Lecture No.9

Lecture Outlines

4.2 Addition and Subtraction (Continued)

- 4.2.6 Flags Affected by Addition and Subtraction
- 4.2.7 Example Program (*AddSubTest*)

4.3 Data-Related Operators and Directives

- 4.3.1 OFFSET Operator
- 4.3.2 ALIGN Directive
- 4.3.3 PTR Operator
- 4.3.4 TYPE Operator
- 4.3.5 LENGTHOF Operator
- 4.3.6 SIZEOF Operator
- 4.3.7 LABEL Directive

4.2 Addition and Subtraction (Continued)

4.2.6 Flags Affected by Addition and Subtraction

When executing arithmetic instructions, we often want to know something about the result. Is it negative, positive, or zero? Is it too large or too small to fit into the destination operand? Answers to such questions can help us detect calculation errors that might otherwise cause erratic program behavior. We use the values of CPU status flags to check the outcome of arithmetic operations. We also use status flag values to activate conditional branching instructions, the basic tools of program logic. Here's a quick overview of the status flags.

- The Carry flag indicates unsigned integer overflow. For example, if an instruction has an 8-bit destination operand but the instruction generates a result larger than 11111111 binary, the Carry flag is set.
- The Overflow flag indicates signed integer overflow. For example, if an instruction has a 16-bit destination operand but it generates a negative result smaller than −32,768 decimal, the Overflow flag is set.
- The Zero flag indicates that an operation produced zero. For example, if an operand is subtracted from another of equal value, the Zero flag is set.
- The Sign flag indicates that an operation produced a negative result. If the most significant bit (MSB) of the destination operand is set, the Sign flag is set.
- The Parity flag indicates whether or not an even number of 1 bits occurs in the least significant byte of the destination operand, immediately after an arithmetic or boolean instruction has executed.
- The Auxiliary Carry flag is set when a 1 bit carries out of position 3 in the least significant byte of the destination operand.

> To display CPU status flag values when debugging, open the Registers window, right-click in the window, and select *Flags*.

Unsigned Operations: Zero, Carry, and Auxiliary Carry

The Zero flag is set when the result of an arithmetic operation equals zero. The following examples show the state of the destination register and Zero flag after executing the SUB, INC, and DEC instructions:

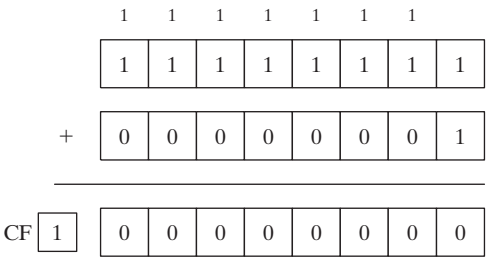
```
mov  ecx,1
sub  ecx,1                          ; ECX = 0, ZF = 1
mov  eax,0FFFFFFFFh
inc  eax                            ; EAX = 0, ZF = 1
inc  eax                            ; EAX = 1, ZF = 0
dec  eax                            ; EAX = 0, ZF = 1
```

Addition and the Carry Flag The Carry flag's operation is easiest to explain if we consider addition and subtraction separately. When adding two unsigned integers, the Carry flag is a copy of the carry out of the most significant bit of the destination operand. Intuitively, we can say CF = 1 when the sum exceeds the storage size of its destination operand. In the next example, ADD sets the Carry flag because the sum (100h) is too large for AL:

```
mov  al,0FFh
add  al,1                           ; AL = 00, CF = 1
```

Figure 4-3 shows what happens at the bit level when 1 is added to 0FFh. The carry out of the highest bit position of AL is copied into the Carry flag.

FIGURE 4–3 Adding 1 to 0FFh sets the Carry flag.



On the other hand, if 1 is added to 00FFh in AX, the sum easily fits into 16 bits and the Carry flag is clear:

```
mov  ax,00FFh
add  ax,1                           ; AX = 0100h, CF = 0
```

But adding 1 to FFFFh in the AX register generates a Carry out of the high bit position of AX:

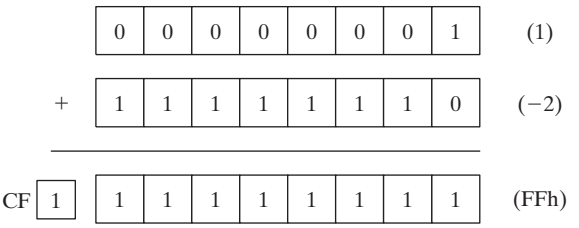
```
mov  ax,0FFFFh
add  ax,1                           ; AX = 0000, CF = 1
```

Subtraction and the Carry Flag A subtract operation sets the Carry flag when a larger unsigned integer is subtracted from a smaller one. Figure 4-4 shows what happens when we subtract 2 from 1, using 8-bit operands. Here is the corresponding assembly code:

```
mov  al,1
sub  al,2                           ; AL = FFh, CF = 1
```

> ***Tip:*** The INC and DEC instructions do not affect the Carry flag. Applying the NEG instruction to a nonzero operand always sets the Carry flag.

Figure 4–4 Subtracting 2 from 1 sets the Carry flag.



Auxiliary Carry The Auxiliary Carry (AC) flag indicates a carry or borrow out of bit 3 in the destination operand. It is primarily used in binary coded decimal (BCD) arithmetic, but can be used in other contexts. Suppose we add 1 to 0Fh. The sum (10h) contains a 1 in bit position 4 that was carried out of bit position 3:

```
mov  al,0Fh
add  al,1                       ; AC = 1
```

Here is the arithmetic:

```
   0 0 0 0 1 1 1 1
+  0 0 0 0 0 0 0 1
------------------
   0 0 0 1 0 0 0 0
```

Parity The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits. The following ADD and SUB instructions alter the parity of AL:

```
mov  al,10001100b
add  al,00000010b               ; AL = 10001110, PF = 1
sub  al,10000000b               ; AL = 00001110, PF = 0
```

After the ADD instruction executes, AL contains binary 10001110 (four 0 bits and four 1 bits), and PF = 1. After the SUB instruction executes, AL contains an odd number of 1 bits, so the Parity flag equals 0.

Signed Operations: Sign and Overflow Flags

Sign Flag The Sign flag is set when the result of a signed arithmetic operation is negative. The next example subtracts a larger integer (5) from a smaller one (4):

```
mov  eax,4
sub  eax,5                      ; EAX = -1, SF = 1
```

From a mechanical point of view, the Sign flag is a copy of the destination operand's high bit. The next example shows the hexadecimal values of BL when a negative result is generated:

```
mov  bl,1                       ; BL = 01h
sub  bl,2                       ; BL = FFh (-1), SF = 1
```

Overflow Flag The Overflow flag is set when the result of a signed arithmetic operation overflows or underflows the destination operand. For example, from Chapter 1 we know that the largest possible integer signed byte value is +127; adding 1 to it causes overflow:

```
mov  al,+127
add  al,1                          ; OF = 1
```

Similarly, the smallest possible negative integer byte value is −128. Subtracting 1 from it causes underflow. The destination operand value does not hold a valid arithmetic result, and the Overflow flag is set:

```
mov  al,-128
sub  al,1                          ; OF = 1
```

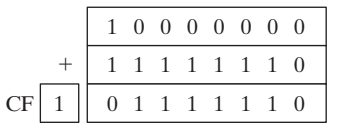
The Addition Test There is a very easy way to tell whether signed overflow has occurred when adding two operands. Overflow occurs when:

- Adding two positive operands generates a negative sum
- Adding two negative operands generates a positive sum

Overflow never occurs when the signs of two addition operands are different.

How the Hardware Detects Overflow The CPU uses an interesting mechanism to determine the state of the Overflow flag after an addition or subtraction operation. The value that carries out of the highest bit position is exclusive ORed with the carry into the high bit of the result. The resulting value is placed in the Overflow flag. In Figure 4-5, we show that adding the 8-bit binary integers 10000000 and 11111110 produces CF = 1, with carryIn(bit7) = 0. In other words, 1 XOR 0 produces OF = 1.

Figure 4–5 Demonstration of how the Overflow flag is set.



NEG Instruction The NEG instruction produces an invalid result if the destination operand cannot be stored correctly. For example, if we move −128 to AL and try to negate it, the correct value (+128) will not fit into AL. The Overflow flag is set, indicating that AL contains an invalid value:

```
mov al,-128                        ; AL = 10000000b
neg al                             ; AL = 10000000b, OF = 1
```

On the other hand, if +127 is negated, the result is valid and the Overflow flag is clear:

```
mov al,+127                        ; AL = 01111111b
neg al                             ; AL = 10000001b, OF = 0
```

How does the CPU know whether an arithmetic operation is signed or unsigned? We can only give what seems a dumb answer: It doesn't! The CPU sets all status flags after an arithmetic operation using a set of boolean rules, regardless of which flags are relevant. You (the programmer) decide which flags to interpret and which to ignore, based on your knowledge of the type of operation performed.

4.2.7 Example Program (*AddSubTest*)

The *AddSubTest* program shown below implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how certain status flags are affected:

```
; Addition and Subtraction     (AddSubTest.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword
.data
Rval   SDWORD ?
Xval   SDWORD 26
Yval   SDWORD 30
Zval   SDWORD 40

.code
main PROC
     ; INC and DEC
     mov   ax,1000h
     inc   ax                    ; 1001h
     dec   ax                    ; 1000h

     ; Expression: Rval = -Xval + (Yval - Zval)
     mov   eax,Xval
     neg   eax                   ; -26
     mov   ebx,Yval
     sub   ebx,Zval              ; -10
     add   eax,ebx
     mov   Rval,eax              ; -36

     ; Zero flag example:
     mov   cx,1
     sub   cx,1                  ; ZF = 1
     mov   ax,0FFFFh
     inc   ax                    ; ZF = 1

     ; Sign flag example:
     mov   cx,0
     sub   cx,1                  ; SF = 1
     mov   ax,7FFFh
     add   ax,2                  ; SF = 1

     ; Carry flag example:
     mov   al,0FFh
     add   al,1                  ; CF = 1,  AL = 00

     ; Overflow flag example:
     mov   al,+127
     add   al,1                  ; OF = 1
     mov   al,-128
     sub   al,1                  ; OF = 1

     INVOKE ExitProcess,0
main ENDP
END main
```

4.3 Data-Related Operators and Directives

Operators and directives are not executable instructions; instead, they are interpreted by the assembler. You can use a number of assembly language directives to get information about the addresses and size characteristics of data:

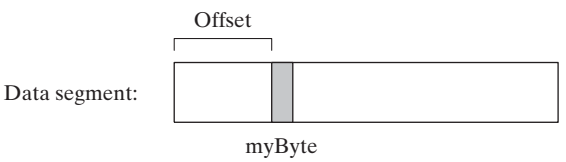
- The OFFSET operator returns the distance of a variable from the beginning of its enclosing segment.
- The PTR operator lets you override an operand's default size.
- The TYPE operator returns the size (in bytes) of an operand or of each element in an array.
- The LENGTHOF operator returns the number of elements in an array.
- The SIZEOF operator returns the number of bytes used by an array initializer.

In addition, the LABEL directive provides a way to redefine the same variable with different size attributes. The operators and directives in this chapter represent only a small subset of the operators supported by MASM. You may want to view the complete list in Appendix D.

4.3.1 OFFSET Operator

The OFFSET operator returns the offset of a data label. The offset represents the distance, in bytes, of the label from the beginning of the data segment. To illustrate, Figure 4-6 shows a variable named **myByte** inside the data segment.

Figure 4–6 A variable named myByte.



OFFSET Examples

In the next example, we declare three different types of variables:

```
.data
bVal  BYTE  ?
wVal  WORD  ?
dVal  DWORD ?
dVal2 DWORD ?
```

If **bVal** were located at offset 00404000 (hexadecimal), the OFFSET operator would return the following values:

```
mov esi,OFFSET bVal                 ; ESI = 00404000h
mov esi,OFFSET wVal                 ; ESI = 00404001h
mov esi,OFFSET dVal                 ; ESI = 00404003h
mov esi,OFFSET dVal2                ; ESI = 00404007h
```

OFFSET can also be applied to a direct-offset operand. Suppose **myArray** contains five 16-bit words. The following MOV instruction obtains the offset of **myArray**, adds 4, and moves the resulting address to ESI. We can say that ESI points to the third integer in the array:

```
.data
myArray WORD 1,2,3,4,5
.code
mov  esi,OFFSET myArray + 4
```

You can initialize a doubleword variable with the offset of another variable, effectively creating a pointer. In the following example, **pArray** points to the beginning of **bigArray**:

```
.data
bigArray DWORD 500 DUP(?)
pArray DWORD bigArray
```

The following statement loads the pointer's value into ESI, so the register can point to the beginning of the array:

```
mov esi,pArray
```

4.3.2 ALIGN Directive

The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary. The syntax is

ALIGN *bound*

Bound can be 1, 2, 4, 8, or 16. A value of 1 aligns the next variable on a 1-byte boundary (the default). If *bound* is 2, the next variable is aligned on an even-numbered address. If bound is 4, the next address is a multiple of 4. If *bound* is 16, the next address is a multiple of 16, a paragraph boundary. The assembler can insert one or more empty bytes before the variable to fix the alignment. Why bother aligning data? Because the CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses.

In the following example, **bVal** is arbitrarily located at offset 00404000. Inserting the ALIGN 2 directive before **wVal** causes it to be assigned an even-numbered offset:

```
bVal  BYTE  ?                ; 00404000h
ALIGN 2
wVal  WORD  ?                ; 00404002h
bVal2 BYTE  ?                ; 00404004h
ALIGN 4
dVal  DWORD ?                ; 00404008h
dVal2 DWORD ?                ; 0040400Ch
```

Note that **dVal** would have been at offset 00404005, but the ALIGN 4 directive bumped it up to offset 00404008.

4.3.3 PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you're trying to access the operand using a size attribute that is different from the one assumed by the assembler.

Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the operand sizes do not match:

```
.data
myDouble  DWORD  12345678h
.code
mov  ax,myDouble                ; error
```

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

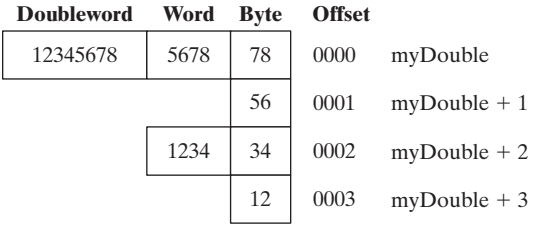
```
mov  ax,WORD PTR myDouble
```

Why wasn't 1234h moved into AX? x86 processors use the *little endian* storage format (Section 3.4.9), in which the low-order byte is stored at the variable's starting address. In Figure 4-7, the memory layout of **myDouble** is shown three ways: first as a doubleword, then as two words (5678h, 1234h), and finally as four bytes (78h, 56h, 34h, 12h).

We can access memory in any of these three ways, independent of the way a variable was defined. For example, if **myDouble** begins at offset 0000, the 16-bit value stored at that address is 5678h. We could also retrieve 1234h, the word at location **myDouble+2**, using the following statement:

```
mov   ax,WORD PTR [myDouble+2]        ; 1234h
```

Figure 4–7 Memory layout of myDouble.



Similarly, we could use the BYTE PTR operator to move a single byte from **myDouble** to BL:

```
mov   bl,BYTE PTR myDouble            ; 78h
```

Note that PTR must be used in combination with one of the standard assembler data types, BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE.

Moving Smaller Values into Larger Destinations We might want to move two smaller values from memory to a larger destination operand. In the next example, the first word is copied to the lower half of EAX and the second word is copied to the upper half. The DWORD PTR operator makes this possible:

```
.data
wordList WORD 5678h,1234h
.code
mov  eax,DWORD PTR wordList            ; EAX = 12345678h
```

4.3.4 TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable. For example, the TYPE of a byte equals 1, the TYPE of a word equals 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8. Here are examples of each:

```
.data
var1 BYTE  ?
var2 WORD  ?
var3 DWORD ?
var4 QWORD ?
```

The following table shows the value of each TYPE expression.

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.5 LENGTHOF Operator

The LENGTHOF operator counts the number of elements in an array, defined by the values appearing on the same line as its label. We will use the following data as an example:

```
.data
byte1    BYTE  10,
array1   WORD  30 DUP(
array2   WORD  5 DUP(3 D
array3   DWORD 1
digitStr BYTE  "12345678",0
```

When nested DUP operators are used in an array definition, LENGTHOF returns the product of the two counters. The following table lists the values returned by each LENGTHOF expression:

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array. Given the following data, LENGTHOF myArray would return the value 5:

```
myArray BYTE 10,20,30,40,50
        BYTE 60,70,80,90,100
```

Alternatively, you can end the first line with a comma and continue the list of initializers onto the next line. Given the following data, LENGTHOF myArray would return the value 10:

```
myArray BYTE 10,20,30,40,50,
             60,70,80,90,100
```

4.3.6 SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. In the following example, **intArray** has TYPE = 2 and LENGTHOF = 32. Therefore, SIZEOF **intArray** equals 64:

```
.data
intArray WORD 32 DUP(0)
.code
mov  eax,SIZEOF intArray        ; EAX = 64
```

4.3.7 LABEL Directive

The LABEL directive lets you insert a label and give it a size attribute without allocating any storage. All standard size attributes can be used with LABEL, such as BYTE, WORD, DWORD, QWORD or TBYTE. A common use of LABEL is to provide an alternative name and size attribute for the variable declared next in the data segment. In the following example, we declare a label just before **val32** named **val16** and give it a WORD attribute:

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov  ax,val16                   ; AX = 5678h
mov  dx,[val16+2]               ; DX = 1234h
```

val16 is an alias for the same storage location as **val32**. The LABEL directive itself allocates no storage.

Sometimes we need to construct a larger integer from two smaller integers. In the next example, a 32-bit value is loaded into EAX from two 16-bit variables:

```
.data
LongValue LABEL DWORD
val1  WORD  5678h
val2  WORD  1234h
.code
mov eax,LongValue               ; EAX = 12345678h
```