

Lesson 4

Fun with W and F

Overview

Introduction	This lesson introduces the first few PIC instructions.	
In this section	The following is a list of topics in this section:	
	Description	See Page
	Writing Programs	2
	Our First Program	4
	Adding Some Instructions	6
	Helping to Understand Our Program	10
	Incrementing and Decrementing	11
	Bit Manipulation	12
	The Simulator	15
	Wrap Up	16

Writing Programs

Introduction

As we said way back in Lesson 1, we use an assembler to help translate mnemonics for the instructions and memory locations into the ones and zeroes that the processor needs to do its thing.

In this lesson, we will do a number of experiments using some of the more basic instructions in the PIC. These instructions manipulate the working register (W) and the file register (F).

Setting up the first project

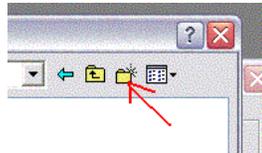
Before we can start to write, we need to have a project for the IDE.

Begin by starting the MPLab.

Select `Project->New...` from the menu and a dialog box with two edit controls will appear. In the upper box, type "Lesson 4a" (without the quotes).

Click on the "Browse..." button on the lower right of the dialog.

Navigate to the "root of all projects" folder that you created in Lesson 3 and click on the "Create Folder" icon (a picture of a folder with a star in the upper right).



A new folder will appear named "New Folder" and the name will be highlighted, ready for editing. Type "Lesson 4" and then double-click on the folder icon.



Check that 'Lesson 4' appears in the top of the dialog then click on the 'Select' button.

Click on OK.

Select 'Configure->Select Device...'. A dialog box will appear. In the dropdown labeled 'Device:', select 'PIC16F84A'. Click OK.

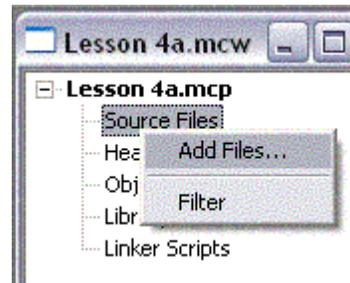
Writing Programs (continued)

Adding files to the project

OK, now we have a project, but it has nothing in it. We need to have at least one assembler source file to type in.

Select 'File->New' from the menu. A new window will appear. Select File->Save and type 'Lesson 4a.asm'. Click Save.

In the project window is a sub-window that lists the different types of files. Right-click 'Source Files' and select 'Add Files...':



A file open dialog will appear. Double-click Lesson 4a.asm. The name will be added to the Lesson 4a.mcp window and the title of the blank window will change from Untitled to the name of your file.

Also notice the asterisk in the title bar of the Lesson 4a.mcp window. This means that the project hasn't been saved. Select 'Project->Save Project' from the main menu.

We now have an empty project, ready for us to go to work.

Our First Program

Introduction

The project consists of a number of files. If you look in the Lesson 4 folder with Windows Explorer, you will see three files at this point. The Lesson 4a.asm file is the one we are currently interested in. The Lesson 4a.mcp file contains the actual project information, that is, what files make up this project. The Lesson 4a.mcw file is the 'Workspace' file. This file remembers what windows are open in our workspace. In the future, if you double-click on the mcw file, the MPLAB will open with all the windows where you last left them.

Basic stuff

There are a few things you need in every program. Might as well get them in the file now.

When entering data into the MPLAB assembler, there are three columns of interest. The columns are separated by whitespace (tabs and spaces). How much whitespace is entirely up to us. We can use a single space, or 10 tabs, really doesn't matter to the assembler. Personally, I like to use two tabs. This makes the columns line up without thinking much about it, and it allows a reasonable length for identifiers.

The first column is anything that starts in column 1. The assembler assumes that this is a *label* that we will reference somewhere in our program.

The second column contains the *opcode*. This is the instruction that tells the PIC what we want it to do.

The third column is the *operand*. This is the thing we want the PIC to do something to.

Besides instructions, there can be assembler directives. These don't end up as instructions in the PIC, instead, they tell the assembler things we want it to know.

We need three directives in any program:

```
processor    16f84a
include     <p16f84a.inc>
end
```

It's also a good idea to include the configuration word. We will talk about this one in more detail, but for now, type in the following:

```
<tab><tab>processor<tab>16f84a<enter>
<tab><tab>include<tab><tab><p16f84a.inc><enter>
<tab><tab>__config<tab>_HS_OSC & _WDT_OFF & _PWRTE_ON<enter>
<tab><tab>end<enter>
```

The `processor` directive tells the assembler which type of PIC we are using. The `include` directive tells the assembler to include a file which contains definitions for a number of symbols relevant to that processor. The `__config` tells the processor that we will be using a crystal (`_HS_OSC`), we want the watchdog timer turned off (`_WDT_OFF`) and we want the power-up timer enabled (`_PWRTE_ON`).

Select 'File->Save' to save your work.

Continued on next page

Our First Program, Continued

Assembling the program

OK, so far, the program doesn't do anything ... there are no instructions. But we can check for typos by assembling the program. From the main menu, select 'Project->Build All'.

We will get a new window with a bunch of junk, but the last line should say:

```
BUILD SUCCEEDED
```

Adding Some Instructions

Introduction

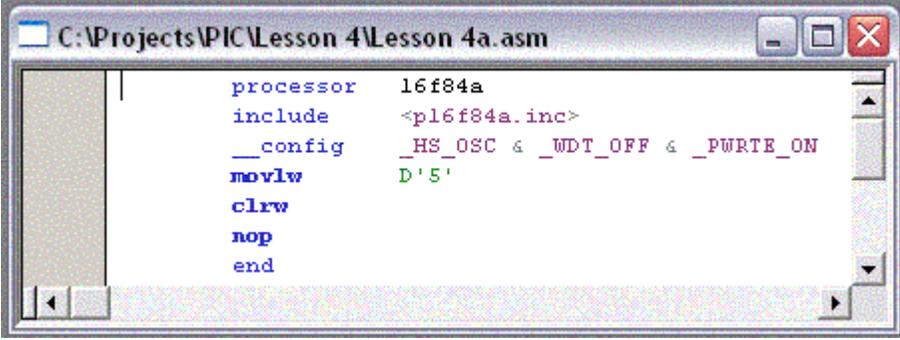
Now that we have the basic skeleton for all programs, we can go ahead and work over the actual instructions for our program. In this lesson, we aren't going to do a lot useful. Our point here is to get to understand how some of the basic instructions work.

At this point, you may find it useful to find the file for the quick reference card, and print out the page titled '14-Bit Core Instruction Set'. Throughout this course we will be referring to this page. There are other parts of the card that are interesting, but this particular page is the one that will get dog-eared.

Our first instructions

We are going to begin with the simplest of instructions. When we enter instructions, we place them after the `__config` directive and before the `end` directive. For our experiments right now, we need a `nop` instruction right before `end`. This is the simplest of instructions, it does nothing!

Let's add two more instructions before our `nop`, a `movlw D'5'` and a `clrw` instruction. These instructions move the number 5 into the W register, then clear it. Our program should now look like this:



```
processor 16f84a
include <p16f84a.inc>
__config _HS_OSC & _WDT_OFF & _PWRTE_ON
movlw D'5'
clr
nop
end
```

Assembling the program

As before, select 'Project->Build All'. With a little luck, you should get the friendly 'BUILD SUCCEEDED'. You can also select the Build All toolbar button:



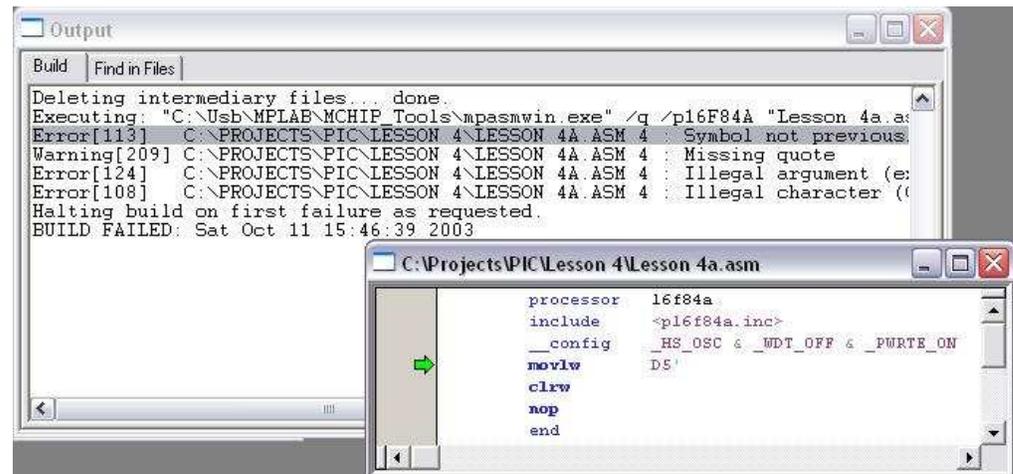
or simply hold down the Ctrl key and press F10.

Continued on next page

Adding Some Instructions, Continued

Suppose there was an error

If we had a typo, this can cause the assembler to get confused and give us a lot of error messages. Don't be concerned if you see a long list of messages. If we left off one of the quotes around the 5 in the `movlw D'5'` instruction we might see something like this:



Double-clicking the error message will cause MPLAB to put a green arrow left of the offending line. It's always good to look at the first error first. The remaining messages could be a result of the first. In this case, they are all on the same line, but sometimes an error on one line causes another line to be in error, so correct the first error first.

Let's see what happens

Once we get the program to assemble correctly, we want to see whether it does what we expect.

From the main menu, select 'Debugger->Select Tool->MPLAB SIM'. Now select 'Debugger->Reset->Processor Reset F6'

Notice at the bottom of the window it says 'pc:0' and 'W:0'. This says that the program counter is pointing at the first address in program memory, zero, and that the working register, W, contains a zero.

Select 'Debugger->Step Into F7'. Several things happen. First, the green arrow moves down one line in our program. At the bottom of the window, it now says, pc:0x1 and W:0x5. The 0x business is a way of warning us that the numbers we are looking at are in hexadecimal. The program counter has incremented by one, as we would expect, and the W register contains a 5, which is what we told it to do with the `movlw D'5'` instruction.

Now press F7 (or select 'Debugger->Step Into F7' again). The green arrow moves yet again, the bottom of the screen changes telling us that we have incremented the program counter one more time, and have cleared the W register, just like we told it to do.

Continued on next page

Adding Some Instructions, Continued

Some more playing with the simulator

Now add a few more lines so our program looks like this:

```
processor    16f84a
include     <p16f84a.inc>
__config    _HS_OSC & _WDT_OFF & _PWRTE_ON
Spot1      equ      H'30'
movlw      D'5'
movwf      Spot1
clrw
clrf       Spot1
nop
end
```

Assemble the program, and select View->File Registers. Arrange the windows so you can see both the program source and the file register window.

Select 'Debugger->Clear Memory->File Registers' and reset the processor (F6). Now as we press F7, there are several things to watch. On the first F7, besides the pc and w changing at the bottom of the screen as before, notice that location 0x02 in the file register also changed to a 0x01. This is because the low 8 bits of the program counter are mapped into location 0x02 of the file register.

The next time we press F7, besides 0x02 of the file register, 0x30 also changes. This is because we used that location to store our value `Spot1`. If we don't want to remember where we put things when we are debugging, we can click on the 'Symbolic' tab of the file register display. When we scroll down to 0x30 we can see the name, `Spot1`, on the right.

Press F7 again and our W register again goes to zero, and yet again and that zero gets stored in `Spot1`.

Continued on next page

Adding Some Instructions, Continued

Let's do some Arithmetic

OK, so we've loaded a number into both the working register and the file registers. Now let's do a little something with those values.

Change our program yet again to look like this:

```
processor    16f84a
include     <p16f84a.inc>
__config    _HS_OSC & _WDT_OFF & _PWRTE_ON
Spot1      equ    H'30'
Spot2      equ    H'31'
           movlw  D'5'
           movwf  Spot1
           movlw  D'2'
           addwf  Spot1,W
           movwf  Spot2
           movlw  D'3'
           subwf  Spot2,W
           movwf  Spot1
           clrw
           clrf   Spot1
           nop
           end
```

Now as we step through the program, we will see us storing the 5 in `Spot1` like before, but then we will load a 2 into the W register, and add it to `Spot1`, then store the result in `Spot2`. Next, we will move a 3 into the W register, subtract that from `Spot2`, and store the result in `Spot1`.

Notice the '`,W`' on the add and subtract instructions. These instructions can store the result either into the W register, or the original memory location. If we had wanted the result to go back into the file register, we would have used '`,F`' instead of '`,W`'.

Helping to Understand Our Program

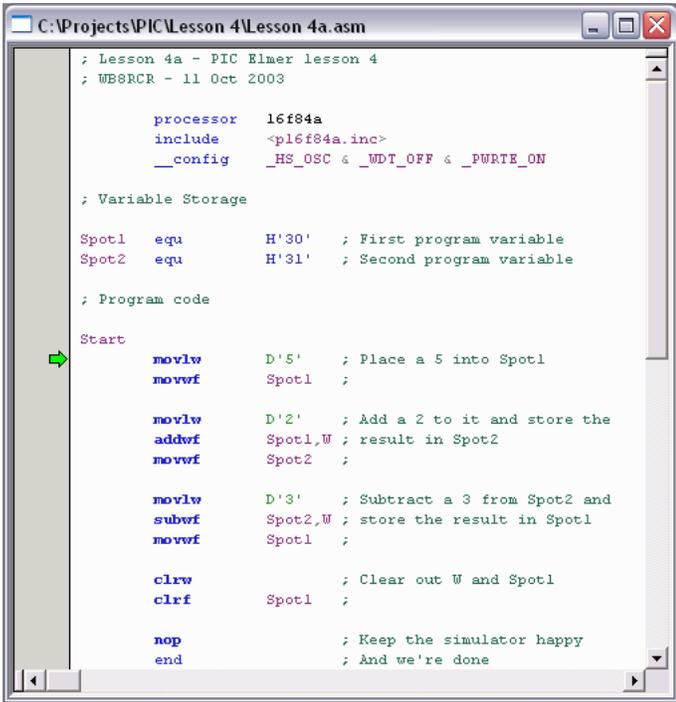
Introduction

So far, we've worried only about the specific instructions that make up the program. As we develop programs, they can get to be a little long. We need some aid in understanding the program, especially when we come back to it after being away a few days, or weeks.

Comments

The assembler allows us to put comments in our code. Whenever the assembler encounters a semicolon, everything after that on the same line is ignored. The assembler also allows us to have lines that are entirely blank, which can help us with readability.

The following assembly is exactly equivalent to what we had before:



```
C:\Projects\PIC\Lesson 4\Lesson 4a.asm
; Lesson 4a - PIC Elmer lesson 4
; WB8RCR - 11 Oct 2003

processor    16f84a
include     <pl16f84a.inc>
__config   _HS_OSC & _WDT_OFF & _PWRTE_ON

; Variable Storage

Spot1 equ   H'30' ; First program variable
Spot2 equ   H'31' ; Second program variable

; Program code

Start
movlw     D'5' ; Place a 5 into Spot1
movwf     Spot1 ;
movlw     D'2' ; Add a 2 to it and store the
addwf     Spot1,W ; result in Spot2
movwf     Spot2 ;
movlw     D'3' ; Subtract a 3 from Spot2 and
subwf     Spot2,W ; store the result in Spot1
movwf     Spot1 ;

clrwf     ; Clear out W and Spot1
clrf     Spot1 ;

nop      ; Keep the simulator happy
end      ; And we're done
```

Notice something else here. The second column is all blue. MPLAB colors PIC instructions and assembler directives that it recognizes as blue. The instructions are bold, while the directives are not. Comments are colored green. If we type in something and it shows up the wrong color, this is a red flag (well, maybe a purple flag) that perhaps we fat-fingered something.

Incrementing and Decrementing

Introduction

Over and over again in our programs we need to increment or decrement a counter. The PIC provides a number of instructions for this. First, let's look at the `incf` and `decf` instructions.

Like the `addwf` and `subwf` instructions, the `incf` and `decf` instructions take a memory location and a destination as operands.

Adding to our program

Add the following 4 lines to the program near the end, just above the NOP instruction:

```
incf    Spot1,F    ; Bump up Spot1 twice
incf    Spot1,F    ;

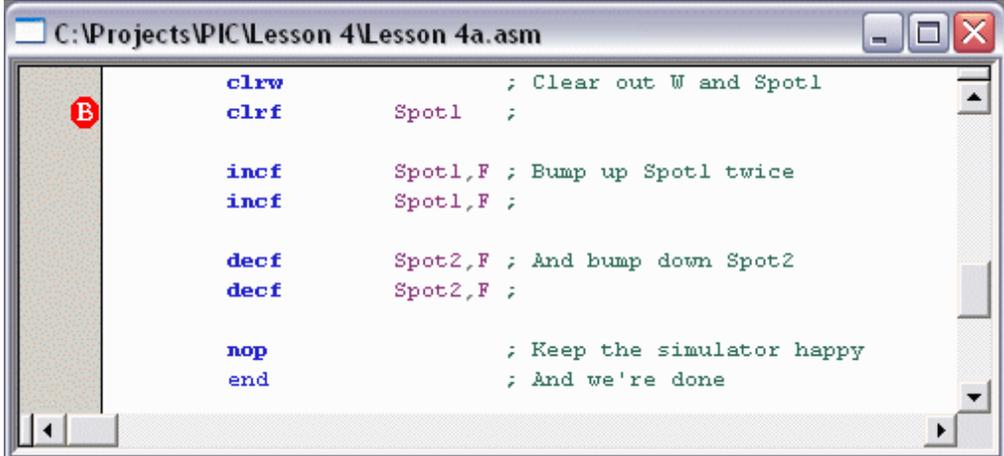
decf    Spot2,F    ; And bump down Spot2
decf    Spot2,F    ;
```

And assemble the program to test for typos.

Testing

Now, we have a lot of stuff in our program that we already know works, and we don't really want to go stepping through the whole thing again. To help us out, the simulator has the idea of a *breakpoint*. Basically, a breakpoint tells the simulator we can go run the program without stopping until we reach the breakpoint.

Right-click on the `clrf` instruction right above our new code, and select 'Set Breakpoint' from the popup menu. A red hexagon with a B in it will appear to mark the breakpoint.



The screenshot shows a window titled "C:\Projects\PIC\Lesson 4\Lesson 4a.asm". The code in the window is as follows:

```
clrw    ; Clear out W and Spot1
clrf    Spot1 ;

incf    Spot1,F ; Bump up Spot1 twice
incf    Spot1,F ;

decf    Spot2,F ; And bump down Spot2
decf    Spot2,F ;

nop     ; Keep the simulator happy
end     ; And we're done
```

A red hexagon with a white 'B' is positioned to the left of the `clrf Spot1` instruction, indicating a breakpoint.

Now, as before, arrange the File Register window so you can see it, reset the processor and clear the File Register memory. However, instead of stepping through the program, select 'Debugger -> Run F9' or press F9. The green arrow moves to the breakpoint indicator, and the File Register memory gets set to where it would be just before the `clrf Spot1` instruction is executed.

Now we can single-step through the remaining instructions like we did earlier, and see that the `Spot1` location gets incremented twice, and then the `Spot2` location gets decremented twice, just as we would expect.

Bit Manipulation

Introduction

Lots and lots of times, especially when we are doing embedded applications, we need to manipulate individual bits within a byte, or perhaps parts of bytes. The PIC provides a complete set of bit manipulation instructions, which we will explore here.

Another Project

Rather than continuing on with our previous program, let's make a new project. This time, call the project Lesson4b, the assembler source Lesson4b.asm, but let's keep it in the Lesson 4 folder.

Let's be boring and start the program off with our favorite four directives, and yes, let's add in our `nop` instruction for now. And of course, let's not forget to add the .asm file to the project.

AND

Before the `nop`, first define a file register location, let's be real creative and call it `Loc1`, then load the working register with a 7, save it in `Loc1`, then load the working register with a 12:

```

processor    16f84a
include     <p16f84a.inc>
__config   _HS_OSC & _WDT_OFF & _PWRTE_ON
Loc1       equ     H'20'
movlw      D'7'
movwf      Loc1
movlw      D'12'
nop
end

```

Now, assemble the program, select the debugger, and step through the program. You should see no surprises. Each of these instructions works as they did before. Now, before the `nop`, add

```
andwf      Loc1,W
```

Now when we step through the program (after assembling it, of course), what do you suppose will happen to the W register? If you skipped ahead and ran the simulator already, you see the result was 4. But why?

The `andwf` instruction told the processor to perform a bitwise AND of the contents of the W register with the contents of `Loc1`, and place the result in the W register. This means the whenever a bit is on in both the W and `Loc1`, the corresponding bit in W will be turned on:

0	0	0	0	1	1	0	0	W register starting
0	0	0	0	0	1	1	1	Loc1
0	0	0	0	0	1	0	0	W register ending

Continued on next page

Bit Manipulation, Continued

AND (continued)

Like other instructions that combine the W register and the file register, we can store the result in the file register. Change the ,W on the `andwf` instruction to a ,F and observe the result.

Notice that the W register remains a 12 (0xc), but file register location H'20' is set to a 4.

Most often, we use the AND function to turn off specific bits or groups of bits. For example, if we wanted to turn off the low order 2 bits in a number, we could AND it with B'11111100' (H'fc'). The result would be the same as the original, except with the rightmost two bits turned off.

Inclusive OR

The inclusive OR is almost the exact opposite of the AND. For a result bit to be on, either of the source bits may be on. Let's try it. Before the `nop` add:

```
iorwf  Loc1,F
```

Since the working register was H'0c' and `Loc1` was a 4, the result, stored at `Loc1`, was a H'0c'. Ok, not so satisfying, so lets add a:

```
movlw  D'3'
```

before our `iorwf`. Now we would expect `Loc1` to contain a 7 (3 IOR 4) when we are done. Try it.

Exclusive OR

The exclusive OR works just like the inclusive OR, except that a bit will be false if both operands are true.

At the end of the previous experiment, `Loc1` should have contained a 7, while W contained a 3. Add three lines before the `nop`:

```
xorwf  Loc1,F
xorwf  Loc1,F
xorwf  Loc1,F
```

Notice when we step through this, the low order 2 bits of `Loc1` toggle, first off, then on, then off again. This is easier to see if you select the 'Symbolic' tab of the File Registers view where you can see the decimal and binary representations as well as the hexadecimal representation of `Loc1`:

The screenshot shows a window titled "File Registers" with a table of memory locations. The table has columns for Address, Hex, Decimal, Binary, Char, and Symbol Name. The entry for Loc1 at address 0020 shows a Hex value of 07, a Decimal value of 7, and a Binary value of 00000111. The Symbol Name column shows "Loc1". Below the table, there are radio buttons for "Hex" and "Symbolic" views.

Address	Hex	Decimal	Binary	Char	Symbol Name
001F	00	0	00000000	.	
0020	07	7	00000111	.	Loc1
0021	00	0	00000000	.	

Continued on next page

Bit Manipulation, Continued

Complementing

Sometimes we want to invert all the bits. The `comf` instruction does this for us. It would be just like `xorwf` if the W register contained a H'ff', but we don't need to use the W register.

Again, add to our program:

```
comf    Loc1, F
```

before the `nop`. Notice that all the bits get inverted, but the W register is unchanged. If we wanted, we could have had the result placed in the W register, which would have left `Loc1` unchanged.

Bit set and clear instructions

The final two instructions `bcf` and `bsf`, set and clear an individual bit in a file register location. Unlike the other logic instructions, these instructions take a bit *number* as an argument.

At the end of our program, just before the `nop`, add:

```
bcf     Loc1, 0
bcf     Loc1, 1
bsf     Loc1, 2
```

These instructions are especially useful on the I/O ports, when we want to change the state of a particular pin, without having to concern ourselves with the states of all the other pins.

The Simulator

Introduction	We dove right in and started using this ‘Simulator’ thing, but what is it, anyway?
The Simulator	<p>The MPLAB is an Integrated Development Environment or IDE. It’s actually a shell that runs a number of other programs in a fairly seamless way. We’ve used the editor pretty extensively without talking much about it. We could have used any old editor we like. We’ve also used the assembler. When we installed MPLAB, we got a choice added to our Start menu that allows us to run the assembler separately. If we wanted, we could have edited our source files in Notepad, and run the assembler from the Start menu.</p> <p>The Simulator is yet another program we run from the IDE. It takes the assembly results and pretends to be a PIC. It interprets the PIC codes and does what they say to do, as if it were a PIC. This interpreting business is complicated, though. In spite of the fact that our PC is probably 100 times faster than a PIC, the simulator runs many times slower than a real PIC.</p>
Why would we do such a thing?	<p>When we run a program in a PIC, it’s pretty hard to see what is going on. We have no way to examine the registers, and most PIC programs have fairly few outputs that are satisfying to watch. We could include instructions in our program to wiggle some line or another to let us know where it is, and then follow those lines with a scope or perhaps attach an LED to the line, but this is pretty clumsy.</p> <p>By pretending to be a PIC, the simulator lets us run the program and see inside. We’ve already seen how we can step through one instruction at a time, examine the registers, and even have the program run until a particular instruction.</p> <p>What we haven’t seen (yet) is how we can change the values in the registers and see how our program reacts, or simulate different types of stimuli on the I/O pins to allow us to work through our program’s logic.</p> <p>The simulator is a very powerful tool for debugging our programs. In the early lessons in this course, we will use the simulator because we haven’t developed the skills needed to debug our program in actual hardware. As we get more proficient, we will test programs on the actual PIC, but even then, we will find it helpful to run our program, or parts of it, on the simulator so we can see what’s going on.</p>
About the nop	<p>The other little detail we never mentioned is that <code>nop</code> instruction we keep putting at the bottom of the program. That is actually for the simulator. In a real PIC program, we would never let the program reach the <code>end</code> directive. To do so would allow the PIC to start executing code in a part of memory where we never stored anything. The results are unlikely to be satisfying, and likely to be unpleasant!</p> <p>To simulate this behavior, the simulator goes off into never never land when it executes the <code>end</code> directive. As a consequence, we wouldn’t be able to see the result of our last instruction. The <code>nop</code> gives us a little room.</p> <p>It would probably be better to use something like:</p> <pre style="text-align: center;">A goto A</pre> <p>But we haven’t talked about <code>goto</code> yet.</p>

Wrap Up

Summary

In this lesson, we have experimented with those instructions that manipulate the working register and the file registers, but have few other effects. This group of instructions comprises fully half of the PIC 16F84A instruction set.

We have also used the simulator to see what those instructions did. The simulator will be our primary tool for understanding our programs as we go forward.

Coming UP

So far, our instructions have done things, but there has been no way to make decisions. Using what we have learned, all our programs have to go in a straight line, and do exactly the same thing every time.

In the next lesson, we will look at some instructions that affect the status register, and instructions that allow us to test the status register. This is where we start to be able to develop some interesting behavior.