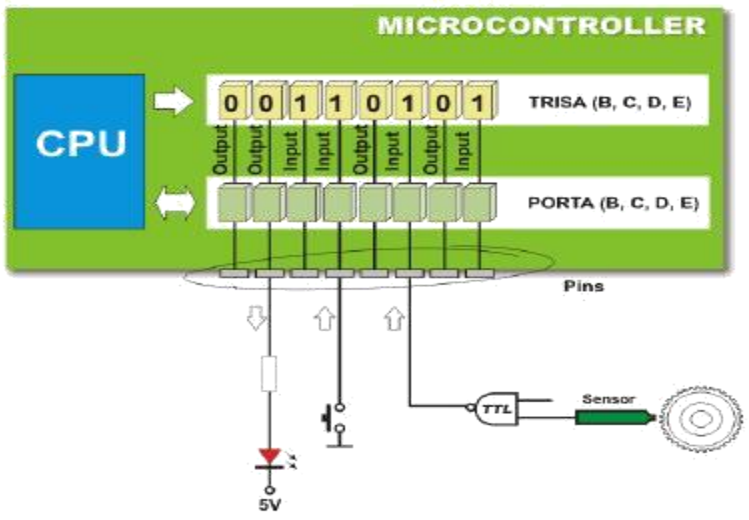


Figure 2

I/O ports of Microcontroller:

PIC microcontrollers' ports are general-purpose bi-directional digital ports. The state of *TRISx* Register controls the direction of the *PORTx* bits. A logic one in a bit position configures the PIC to act as an input and if it has a zero to act as an output. However, a pin can only act as either input or output at any one time but not simultaneously. This means that each pin has a **distinct direction** state. *See the following figure.*



Examples:

Movlw 0x0F Movwf TRISB	Clrf TRISC	Clrf TRISD Comf TRISD, F	Movlw B'00110011' Movwf TRISB
The high nibble of PORTB is output, low nibble is input	Whole PORTC as output	Whole PORTD as input	Bits 2, 3, 6, 7 as output Bits 0, 1, 4, 5 as input

How to decide whether microcontroller's ports must be configured as inputs or outputs?

Input ports "Get Data" from the outside world into the microcontroller while output ports "Send Data" to the outside world.

- LEDs, 7-Segment displays, motors, and LCDs (write mode) that are interfaced to microcontroller‟s ports should be configured as output.
- Switches, push buttons, sensors, keypad and LCDs (read mode) that are interfaced to microcontroller‟s ports should be configured as input.

Second: The PROTEUS Environment:

Proteus PIC Bundle is the complete solution for developing, testing and virtually prototyping your system designs based around the Microchip Technology series of Microcontroller. This software allows you to perform *schematic capture* and to *simulate* the circuits you design.

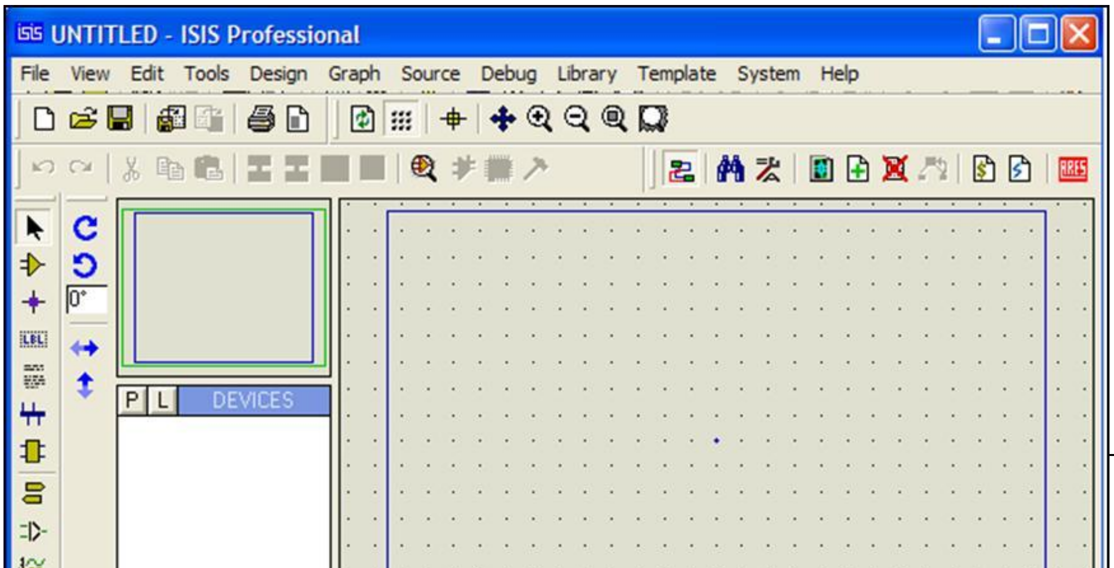

UNTITLED - ISIS Professional
File View Edit Tools Design Graph Source Debug Library Template System Help
0°
P L DEVICES

Figure 1: A screen shot of the Proteus IDE

Proteus How to Start Drawing the Circuit

Start a fresh design, select New Design from File menu then the Create New Design dialogue now appears as shown in Figure 2 and 3. Select Default and press OK

.

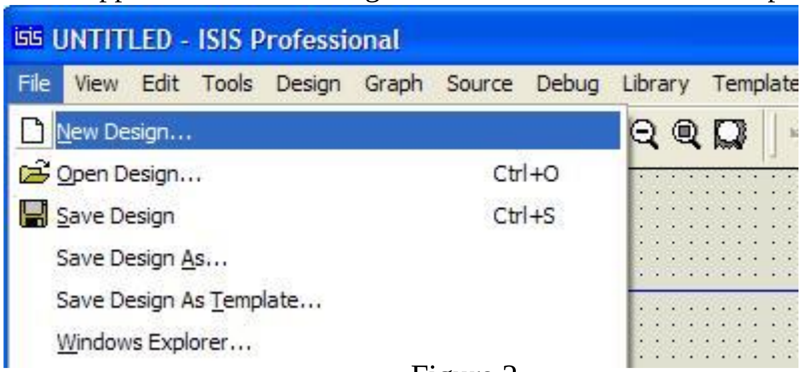
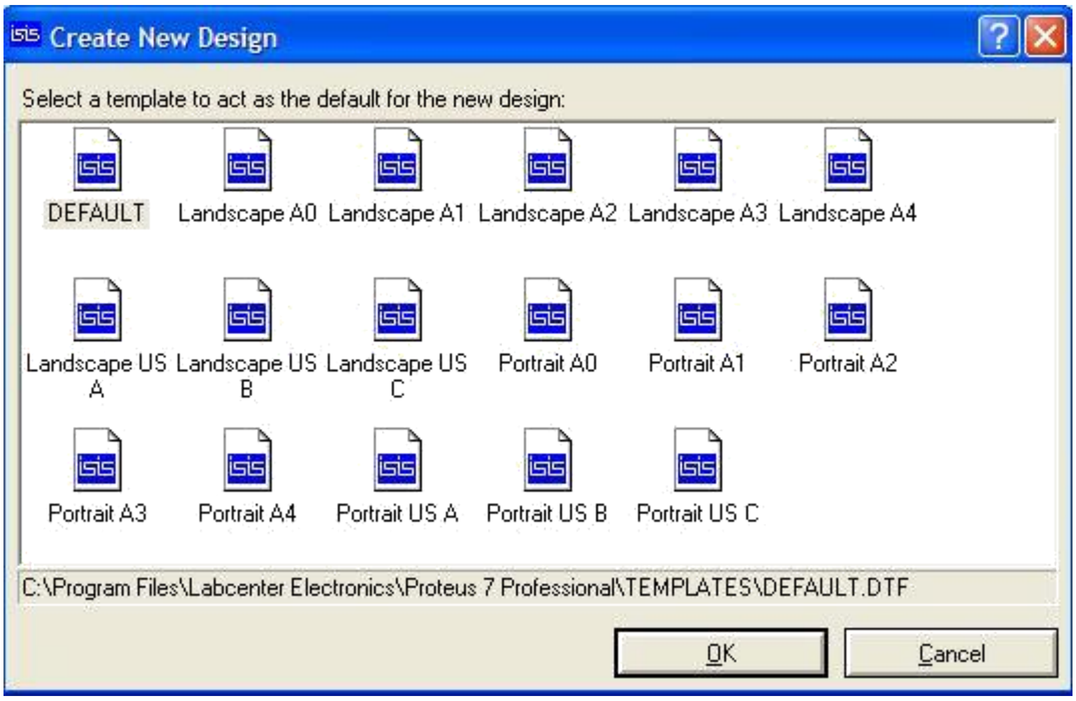


Figure 2


Create New Design

Select a template to act as the default for the new design:

C:\Program Files\Labcenter Electronics\Proteus 7 Professional\TEMPLATES\DEFAULT.DTF

Figure 3

From the Library menu select Pick Device/Symbol see Figure 4 or left click on the letter 'P' above the Object Selector as shown in Figure 5 to launch the Library Browser or Press the 'P' button on the keyboard. The Library Browser will now appear over the Editing Window see Figure 6.

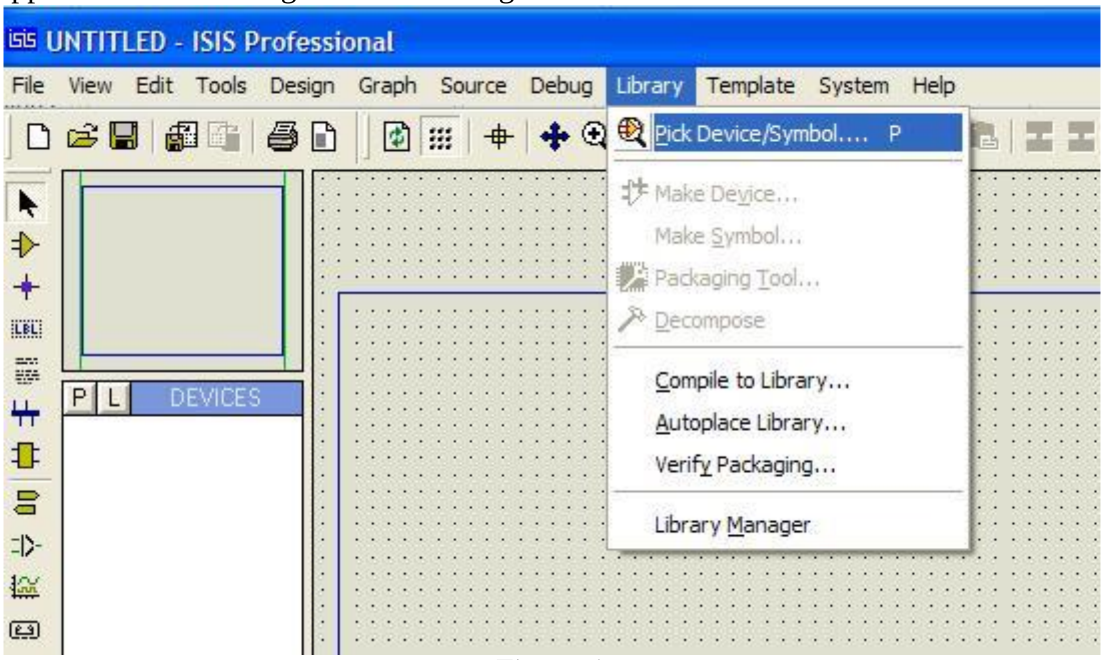


Figure 4

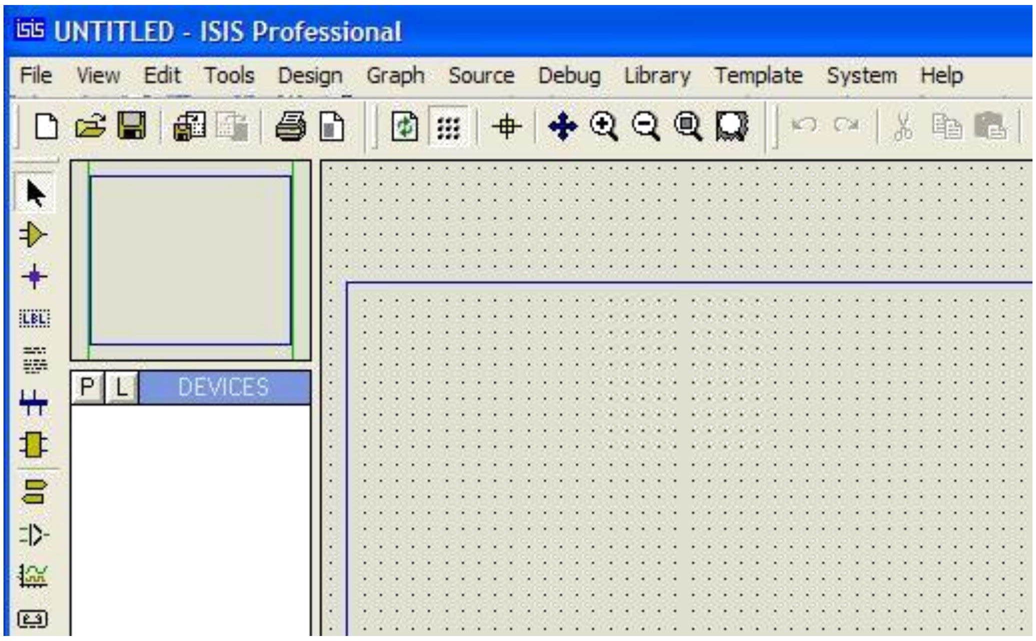
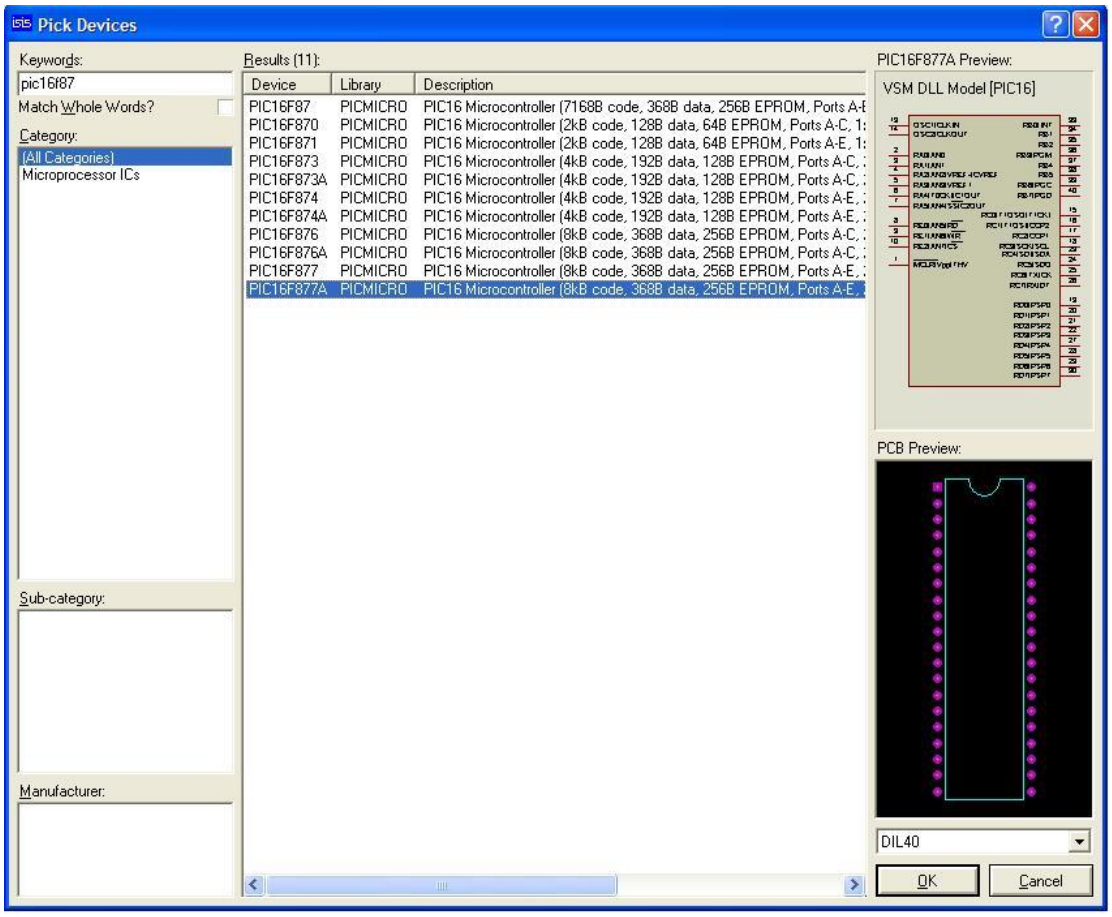


Figure 5



Keywords: pic16f87

Match Whole Words?

Category:
(All Categories)
Microprocessor ICs

Sub-category:

Manufacturer:

Results (11):

Device	Library	Description
PIC16F87	PICMICRO	PIC16 Microcontroller (7168B code, 368B data, 256B EPROM, Ports A-E
PIC16F870	PICMICRO	PIC16 Microcontroller (2kB code, 128B data, 64B EPROM, Ports A-C, 1:
PIC16F871	PICMICRO	PIC16 Microcontroller (2kB code, 128B data, 64B EPROM, Ports A-E, 1:
PIC16F873	PICMICRO	PIC16 Microcontroller (4kB code, 192B data, 128B EPROM, Ports A-C, :
PIC16F873A	PICMICRO	PIC16 Microcontroller (4kB code, 192B data, 128B EPROM, Ports A-C, :
PIC16F874	PICMICRO	PIC16 Microcontroller (4kB code, 192B data, 128B EPROM, Ports A-E, :
PIC16F874A	PICMICRO	PIC16 Microcontroller (4kB code, 192B data, 128B EPROM, Ports A-E, :
PIC16F876	PICMICRO	PIC16 Microcontroller (8kB code, 368B data, 256B EPROM, Ports A-C, :
PIC16F876A	PICMICRO	PIC16 Microcontroller (8kB code, 368B data, 256B EPROM, Ports A-C, :
PIC16F877	PICMICRO	PIC16 Microcontroller (8kB code, 368B data, 256B EPROM, Ports A-E, :
PIC16F877A	PICMICRO	PIC16 Microcontroller (8kB code, 368B data, 256B EPROM, Ports A-E, :

DIL40

Figure 6

Type in ' PIC16F877A ' in the Key words field and double click on the result to place the PIC16F877A into the Object Selector. Do the same for the LEDs, Buttons, Crystal oscillator, capacitors, 7 SEG-COM-Cathode, Resistors.

Once you have selected all components into the design close the Library Browser and left click once on any component in the Object Selector (This should highlight your selection and a preview of the component will appear in the Overview Window at the top right of the screen see Figure 7). Now left click on the Editing Window to place the component on the schematic - repeat the process to all components on the schematic.

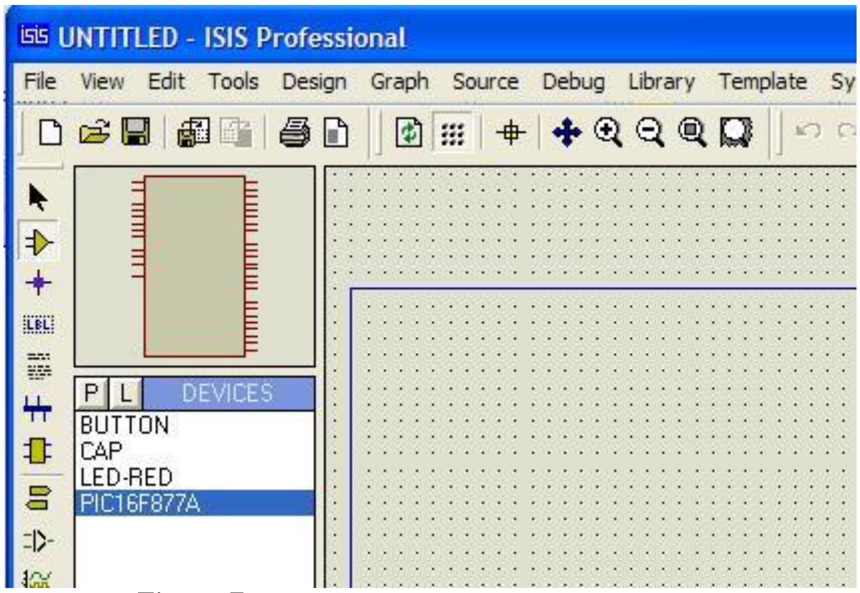

UNTITLED - ISIS Professional
File View Edit Tools Design Graph Source Debug Library Template Sy
P L DEVICES
BUTTON
CAP
LED-RED
PIC16F877A

Figure 7

In order to place ground or 5 voltage right click on the Editing Window ,select place then terminal then select ground(0 V)or power (5V).Connect the components to obtain the circuit you need.

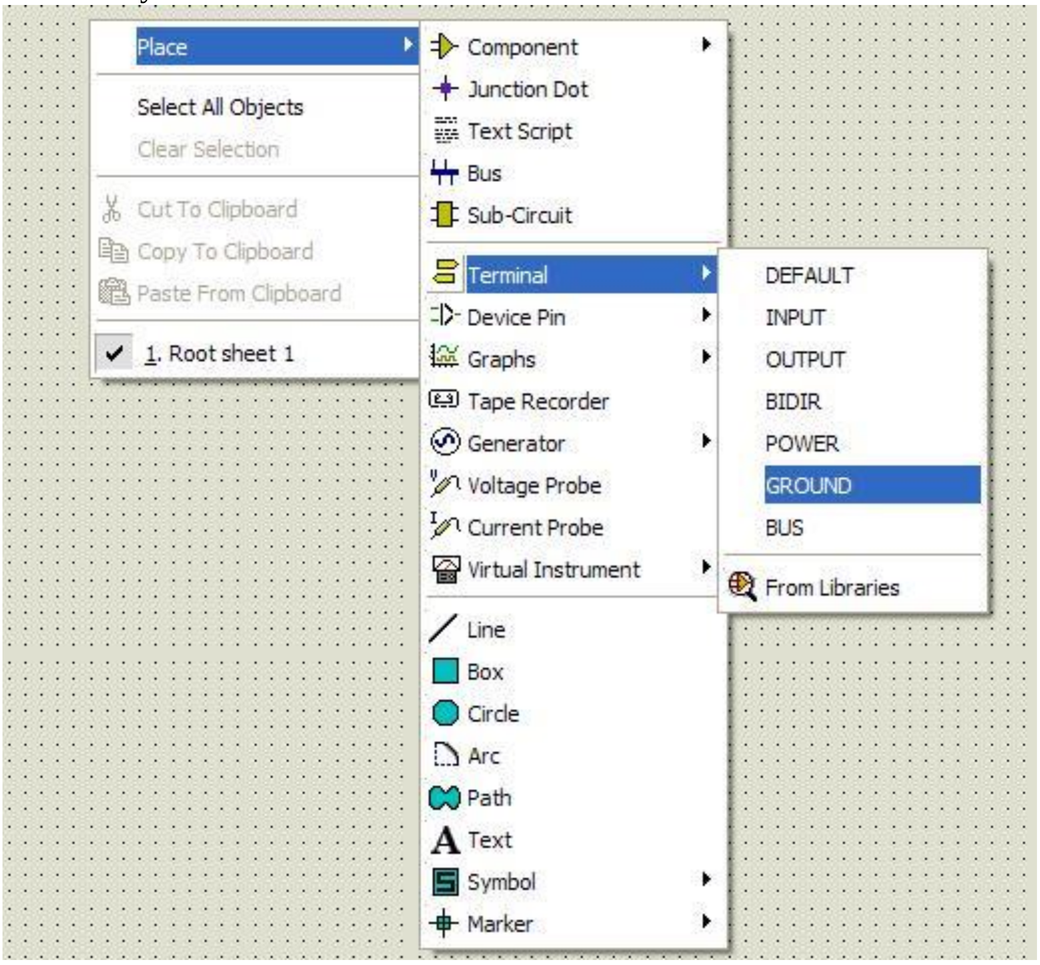


Figure 8

Attaching the Source File

The next stage is to attach the source file to our design in order to successfully simulate the design. We do this through the commands on the Source Menu.

Go to the Source Menu now and select the Add/Remove Source Files Command as shown down.

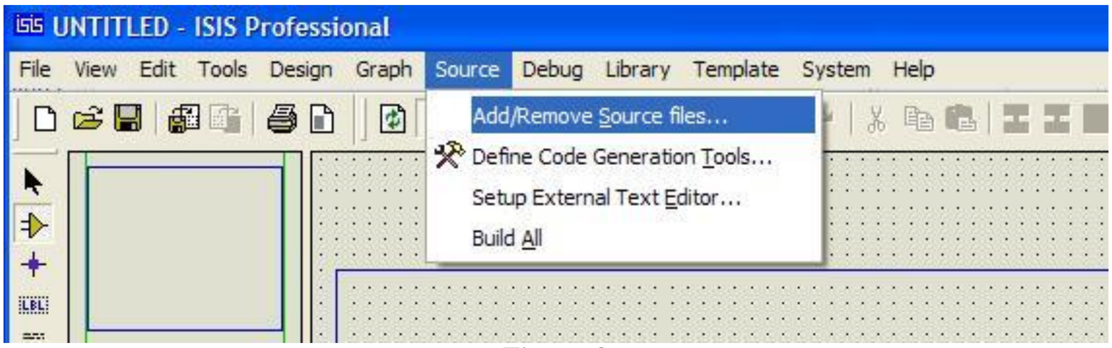

UNTITLED - ISIS Professional
File View Edit Tools Design Graph Source Debug Library Template System Help
Add/Remove Source files...
Define Code Generation Tools...
Setup External Text Editor...
Build All

Figure 9

Click on the New button, browse until you reach the desired directory and select your ASM file.

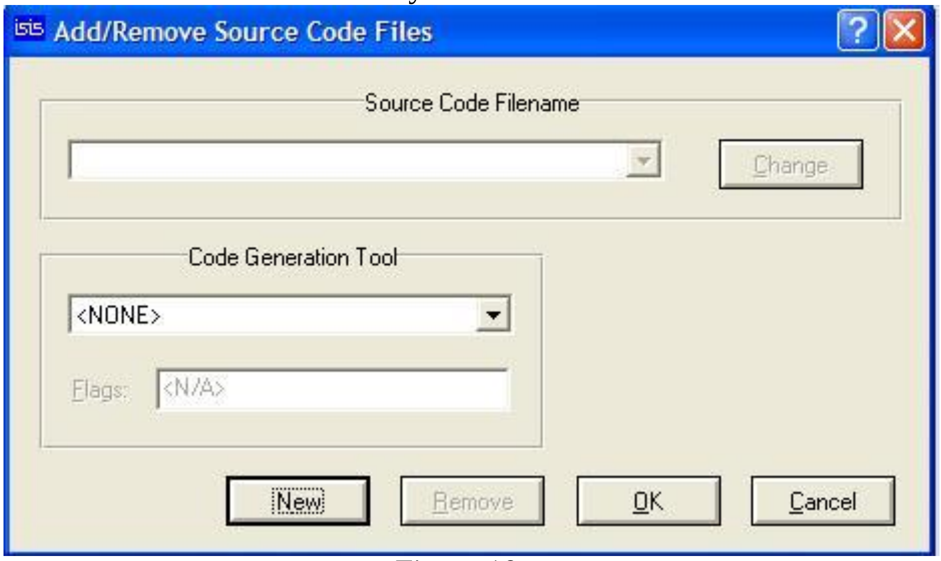
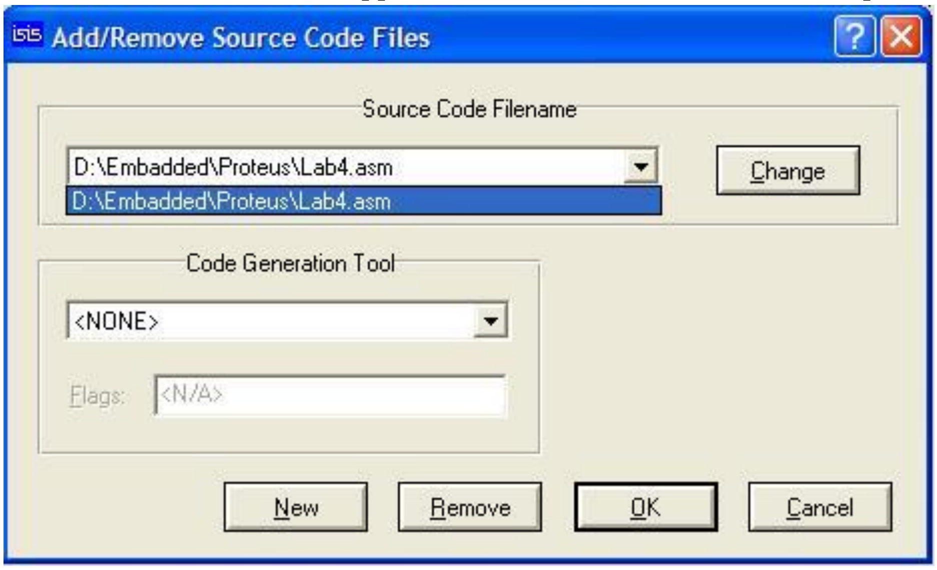


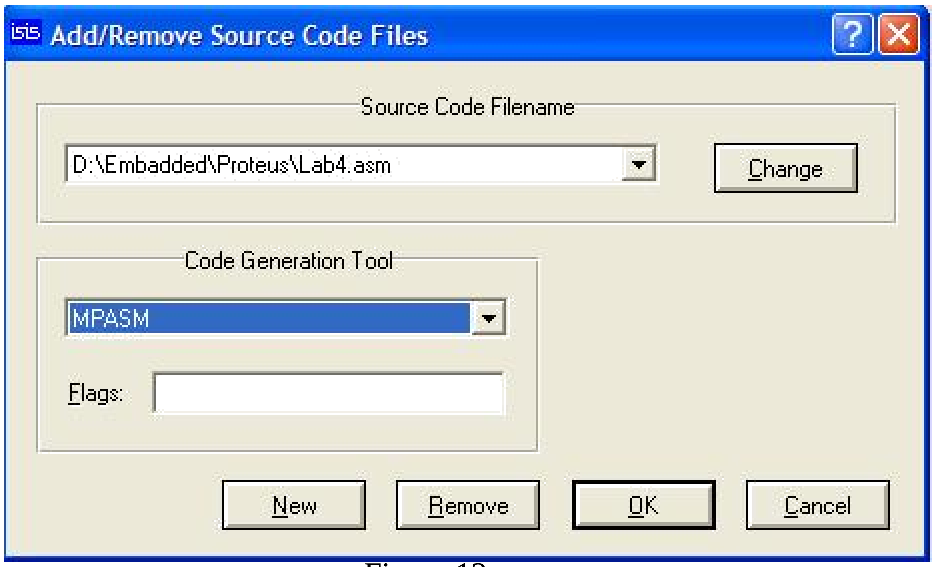
Figure 10

Click open and the file should now appear in the Source Code Filename drop down listbox.



Source Code Filename: D:\Embadded\Proteus\Lab4.asm

Figure 11


Add/Remove Source Code Files

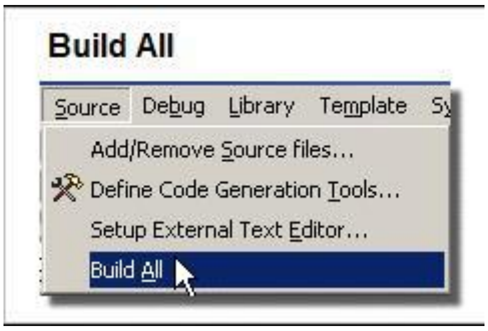
Source Code Filename: D:\Embadded\Proteus\Lab4.asm

Code Generation Tool: MPASM

Flags:

Figure 12

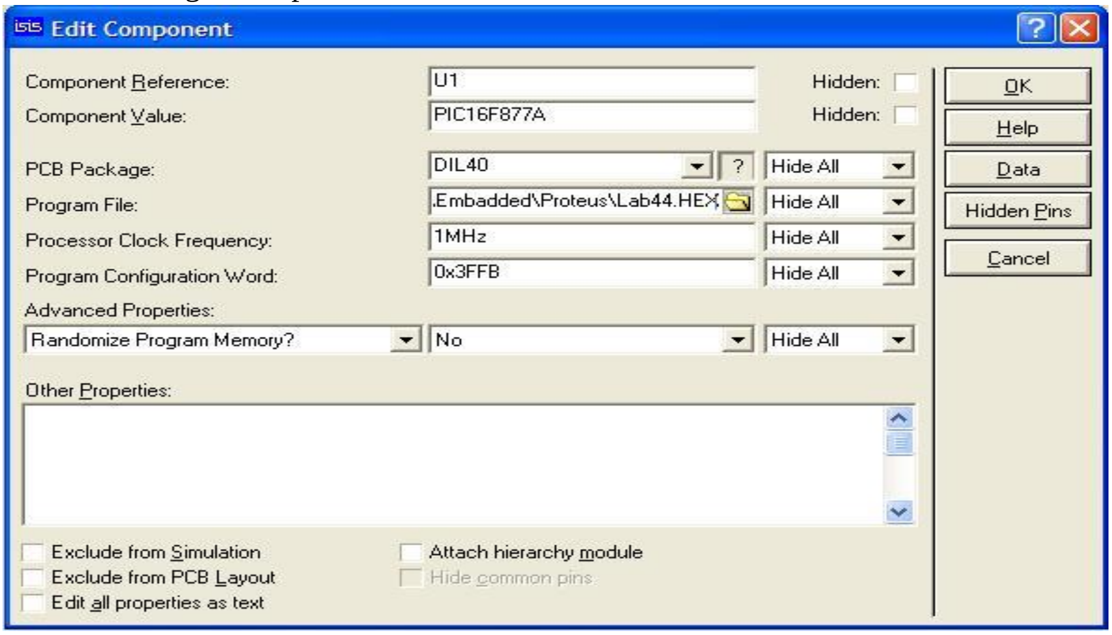
We now need to select the code generation tool for the file. For our purposes the MPASM tool will suffice. This option should be available from the drop down listbox and so left clicking will select it in the usual fashion.



Select the Build All command from the source menu as shown in Figure 13, this provides an excellent way to check that everything has built without errors or warnings.

Figure 13

Finally, it is necessary to specify which file the processor is to run. In our example this will be filename.hex (the hex file produced from MPASM subsequent to assembling filename.asm). To attach this file to the processor, right click on the schematic part for the PIC and then left click on the part. This will bring up the Edit Component dialogue form which contains a field for Program File. If it is not already specified as filename.hex either enter the path to the file manually or browse to the location of the file via the button to the right of the field. Once you have specified the hex file to be run press ok to exit the dialogue form. We have now attached the source file to the design and specified which Code Generation Tool will be used.


Edit Component

Component Reference:	U1	Hidden:
Component Value:	PIC16F877A	Hidden:
PCB Package:	DIL40	Hide All
Program File:	.Embadded\Proteus\Lab44.HEX	Hide All
Processor Clock Frequency:	1MHz	Hide All
Program Configuration Word:	0x3FFB	Hide All
Randomize Program Memory?	No	Hide All

Other Properties:

Figure 14

Debugging the Program Simulating the Circuit

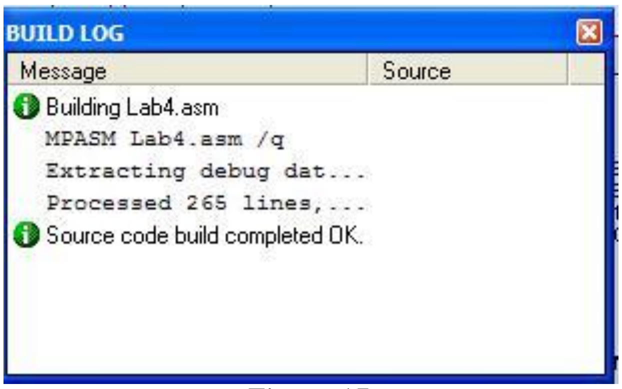
In order to simulate the circuit point the mouse over the Play Button on the animation panel at the bottom right of the screen see Figure 1 and click left. The status bar should appear with the time that the animation has been active for.

Working on your Source Code

To edit a source file:

- Select the source file it will be Filename.asm from the Source menu.
- Edit the source file.
- Select the Build All command from the Source menu.

You should see a small window showing details of the compilation process and the following message "Source code build completed OK".



```
Building Lab4.asm
  MPASM Lab4.asm /q
  Extracting debug dat...
  Processed 265 lines,...
Source code build completed OK.
```

Figure 15

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 5
7-Segment Display

1.1 Objectives:

1. Learn how to interface the 7-segment display with PIC Microcontroller.
2. Learn how to distinguish between the common anode and the common cathode of the 7-segment.

1.2 Pre-lab Preparation:

- Read the experiment thoroughly BEFORE coming to the lab.

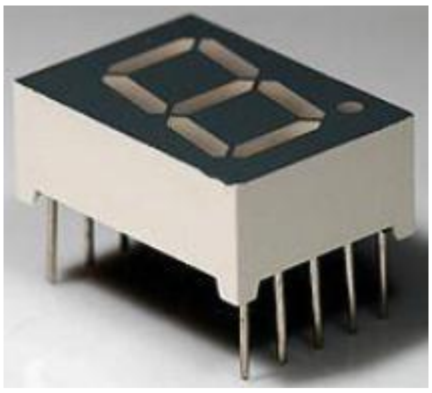
1.3 Equipments:

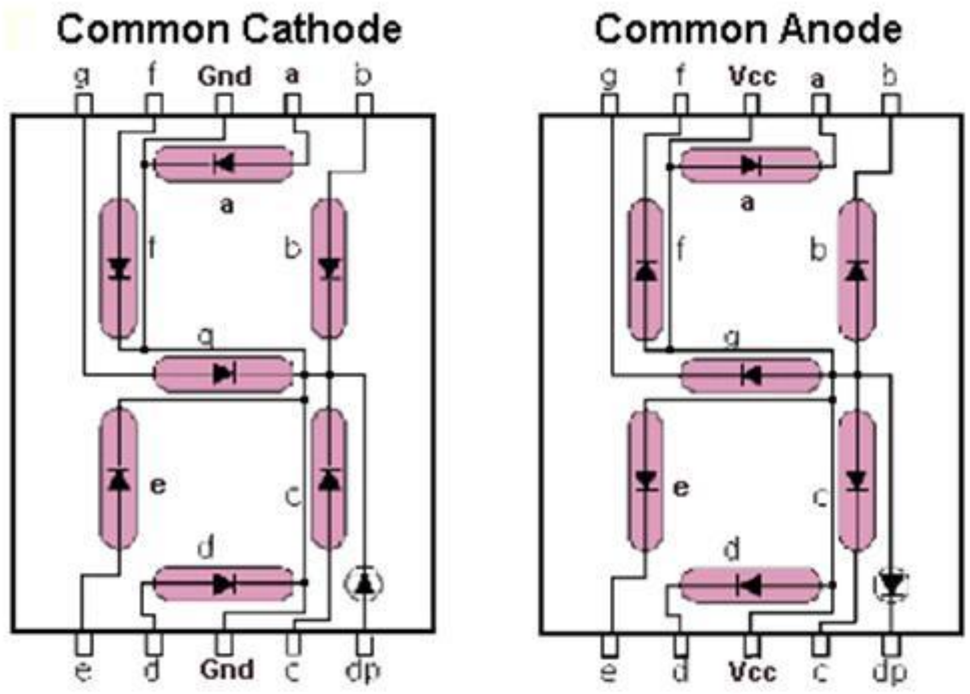
Personal computer with MPLAB and *Proteus* software installed on it.

1.4 Theoretical backgrounds:

Seven-Segment Display

A Seven-Segment display, as its name implies, is composed of seven segments (or technically of seven LEDs) which can be individually switched on or off. This ability to individually control each segment and the layout in which these segments are distributed allows for the representation of the numerals and some characters. If the anode ends of all LEDs are connected together, it is called common anode display. If the cathodes of all LEDs are connected together, it is called a common cathode display. To switch a LED on in a common cathode configuration, you have to send logic high to the segment pin. Conversely, to switch a LED on in a common anode configuration, you have to send logic low to the segment pin.





Interfacing a Seven-Segment display is independent of the type of the module, whether it is common anode or common cathode, only the logic level sent to the display differs. Finally, since the display is basically LEDs, current limiting resistors are used for each segment.