Lesson 7
Multiple Precision Arithmetic

Overview

Introduction

The PIC stores data as 8-bit bytes. Up until now, we have only used values of 8 bits or less. But what if we have a greater range of values? In this section, we will examine addition and subtraction of larger numbers.

In this section

Following is a list of topics in this section:

Representing Larger Numbers

Introduction

The PIC has an 8 bit data word. How can numbers larger than 255 be handled? The answer is to use multiple bytes to store the value.

Interpretation

Throughout this course, we will be faced with the reality that symbols in our programs only have what meaning that we assign to them. While this is true of higher level computer languages as well, in assembler, there is almost nothing that carries meaning by itself; things only mean what we want them to mean.

In lesson 4, we assigned a value to a symbol, and used that to reference a place in memory. The value, (Spot1 in that case) “meant” to us a location in the file register memory. In lesson 6, we assigned a value to the symbol XMTR and interpreted it as a bit number. These assignments can carry whatever interpretation we wish to apply.

The same is true of the values stored in memory. We have already talked about how a byte could be interpreted as a value from 0 to 255, or as a value from -128 to +127, depending on what we want. However, the byte might also represent a letter, several small numbers packed together, or even just a jumble of individual bits.

Multiple digit representation

This happens in normal arithmetic, too. In the decimal numbering system, a digit can have one of ten possible values, from zero through nine. However, we can represent a number of arbitrary size by simply stringing digits together. Each digit to the left is interpreted as meaning ten times more than the digit to its right. So the graphic ‘6’ could represent a value of six, or sixty, or six hundred depending on how many digits are to the right.

Similarly, we could string bytes together and interpret them as representing successively higher values. Since a byte can have 256 different values, the individual bytes can represent the one’s place, the 256’s place, the 65536’s place, etc. Because each individual ‘digit’ has 256 possible values, instead of the 10 in decimal numbers, the values go up a lot faster. In just four bytes, we can represent over 4 billion possible values.

Suppose, for example, we wanted to represent 1000 decimal. 1000 decimal is 768 plus 232, so we could represent 1000 as a 3 (768 / 256) with a remainder (digit) of 232.

Little endian or big endian

You may have heard of the term ‘endian’. It refers to whether the least significant byte is stored first or last. In the PC, the least significant byte is stored in the lowest numbered memory cell (little endian). On the Mac, the most significant byte is stored in the lowest numbered memory cell (big endian). It turns out that each has its advantage, and on those machines, that structure is embedded in the hardware.

On the PIC, however, there are no instructions that access more than a single byte, therefore the PIC has no “endian-ness”. Authors tend to put the low order byte first in memory for binary values (and interestingly, the opposite for decimal values), but there is nothing that makes one choice better than another. Indeed, there is no advantage (other than readability) to having the bytes adjacent!

Adding Larger Numbers

Introduction

If we are going to use multiple bytes to represent larger numbers, we need to know how to add them if they are to be at all meaningful.

Adding Algorithm

When we add two numbers of more than one digit in decimal, what do we do? Well, we start with the least significant digits. We add them, if the result will not fit in a single digit, we take the part that will not fit, divide it by ten, and we *carry* it into the next higher digit. (We might not have thought of the divide part, but if we add eight and six, we get fourteen. We keep the four, and we carry the ten, but before we use it, we divide the ten by ten, giving us one.) A little detail you may not have noticed; if we are adding only two numbers, we never have a carry of more than one.

The same is true in binary. If we add the least significant bytes, and the sum exceeds what will fit in a single byte, we take the part that will not fit, divide it by 256, and we carry it to the next byte.

Just as we can in decimal, we can continue this process of adding and carrying for an arbitrary number of bytes, so we can add numbers of any size we wish.

An Example

Imagine we have three, two-byte memory locations set up by the following statements:

```
      cblock            H'40'[1]
            v1_lo             ; Variable 1, low byte
            v1_hi             ; Variable 1, high byte
            v2_lo             ; Variable 2, low byte
            v2_hi             ; Variable 2, high byte
            res_lo            ; Result, low byte
            res_hi            ; Result, high byte
      endc
```

We could add them as simply as follows (In a subroutine, of course):

```
Add16
      movf        v1_lo,W     ; Low byte first operand
      addwf       v2_lo,W     ; Add in low byte second operand
      movwf       res_lo      ; Store result
      movf        v1_hi,W     ; Pick up high byte first operand
      btfsc       STATUS,C    ; Was there a carry?
      addlw       H'01'       ; Yes, add in carry
      addwf       v2_hi,W     ; Add in high byte second operand
      movwf       res_hi      ; Store high byte result
      return
```

Notice that this is not any different from what we would do if we were adding decimal numbers by hand.

[1] In previous examples, these blocks have always started at H'20'. There is no magic to the location, though. File register addresses in the 16F84A run from H'0c' to H'4f'. As long as we start after H'0b' and end before H'50' we can choose whatever we want. One minor annoyance, though. The p16f84a include file defines _CP_ON as H'0f'. Since this definition is encountered before ours, if we start our use of the memory at H'0c', the file register display shows H'0f' as _CP_ON rather than whatever we have assigned. This does not affect how the program runs at all, but it is an annoyance when we are debugging.

Subtracting Larger Numbers

Introduction

Subtraction, like addition, simply follows the rules we learned in grammar school, except on binary, rather than decimal, values. We address the low order byte first, remember whether we had a borrow, and then subtract the borrow from the next higher-order byte.

Subtraction Routine

The following is an example subtraction using the same variables used above:

```
;      16 bit subtraction
Sub16
             movf         v2_lo,W       ; Low byte first operand
             subwf        v1_lo,W       ; Subtract low byte second
operand
             movwf        res_lo        ; Store result
             movf         v2_hi,W       ; High byte second operand
             btfss        STATUS,C      ; Borrow?
             sublw        H'01'         ; Yes, take borrow
             subwf        v1_hi,W       ; Subtract high byte
             movwf        res_hi        ; Store result
             return
```

This time we do the subtraction when the carry has been cleared because of the borrow, rather than, as in addition, when set because of a carry.

Although these examples show only two bytes, the same logic can be carried on for any arbitrary size number.

Testing the Routines

Introduction

In the previous pages, we have written two subroutines, one to do a 16-bit addition, and another for a 16-bit subtraction. Now let's write a mainline to test them out.

Writing a test program

If you have already entered the previous routines, and maybe assembled them, you have already built the project, added the assembler source file to the project, and put in the processor, configuration, and include statements. Typically, we like to have our subroutines at the front, so what you might not have thought of is the `goto` statement to skip over the subroutines to get to our mainline.

What we would like to do is to test both the addition and subtraction with and without a carry and borrow so we can see that we have covered all the cases. If we make `v1` be 516 and `v2` be 258 we can cover the non-borrow/carry cases :

```
; Set up arguments without carry/borrow
        movlw           H'04'   ; First value H'0204'
        movwf           v1_lo   ; = 516 decimal
        movlw           H'02'
        movwf           v1_hi

        movlw           H'02'   ; Second value H'0102'
        movwf           v2_lo   ; = 258 decimal
        movlw           H'01'
        movwf           v2_hi

; Do an addition
        call            Add16   ; Result should be 774
                                ; or H'0306'

; Do a subtraction
        call            Sub16   ; Result should be 514
                                ; or H'0102'
```

After you have stepped through each of the routines to satisfy yourself that you understand what is going on, try the following; Step down to the `call` instruction, then click on the step over button. The subroutine will be executed, but you will not be taken into it step by step. Instead, you will see just the results. This can be handy when you are confident with your subroutine logic, but you want to test a variety of different cases.

One of the differences between the 6.30 and 6.40 versions of MPLAB is that the 6.40 version includes a “Step Out” button. This allows you to step partway through a subroutine then skip over the rest of the subroutine and return to the calling program. This isn't nearly as handy as the step over button, but it can be useful in some cases.

Now, try testing the routine with values that require a carry and a borrow. 258 and 255 will work here. When you step through the subroutines, notice that in this case we take the path where we add in the carry (or subtract the borrow).

Continued on next page

Testing the Routines, Continued

Completed Example

The following is the entire program we have built up so far:

```
;=================================================================================
;         Lesson7a.asm - example of double byte arithmetic
;         8-Jan-2003
;=================================================================================
;
                    processor pic16f84a
                    include             <p16f84a.inc>
                    __config  _XT_OSC & _PWRTE_ON & _WDT_OFF

;=================================================================================
;         File register reservations
                    cblock              H'40'
                              v1_lo                         ; Variable 1, low byte
                              v1_hi                         ; Variable 1, high byte
                              v2_lo                         ; Variable 2, low byte
                              v2_hi                         ; Variable 2, high byte
                              res_lo                        ; Result, low byte
                              res_hi                        ; Result, high byte
                    endc

;---------------------------------------------------------------------------------
                    goto                Start               ; Skip past subroutines

;=================================================================================
;         Subroutines begin here
;
;---------------------------------------------------------------------------------
;         16 bit addition
Add16
                    movf                v1_lo,W   ; Low byte first operand
                    addwf               v2_lo,W   ; Add in low byte second operand
                    movwf               res_lo    ; Store result
                    movf                v1_hi,W   ; Pick up high byte first operand
                    btfsc               STATUS,C; Was there a carry?
                    addlw               H'01'     ; Yes, add in carry
                    addwf               v2_hi,W   ; Add in high byte second operand
                    movwf               res_hi    ; Store high byte result
                    return

;---------------------------------------------------------------------------------
;         16 bit subtraction
Sub16
                    movf                v2_lo,W   ; Low byte first operand
                    subwf               v1_lo,W   ; Subtract low byte second operand
                    movwf               res_lo    ; Store result
                    movf                v2_hi,W   ; High byte second operand
                    btfss               STATUS,C; Borrow?
                    sublw               H'01'     ; Yes, take borrow
                    subwf               v1_hi,W   ; Subtract high byte
                    movwf               res_hi    ; Store result
                    return

;=================================================================================
;         Mainline starts here
Start
          ;         Set up arguments without carry/borrow
                    movlw               H'04'     ; First value H'0204'
                    movwf               v1_lo     ; = 516 decimal
                    movlw               H'02'
                    movwf               v1_hi

                    movlw               H'02'     ; Second value H'0102'
                    movwf               v2_lo     ; = 258 decimal
                    movlw               H'01'
                    movwf               v2_hi

          ; Do an addition
                    call                Add16     ; Result should be 774
                                                                      ; or H'0306'

          ;         Do a subtraction
                    call                Sub16     ; Result should be 514
                                                                      ; or H'0102'

;---------------------------------------------------------------------------------
          ;         Set up arguments with carry/borrow
                    movlw               H'02'     ; First value H'0102'
                    movwf               v1_lo     ; = 258 decimal
                    movlw               H'01'
                    movwf               v1_hi
```

Continued on next page

Testing the Routines, Continued

Completed Example (continued)

```
                    movlw           H'ff'     ; Second value H'00ff'
                    movwf           v2_lo     ; = 255 decimal
                    clrf            v2_hi
; Do an addition
                    call            Add16     ; Result should be 513
                                                                  ; or H'0201'

          ;         Do a subtraction
                    call            Sub16     ; Result should be 3
                                                                  ; or H'0003'
Loop
                    goto            Loop

          end
```

Still Larger Numbers

Introduction

In the above examples, we used 16 bit numbers. This allows us to represent numbers from 0 to 65535 or from -32768 to +32767, depending on whether we want to interpret the number as signed or not. But, what if we want still larger numbers?

Extending the model

As we mentioned earlier, we can extend the model as far as we wish. If we have a number of `n` bits, it can take `2^n` values. Thus, 3 bytes (24 bits) can take `2^24` values, or 16,777,216 possible values. 4 bytes can take 4,294,467,296 possible values, which we can interpret as 0 to 4,294,467,295 (remember, zero counts as one of our 4,294,467,296 values) or as -2,147,483,648 to +2,147,483,647. It is relatively uncommon for us to need more than 4 bytes, but the same model can be extended indefinitely.

Homework Assignment 1

OK, we've been letting you off too easy. Time for some work. The first assignment is relatively straightforward. Represent in PIC assembler the 20 meter QRP calling frequency of 14,060,000 Hz. You may use three or four bytes, and you can represent the individual bytes in binary, hex, decimal, or octal, your choice.

Homework Assignment 2

This one is a little more interesting. Imagine, if you will, that we are building a receiver that uses the common 4.9152 MHz crystals for the IF filter. Write a short program that, given a receive frequency, calculates the local oscillator frequency so that later we can feed that to our DDS daughtercard.

Hint: This is no different that what we just did, except you will need to extend the routines to more bytes to get 1 Hz precision.

Wrap Up

Summary	In this lesson, we looked at how to represent numbers that are too large to fit in a single byte. We also developed routines for adding and subtracting these larger numbers.
Coming Up	Up until now, all of our programs have been totally self-contained within the PIC. We have had no way to interact with the user or with other circuitry. In the next lesson, we will look at how the file register memory in the PIC is banked, and how we interact with the outside world.