Lesson 5
Let's play with the Status Register

Overview

Introduction	In the previous lesson we saw some of the simpler instructions in the PIC. Many of the more interesting instructions affect a special register called the status register. In this lesson, we will explore those instructions.
In this section	Following is a list of topics in this section:

Introduction

Introduction

One of the special registers in the PIC is called the Status register. It is mapped into the file register address space like most registers, but it is important enough that not only it's location, but many of its bits, have special names assigned in the processor's include file.

Most of the arithmetic and logical instructions affect the status register. There are a number of instructions whose behavior is affected by the status register.

The Status Register

Like every other file register location, the status register has eight bits. However, each bit in the status register has a special purpose. The 16F84A data sheet includes a picture of the status register that looks something like this:

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**	**TO**	**PD**	**Z**	**DC**	**C**
bit7							bit0

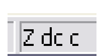
The most significant three bits have to do with memory. **IRP** and **RP1** are not used in the PIC16F84A, and must be zero. We will talk about **RP0** in some detail a little later in the course.

The **TO** bit is set to one when the processor is powered up, or whenever a `clrwdt` (clear watch dog timer) or `sleep` instruction is executed. It is set to zero when a watchdog timer timeout occurs.

The **PD** is similar. It is set to one at power up or when a `clrwdt` instruction is executed. It is set to zero by the `sleep` instruction.

Together, these two bits allow us to respond when we are interrupted from a `sleep` instruction or from a power down.

The right most three bits are the ones we will talk about today. Their values are displayed at the status bar at the bottom of the MPLAB workspace as "Z DC C". Each will be shown in upper case when it is a one (set) and in lower case when it is a zero (cleared). These bits are affected by many (but not all) of the arithmetic and logic instructions.



The **Z** bit being set means that the execution of an arithmetic or logic statement resulted in a zero. **DC** means that there was a carry out of bit 3 of an arithmetic operation (digit carry). **C** means that there was a carry out of bit 7 of an arithmetic operation. In this lesson, we will explore how these are manipulated, and how we use those results.

Instructions affecting the Z, C, and DC bits

Introduction	If you turn to page 36, Table 7-2, in your PIC16F84A Datasheet, you will see a summary of the instruction set with an indication of how each instruction affects the status register. At first this may seem a bit imposing – there are a lot of possible combinations scattered through the instruction set, apparently randomly. In fact, it's not so bad. With a very few exceptions, those instructions that affect the status bits are those instructions where it *makes sense* for them to affect the status bits.
The Exceptions	Let's first take a look at the oddballs. There are only a few. The `movwf` and `swapf` instructions don't affect the status register, even though their result may be zero. Similarly, the literal loads, `movlw` and `retlw` don't affect the status byte, either. The various instructions that manipulate individual bits, `bcf`, `bsf`, don't affect the status register even though `bcf` clearly could have a zero result. Probably the most surprising are the increment/decrement F with skip instructions, `incfsz` and `decfsz`. Even though these instructions test whether their result is zero, they do not affect the Z bit.
The Arithmetic Instructions	The arithmetic instructions, `addwf`, `addlw`, `subwf` and `sublw` all have the effect we would expect on the status bits. If you notice, these are the only instructions that can affect all three of Z, C, and DC. If you think about it for a minute, they are the only ones where that makes sense. If we perform an add or subtract, and the result is zero, then the Z bit will be set. This is what we would expect. If we perform an add, and the operation results in a carry (for example, F contained 253 and we added 7), then the C bit will be set. Subtract is a little trickier. If we set the C bit, then perform a subtract operation which results in a borrow, the C will be cleared. The DC (digit carry) is similar to the C except that it depends only on the low order 4 bits. So for example, adding a 1 to 15 will result in the DC bit being set. This is useful if we are formatting data for a display, for example, and have stored a digit in each 4 bits of a file register location. The increment and decrement instructions, `incf` and `decf`, affect only the Z bit.
The Logic Operations	The logic operations, `andwf`, `andlw`, `iorwf`, `iorlw`, `xorwf`, `xorlw`, `clrf`, `clrw` and `comf` affect only the Z bit. This makes sense, since for none of these operations would a carry be the sort of thing you would expect.

Testing the Status Register

Introduction

In this section, we will experiment with the arithmetic and logic instructions and see how they affect the various bits in the status byte.

Set up the project

Yet again, create a folder, Lesson5, and a project Lesson5a. Add a single source file, Lesson5a.asm, to the project. In later lessons, we won't even mention this step anymore. Whenever you want to start a project make a folder for it, and any related projects, create the project in MPLAB, and add in a source file.

Add some code

Insert the following code:

```
; Lesson 5a - PIC Elmer lesson 5
; WB8RCR - 17 Nov 2003

        processor   16f84a
        include     <p16f84a.inc>
        __config    _HS_OSC & _WDT_OFF & _PWRTE_ON

; Variable Storage

        cblock      H'20'
            Spot1           ; First program variable
            Spot2           ; Second program variable
        endc

; Program code

Start
    ;   Clear the status bits so we know their state
        bcf         STATUS,Z
        bcf         STATUS,C
        bcf         STATUS,DC

    ;   Show how clrf affects the Z flag
        clrf        Spot1   ; Clear out Spot1

    ;   Show how a carry out of bit 7 affects the C flag
        movlw       H'f0'   ; Store H'f0' in Spot2
        movwf       Spot2   ;

        movlw       H'10'   ; Add a H'10' to Spot2
        addwf       Spot2,W ;

        nop
        end                 ; And we're done
```

OK, maybe that's a little long. Let's talk about it.

Continued on next page

Testing the Status Register, Continued

cblock

Ok, what's this `cblock` stuff?

If you recall, in earlier lessons, we allocated locations in the file register for our various memory needs, and we assigned names to their locations with `equ` statements. There's nothing wrong with this. But the `cblock` directive has a number of advantages.

The sequence

```
            cblock      H'20'
                  Spot1             ; First program variable
                  Spot2             ; Second program variable
            endc
```

Is exactly the same as

```
      Spot1 equu        H'20'       ; First program variable
      Spot2 equ         H'21'       ; Second program variable
```

But has the advantage that the assembler keeps track of adding one each time we use another location. Obviously, this isn't a big win for only 2 locations. But as our programs get longer, it's a bigger help.

There's another reason we want to use this construct for allocating file register memory. If we later decide we want to use a different PIC model, this can save us some work in modifying the code for the new processor. For example, the PIC16F84A has file register memory starting at H'0C'. If we run out of program memory and decide to move to a PIC16F628, we have more program memory as well as file register memory, but the file register starts at H'20'. We may have dozens of lines to edit if we used the `equ` form, and plenty of opportunity for errors. With the `cblock`, we have only one directive to change.

We will continue to use `equ` to define manifest constants, and this convention has the additional advantage of making our memory allocation definitions stand out from constant declarations.

Watching it play

Now, we want to assemble the code and start the simulator, like we did in the previous lesson.

Before clicking Step Into for the first time, notice at the bottom of the workspace the status bar entry for the status byte. Typically, when you first start MPLAB, these will all be lower case, indicating that the Z, DC and C bits are all clear. Notice that this isn't necessarily the case on the PIC after a reset.

To be sure that we know the initial states, the first thing we do is to clear those three bits with the first three instructions. Clicking 'Step Into' three times will do not much more than increment the program counter.

Continued on next page

Testing the Status Register, Continued

Watching it play (continued)

However, the fourth click, executing the `clrf` instruction, will cause the Z to become upper case indicating the Z bit in the status byte has been set. This indicates that the result of the instruction was zero. Note that not all instructions with a zero result will set the Z flag. You should check Table 7-2 in the datasheet if this matters to you for a particular instruction. If we hadn't been doing anything before, the file register will contain all zeroes, so we won't see any effect of the `clrf` instruction there. Again, in actual hardware the file register powers up with random contents, so if we expect a register to contain zero, we need to put the zero there.

Next, we're going to put a value into `Spot2`. Notice that neither the `movlw` nor the `movwf` instructions affect the Z bit. Moving the `H'10'` into W doesn't affect any status bits either, but notice when we do the `addwf` that the C bit becomes set. Remember what we did was to add a `H'10'` to `H'f0'`. We would expect the result to be `H'100'`, but the working register can't hold any number greater than `H'ff'`. The result is a ***carry*** out of bit 7, which is recorded by setting the C bit in the status register. Had we instead added, say, a `H'06'` to `H'f0'` the result would have been `H'f6'` and we would not have recorded a carry.

Digit Carry

Now, let's add a little more code:

```
; Show how a carry out of bit 3 affects the DC flag
        movlw    D'15'    ; Store a 15 (H'0f') in spot2
        movwf    Spot2    ;
        movlw    D'03'    ;  Add a 3 to Spot2
        addwf    Spot2,F  ;
```

Now, stepping through this code, notice that the `addwf` causes a carry out of the low nibble, resulting in the DC (digit carry) bit being set.

Ending our test code

Introduction	Up to now, we have always ended our programs with a `nop`. The reason for this is that when the simulator runs off the end of the program it sets some of the register contents to random values. In an actual PIC, running off the end of the program will result in unpredictable behavior.

A better choice	The problem with this approach is that if we click step one too many times, the result we were looking for may have been lost. Further, sometimes we want to run the simulator's animate, and we would like a friendlier result.
	
	Replace the `nop` with the following:
	
	`alldone`
	`        goto    alldone  ; Keep the simulator happy`
	
	The `goto` instruction, obviously, causes the program control to transfer to the label specified. By looping like this, we never run off the end of the program, and we don't affect any of the registers, either.
	
	In real programs, we will generally loop back to somewhere near the start of the program. Typically, we want the PIC to do something over and over, so our loop will include all of our program except, perhaps, for some initialization.

Subtraction

Introduction	When we perform an addition we can have a carry, just like we would if we were adding numbers manually. In subtraction, we can have a ***borrow***, again, just like we were doing it on paper. (Yes, Matilda, it really is possible to do a subtraction on paper.)

The code	To do a subtraction, we want to initially set the C bit, so it is available to borrow *from*. Before `alldone`, set up the following code:
	`;  set up a subtraction`
	`      movlw    H'03'`
	`      bsf      STATUS,C`
	`      subwf    Spot2,F`
	
	`      movlw    H'0f'`
	`      subwf    Spot2,F`

Testing the code	Now, step down until you are ready to execute the `movlw H'03'`. Notice that at this point the file register location `Spot2` contains a `H'0f'`, leftover from the add. Also, the carry bit is clear. Stepping once we change the W but nothing else. However, when we execute the `bsf STATUS,C`, the C bit becomes set. Remember, both `STATUS` and `C` are defined in p16f84a.inc. We could have just as easily said `bsf H'03','H'00'`, but it's easier to remember the mnemonics. Now, when we step again, `Spot2` changes to `H'0c'` but the carry bit remains set. This is because we didn't need to do a borrow for the subtraction. Now we'll load a `H'0f'` into the W. This is larger than `H'0c'` so when we do the subtraction, we borrow the carry, and end up with the result `H'fd'`.

Two's Complement Arithmetic

Introduction	In the previous map, we subtracted 15 from 12 and got a result of 253. Had we stopped and thought about that for a moment, we might have questioned that result. What's happening here is a thing called two's complement arithmetic.
Negative Number Representation	Back in the early days of digital computers, there was some debate about how to represent negative numbers. For whatever reason, very early on it was agreed that having the high bit of a value be true would represent a negative number. For a very few early computers, that's all that was done. If a 2 was represented by `B'00000010'`, then a -2 would be represented as `B'10000010'`. This had the problem that the values `B'00000000'` and `B'10000000'` both represented zero. This turns out to be messy, though. When doing arithmetic this way, there is an odd transition going from positive to negative. Another scheme which was fairly popular in the 60's was to use one's complement arithmetic. In this scheme, to make a number negative, you simply reverse all the bits. So our -2 would be represented as `B'11111101'`. This has some appeal, but it did make for a little bump right around zero. Again, we have 2 values for zero: the value `B'00000000'` and `B'11111111'` both represented zero. Eventually, the world settled on a scheme called two's complement. In this scheme, to make a number negative, you invert all the bits and add one. So, to take our `H'fd'` (`B'11111101'`) and make it negative, we invert all the bits (`B'00000010'`) and add one to end up with `B'00000011'` (`H'03'`). So, subtracting 15 from 12 results in -3, just as we would expect. We still keep the rule that if the high order bit is a one, then the value is negative. As a result, the range of numbers that can be stored in a byte (8 bits) is from -128 to +127. Practically all modern computers use 2's complement arithmetic.
It's all in how you look at it	One of the advantages of two's complement is that there are no unusual bumps in the math as we cross particular thresholds. As a result, we can *interpret* values in the range `H'80'` to `H'ff'` as either positive or negative, depending on what our application requires. If we were storing an RIT setting, we may choose to interpret the value of a byte as hertz (or tens of hertz) positive or negative of the VFO's setting. On the other hand, if we were storing a code speed, we might choose to allow the entire range of a byte to represent 0 to 51 WPM, in 0.2 WPM increments. Just because we *could* look at a value as being negative doesn't mean we have to. The beauty of two's complement arithmetic is that there is no penalty for making either choice.

Logic Instructions

Introduction	Besides adding and subtracting, we can and, or and exclusive or. These instructions each can affect the Z bit in the status register, but no others. Since this kind of operation doesn't have the opportunity for a carry or a borrow, this makes sense.
	
	There are six instructions in this category
	
	`andwf` `andlw`
	`iorwf` `iorlw`
	`xorwf` `xorlw`

Trying them out	Just before `alldone`, try a little code like the following:
	
	`movlw H'ee'`
	`andlw H'e0'`
	`iorlw H'e0'`
	`xorlw H'f8'`
	`andwf STATUS,F`
	
	And try it out. Notice that the Z bit is only affected when the result changes between zero and non-zero. Also notice the last instruction. We can apply the operation directly to the status register, in this case, since the W contained `H'f8'` (B'11111000') this had the effect of clearing the rightmost 3 bits of the status register.

Incrementing and Decrementing

Introduction

Back in lesson 4 we looked at incrementing and decrementing. This is something we do over and over, so we are going to revisit it here.

Simply counting up and down

In Lesson 4 we did some simple incrementing and decrementing, but we never looked at the status byte. Let's do almost the same thing we did there, but let's pay closer attention to this register.

Just before `alldone` again, add the code:

```
;  Show increment and decrement
    clrw                ; Clear 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    ;
```

Again, assemble the program, skip down to the start of this code, and let's watch what happens.

First the `clrw` sets the Z bit since the result is zero. Next, `clrf` with another zero result leaves it set. The increments and decrements, having non-zero results, leave the Z bit clear.

Increment and decrement instructions don't affect the C or DC bits, although you may think they should. The only status bit they affect is the Z bit.

Looping

There is another pair of increment/decrement instructions. They are `incfsz` and `decfsz` (increment F and skip if zero, likewise for decrement). Try this (again, before `alldone`):

```
      ;  Lets do a counter
         clrf      Spot1
loop
         incsfz    Spot1,F
         goto      loop
```

Now, when we run this, watch what happens to the file register location Spot1.

Notice that the first time the `incsfz` is executed, the file register gets bumped up to one. Two more clicks of the Step Into button and it becomes two.

Now select `Debugger->Animate` and watch the file register. The program runs freely, but the screen is updated after each instruction so we can watch the file register location increment. When it wraps around to zero, the program leaves the loop (because the `incsfz` instruction finally skipped the `goto`) and reaches our `alldone` loop.

The `incsfz` instruction changes none of the status bits, but it does take action (skipping the next instruction) when the result is zero.

Bit Testing

Introduction

While not specifically "instructions that affect the status register", there are two instructions that are used frequently with the status register, `btfss` and `btfsc`. These instructions test a particular bit in a file register cell, and skip the next instruction if the bit is set (`btfss`) or clear (`btfsc`).

While these instructions may be used on any file register location, they are very frequently used to test the condition of a bit in the status register.

Example

Consider the following code snippet:

```
        ; Bit testing
            movlw    D'03'      ; Initialize Spot1
            movwf    Spot1      ;
      loop2
            clrw                ; Test whether Spot1 is
            xorwf    Spot1,W    ; zero by xoring it with
            btfsc    STATUS,Z   ; a zero
            goto     donebt     ; If zero, we're done
            decf     Spot1,F    ; Otherwise do work
            goto     loop2      ; and go try again
      donebt
```

We set a value into a location. By performing an XOR operation with a zero on the location, we set the Z bit to reflect whether the cell contains a zero. (Notice that performing an XOR operation with anything doesn't change the original).

While this particular snippet may look a lot like our increment loop, it has the feature that the `Spot1` location never actually gets below zero. If we were wanting to limit the range of some parameter we might use this kind of approach.

Rock and Roll

Introduction

There is one more pair of instructions that affect the status register. The `rlf` and `rrf` instructions rotate the specified file register location left or right, and include the C bit in the rotation. In the case of `rlf`, each of the bits in the file register location get moved left one bit. The carry bit gets moved into bit 0, and bit 7 gets moved to the carry. The `rrf` is the same, except the bits are moved to the right, bit 0 goes into the carry, and the carry goes into bit 7.

You might wonder why I would want to do such a thing. Well, there are two really common uses. Perhaps most obvious, rotating a byte left multiplies the value by two. Rotating right divides by two. If I need to do a multiplication or division by a power of two (pretty common, actually), these instructions are orders of magnitude faster than a full blown multiply or divide.

Perhaps more common, though, is in serial communications. If I want to communicate with something and not use a whole bunch of pins, I need to send the bits out one after the other. This is useful not only in RS-232 communications to a PC; A/D converters, external EEPROMs, DDS chips all use this kind of communication.

Another test

OK, let's try the following code:

```
    ; Rock and roll
      movlw    B'01100010'  ; Place a pattern to rotate
      movwf    Spot2        ; into Spot2
      movlw    H'f8'        ; Will rotate it 248 times
      movlw    Spot1        ;
loop3
      rlf      Spot2,F      ; Rotate the word
      decfsz   Spot1,F      ; Count down the number of rlf's
      goto     loop3        ; do it again
```

If you haven't already figured it out, using the simulator's run instruction to run up to a breakpoint is a whole lot faster than stepping through these loops we've written. Step through the code noticing what happens with the carry bit. Then, arrange your windows so you can see the binary representation of `Spot2` in the file register window (Symbolic tab) and click Animate (two blue arrows on the toolbar). In the binary view, you will be able to see the bits walk through the byte.

Wrap Up

Summary	In this lesson, we have examined the instructions that affect the status register, and the instructions that test bits so we can examine the results. We have also begun to see how to implement program flow control; we have used the `goto` instruction to cause our program to do something other than go in a straight line, and we've used some of the instructions that allow us to change the flow of the program based on the results of earlier operations. At this point, we have seen most of the PIC instructions. The remaining instructions consist of those instructions that have to do with subroutines, and then a few odd instructions that have specialized uses.
Coming Up	In the next lesson, we will examine the use of subroutines. Subroutines are little packages of logic that we can use over and over again in our programs. They are key to keeping our programs understandable, and to make maximum use out of the relatively limited resources in the PIC.