7

Integer Arithmetic

This chapter introduces the fundamental binary shift and rotation techniques, playing to one of the great strengths of assembly language. In fact, bit manipulation is an intrinsic part of computer graphics, data encryption, and hardware manipulation. Instructions that do this are powerful tools, and are only partially implemented by high-level languages, and somewhat obscured by their need to be platform-independent. We show quite a few ways you can apply bit shifting, including optimized multiplication and division.

Arithmetic with arbitrary-length integers is not supported by all high-level languages. But assembly language instructions make it possible to add and subtract integers of virtually any size. You will also be exposed to specialized instructions that perform arithmetic on packed decimal integers and integer strings.

7.1 Shift and Rotate Instructions

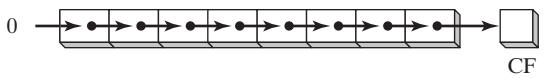
Along with bitwise instructions introduced in Chapter 6, shift instructions are among the most characteristic of assembly language. *Bit shifting* means to move bits right and left inside an operand. x86 processors provide a particularly rich set of instructions in this area (Table 7-1), all affecting the Overflow and Carry flags.

Table 7-1 Shift and Rotate Instructions.

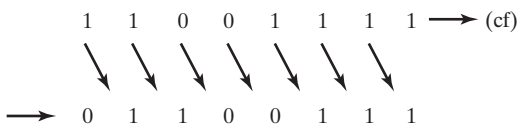
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

7.1.1 Logical Shifts and Arithmetic Shifts

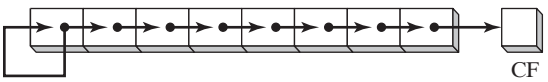
There are two ways to shift an operand's bits. The first, *logical shift*, fills the newly created bit position with zero. In the following illustration, a byte is logically shifted one position to the right. In other words, each bit is moved to the next lowest bit position. Note that bit 7 is assigned 0:



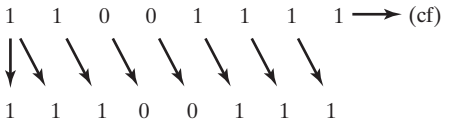
The following illustration shows a single logical right shift on the binary value 11001111, producing 01100111. The lowest bit is shifted into the Carry flag:



Another type of shift is called an *arithmetic shift*. The newly created bit position is filled with a copy of the original number's sign bit:

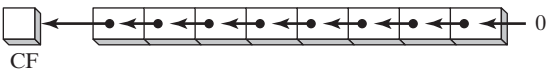


Binary 11001111, for example, has a 1 in the sign bit. When shifted arithmetically 1 bit to the right, it becomes 11100111:

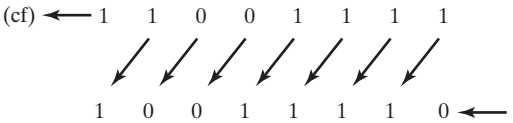


7.1.2 SHL Instruction

The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0. The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is discarded:



If you shift 11001111 left by 1 bit, it becomes 10011110:



The first operand in SHL is the destination and the second is the shift count:

```
SHL  destination,count
```

The following lists the types of operands permitted by this instruction:

```
SHL  reg,imm8
SHL  mem,imm8
SHL  reg,CL
SHL  mem,CL
```

x86 processors permit *imm8* to be any integer between 0 and 255. Alternatively, the CL register can contain a shift count. Formats shown here also apply to the SHR, SAL, SAR, ROR, ROL, RCR, and RCL instructions.

Example In the following instructions, BL is shifted once to the left. The highest bit is copied into the Carry flag and the lowest bit position is assigned zero:

```
mov  bl,8Fh                    ; BL = 10001111b
shl  bl,1                      ; CF = 1, BL = 00011110b
```

When a value is shifted leftward multiple times, the Carry flag contains the last bit to be shifted out of the most significant bit (MSB). In the following example, bit 7 does not end up in the Carry flag because it is replaced by bit 6 (a zero):

```
mov  al,10000000b
shl  al,2                          ; CF = 0, AL = 00000000b
```

Similarly, when a value is shifted rightward multiple times, the Carry flag contains the last bit to be shifted out of the least significant bit (LSB).

Bitwise Multiplication *Bitwise multiplication* is performed when you shift a number's bits in a leftward direction (toward the MSB). For example, SHL can perform multiplication by powers of 2. Shifting any operand left by n bits multiplies the operand by 2^n. For example, shifting the integer 5 left by 1 bit yields the product of $5 \times 2^1 = 10$:

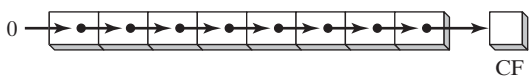
```
mov  dl,5        Before:  0 0 0 0 0 1 0 1  = 5
shl  dl,1        After:   0 0 0 0 1 0 1 0  = 10
```

If binary 00001010 (decimal 10) is shifted left by two bits, the result is the same as multiplying 10 by 2^2:

```
mov  dl,10                         ; before:  00001010
shl  dl,2                          ; after:   00101000
```

7.1.3 SHR Instruction

The SHR (shift right) instruction performs a logical right shift on the destination operand, replacing the highest bit with a 0. The lowest bit is copied into the Carry flag, and the bit that was previously in the Carry flag is lost:



SHR uses the same instruction formats as SHL. In the following example, the 0 from the lowest bit in AL is copied into the Carry flag, and the highest bit in AL is filled with a zero:

```
mov  al,0D0h                       ; AL = 11010000b
shr  al,1                          ; AL = 01101000b, CF = 0
```

In a multiple shift operation, the last bit to be shifted out of position 0 (the LSB) ends up in the Carry flag:

```
mov  al,00000010b
shr  al,2                          ; AL = 00000000b, CF = 1
```

Bitwise Division *Bitwise division* is accomplished when you shift a number's bits in a rightward direction (toward the LSB). Shifting an unsigned integer right by n bits divides the operand by 2^n. In the following statements, we divide 32 by 2^1, producing 16:

```
mov  dl,32       Before:  0 0 1 0 0 0 0 0  = 32
shr  dl,1        After:   0 0 0 1 0 0 0 0  = 16
```

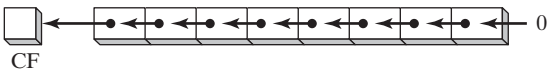
In the following example, 64 is divided by 2^3:

```
mov  al,01000000b                ; AL = 64
shr  al,3                        ; divide by 8, AL = 00001000b
```

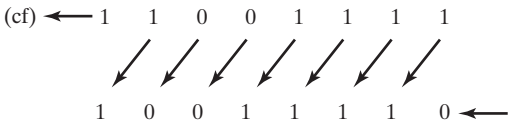
Division of signed numbers by shifting is accomplished using the SAR instruction because it preserves the number's sign bit.

7.1.4 SAL and SAR Instructions

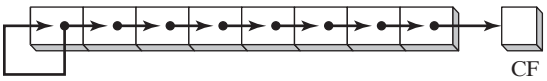
The SAL (shift arithmetic left) instruction works the same as the SHL instruction. For each shift count, SAL shifts each bit in the destination operand to the next highest bit position. The lowest bit is assigned 0. The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is discarded:



If you shift binary 11001111 to the left by one bit, it becomes 10011110:



The SAR (shift arithmetic right) instruction performs a right arithmetic shift on its destination operand:



The operands for SAL and SAR are identical to those for SHL and SHR. The shift may be repeated, based on the counter in the second operand:

```
SAR  destination,count
```

The following example shows how SAR duplicates the sign bit. AL is negative before and after it is shifted to the right:

```
mov  al,0F0h                     ; AL = 11110000b (-16)
sar  al,1                        ; AL = 11111000b (-8), CF = 0
```

Signed Division You can divide a signed operand by a power of 2, using the SAR instruction. In the following example, –128 is divided by 2^3. The quotient is –16:

```
mov  dl,-128                     ; DL = 10000000b
sar  dl,3                        ; DL = 11110000b
```

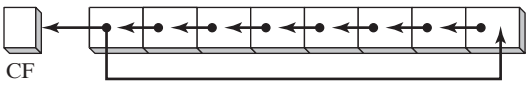
Sign-Extend AX into EAX Suppose AX contains a signed integer and you want to extend its sign into EAX. First shift EAX 16 bits to the left, then shift it arithmetically 16 bits to the right:

```
mov  ax,-128                     ; EAX = ????FF80h
shl  eax,16                      ; EAX = FF800000h
sar  eax,16                      ; EAX = FFFFFF80h
```

7.1.5 ROL Instruction

Bitwise rotation occurs when you move the bits in a circular fashion. In some versions, the bit leaving one end of the number is immediately copied into the other end. Another type of rotation uses the Carry flag as an intermediate point for shifted bits.

The ROL (rotate left) instruction shifts each bit to the left. The highest bit is copied into the Carry flag and the lowest bit position. The instruction format is the same as for SHL:



Bit rotation does not lose bits. A bit rotated off one end of a number appears again at the other end. Note in the following example how the high bit is copied into both the Carry flag and bit position 0:

```
mov  al,40h                    ; AL = 01000000b
rol  al,1                      ; AL = 10000000b, CF = 0
rol  al,1                      ; AL = 00000001b, CF = 1
rol  al,1                      ; AL = 00000010b, CF = 0
```

Multiple Rotations When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the MSB position:

```
mov  al,00100000b
rol  al,3                      ; CF = 1, AL = 00000001b
```

Exchanging Groups of Bits You can use ROL to exchange the upper (bits 4–7) and lower (bits 0–3) halves of a byte. For example, 26h rotated four bits in either direction becomes 62h:

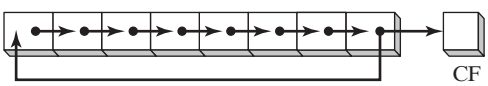
```
mov  al,26h
rol  al,4                      ; AL = 62h
```

When rotating a multibyte integer by four bits, the effect is to rotate each hexadecimal digit one position to the right or left. Here, for example, we repeatedly rotate 6A4Bh left four bits, eventually ending up with the original value:

```
mov  ax,6A4Bh
rol  ax,4                      ; AX = A4B6h
rol  ax,4                      ; AX = 4B6Ah
rol  ax,4                      ; AX = B6A4h
rol  ax,4                      ; AX = 6A4Bh
```

7.1.6 ROR Instruction

The ROR (rotate right) instruction shifts each bit to the right and copies the lowest bit into the Carry flag and the highest bit position. The instruction format is the same as for SHL:



In the following examples, note how the lowest bit is copied into both the Carry flag and the highest bit position of the result:

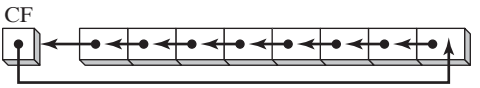
```
mov  al,01h                    ; AL = 00000001b
ror  al,1                      ; AL = 10000000b, CF = 1
ror  al,1                      ; AL = 01000000b, CF = 0
```

Multiple Rotations When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the LSB position:

```
mov  al,00000100b
ror  al,3                      ; AL = 10000000b, CF = 1
```

7.1.7 RCL and RCR Instructions

The RCL (rotate carry left) instruction shifts each bit to the left, copies the Carry flag to the LSB, and copies the MSB into the Carry flag:



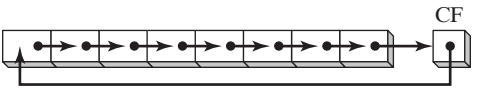
If we imagine the Carry flag as an extra bit added to the high end of the operand, RCL looks like a rotate left operation. In the following example, the CLC instruction clears the Carry flag. The first RCL instruction moves the high bit of BL into the Carry flag and shifts the other bits left. The second RCL instruction moves the Carry flag into the lowest bit position and shifts the other bits left:

```
clc                            ; CF = 0
mov  bl,88h                    ; CF,BL = 0 10001000b
rcl  bl,1                      ; CF,BL = 1 00010000b
rcl  bl,1                      ; CF,BL = 0 00100001b
```

Recover a Bit from the Carry Flag RCL can recover a bit that was previously shifted into the Carry flag. The following example checks the lowest bit of **testval** by shifting its lowest bit into the Carry flag. If the lowest bit of testval is 1, a jump is taken; if the lowest bit is 0, RCL restores the number to its original value:

```
.data
testval BYTE  01101010b
.code
shr  testval,1                 ; shift LSB into Carry flag
jc   exit                      ; exit if Carry flag set
rcl  testval,1                 ; else restore the number
```

RCR Instruction The RCR (rotate carry right) instruction shifts each bit to the right, copies the Carry flag into the MSB, and copies the LSB into the Carry flag:



As in the case of RCL, it helps to visualize the integer in this figure as a 9-bit value, with the Carry flag to the right of the LSB.

The following code example uses STC to set the Carry flag; then, it performs a rotate carry right operation on the AH register:

```
stc                             ; CF = 1
mov  ah,10h                     ; AH, CF = 00010000 1
rcr  ah,1                       ; AH, CF = 10001000 0
```

7.1.8 Signed Overflow

The Overflow flag is set if the act of shifting or rotating a signed integer by one bit position generates a value outside the signed integer range of the destination operand. To put it another way, the number's sign is reversed. In the following example, a positive integer (+127) stored in an 8-bit register becomes negative (−2) when rotated left:

```
mov  al,+127                    ; AL = 01111111b
rol  al,1                       ; OF = 1, AL = 11111110b
```

Similarly, when –128 is shifted one position to the right, the Overflow flag is set. The result in AL (+64) has the opposite sign:

```
mov  al,-128                    ; AL = 10000000b
shr  al,1                       ; OF = 1, AL = 01000000b
```

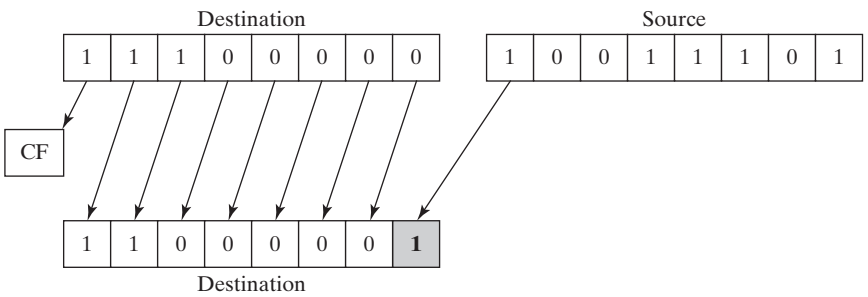
The value of the Overflow flag is undefined when the shift or rotation count is greater than 1.

7.1.9 SHLD/SHRD Instructions

The SHLD (shift left double) instruction shifts a destination operand a given number of bits to the left. The bit positions opened up by the shift are filled by the most significant bits of the source operand. The source operand is not affected, but the Sign, Zero, Auxiliary, Parity, and Carry flags are affected:

```
SHLD  dest, source, count
```

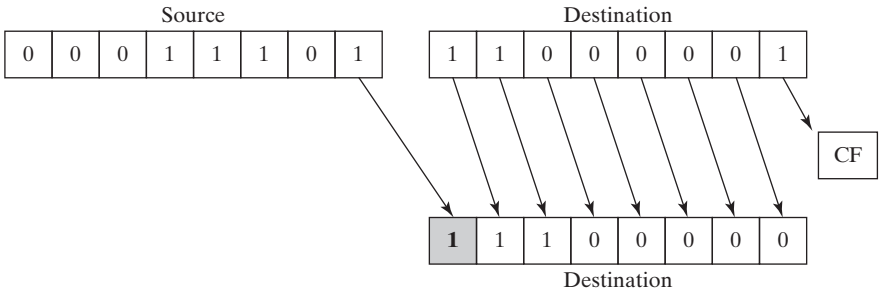
The following illustration shows the execution of SHLD with a shift count of 1. The highest bit of the source operand is copied into the lowest bit of the destination operand. All the destination operand bits are shifted left:



The SHRD (shift right double) instruction shifts a destination operand a given number of bits to the right. The bit positions opened up by the shift are filled by the least significant bits of the source operand:

```
SHRD   dest, source, count
```

The following illustration shows the execution of SHRD with a shift count of 1:



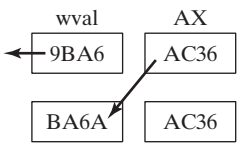
The following instruction formats apply to both SHLD and SHRD. The destination operand can be a register or memory operand, and the source operand must be a register. The count operand can be the CL register or an 8-bit immediate operand:

```
SHLD  reg16,reg16,CL/imm8
SHLD  mem16,reg16,CL/imm8
SHLD  reg32,reg32,CL/imm8
SHLD  mem32,reg32,CL/imm8
```

Example 1 The following statements shift **wval** to the left 4 bits and insert the high 4 bits of AX into the low 4 bit positions of **wval:**

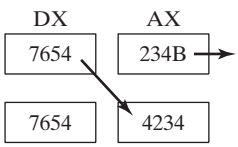
```
.data
wval WORD 9BA6h
.code
mov   ax,0AC36h
shld  wval,ax,4                ; wval = BA6Ah
```

The data movement is shown in the following figure:



Example 2 In the following example, AX is shifted to the right 4 bits, and the low 4 bits of DX are shifted into the high 4 positions of AX:

```
mov   ax,234Bh
mov   dx,7654h
shrd  ax,dx,4
```



SHLD and SHRD can be used to manipulate bit-mapped images, when groups of bits must be shifted left and right to reposition images on the screen. Another potential application is data encryption, in which the encryption algorithm involves the shifting of bits. Finally, the two instructions can be used when performing fast multiplication and division with very long integers.

The following code example demonstrates SHRD by shifting an array of doublewords to the right by 4 bits:

```
.data
array DWORD 648B2165h,8C943A29h,6DFA4B86h,91F76C04h,8BAF9857h

.code
    mov   bl,4                        ; shift count
    mov   esi,OFFSET array            ; offset of the array
    mov   ecx,(LENGTHOF array) – 1    ; number of array elements

L1: push  ecx                         ; save loop counter
    mov   eax,[esi + TYPE DWORD]
    mov   cl,bl                       ; shift count
    shrd  [esi],eax,cl                ; shift EAX into high bits of
                                        [ESI]
    add   esi,TYPE DWORD              ; point to next doubleword pair
    pop   ecx                         ; restore loop counter
    loop  L1

    shr DWORD PTR [esi],COUNT         ; shift the last doubleword
```

7.1.10 Section Review

1. Which instruction shifts each bit in an operand to the left and copies the highest bit into both the Carry flag and the lowest bit position?
2. Which instruction shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position?
3. Which instruction performs the following operation (CF = Carry flag)?

```
Before:  CF,AL = 1 11010101
After:   CF,AL = 1 10101011
```

4. What happens to the Carry flag when the SHR AX,1 instruction is executed?
5. *Challenge:* Write a series of instructions that shift the lowest bit of AX into the highest bit of BX without using the SHRD instruction. Next, perform the same operation using SHRD.
6. *Challenge:* One way to calculate the parity of a 32-bit number in EAX is to use a loop that shifts each bit into the Carry flag and accumulates a count of the number of times the Carry flag was set. Write a code that does this, and set the Parity flag accordingly.

7.2 Shift and Rotate Applications

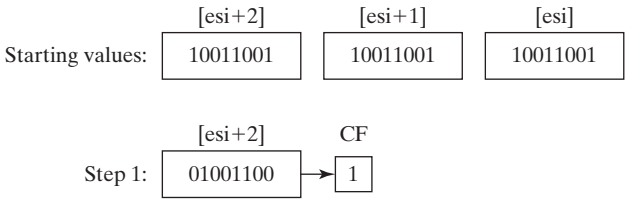
When a program needs to move bits from one part of an integer to another, assembly language is a great tool for the job. Sometimes, we move a subset of a number's bits to position 0 to make it easier to isolate the value of the bits. In this section, we show a few common bit shift and rotate applications that are easy to implement. More applications will be found in the chapter exercises.

7.2.1 Shifting Multiple Doublewords

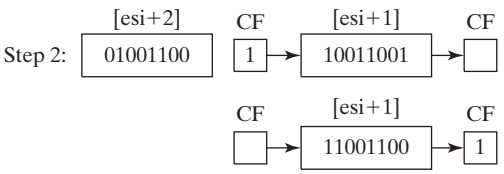
You can shift an extended-precision integer that has been divided into an array of bytes, words, or doublewords. Before doing this, you must know how the array elements are stored. A common way to store the integer is called *little-endian order*. It works like this: Place the low-order byte at the array's starting address. Then, working your way up from that byte to the high-order byte, store each in the next sequential memory location. Instead of storing the array as a series of bytes, you could store it as a series of words or doublewords. If you did so, the individual bytes would still be in little-endian order, because x86 machines store words and doublewords in little-endian order.

The following steps show how to shift an array of bytes 1 bit to the right:

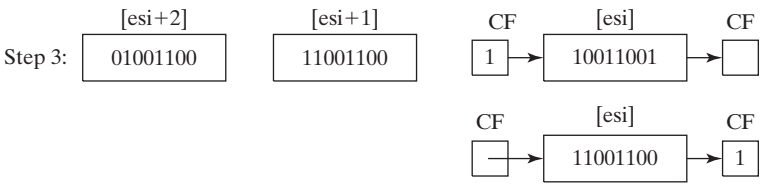
Step 1: Shift the highest byte at [ESI+2] to the right, automatically copying its lowest bit into the Carry flag.



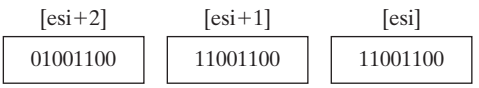
Step 2: Rotate the value at [ESI+1] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:



Step 3: Rotate the value at [ESI] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:



After Step 3 is complete, all bits have been shifted 1 position to the right:



The following code excerpt from the *Multishift.asm* program implements the steps we just outlined:

```
.data
ArraySize = 3
array BYTE ArraySize DUP(99h)   ; 1001 pattern in each nybble
.code
main PROC
     mov esi,0
     shr array[esi+2],1         ; high byte
     rcr array[esi+1],1         ; middle byte, include Carry flag
     rcr array[esi],1           ; low byte, include Carry flag
```

Although our current example only shifts 3 bytes, the example could easily be modified to shift an array of words or doublewords. Using a loop, you could shift an array of arbitrary size.

7.2.2 Binary Multiplication

Sometimes programmers squeeze every performance advantage they can into integer multiplication by using bit shifting rather than the MUL instruction. The SHL instruction performs unsigned multiplication when the multiplier is a power of 2. Shifting an unsigned integer n bits to the left multiplies it by 2^n. Any other multiplier can be expressed as the sum of powers of 2. For example, to multiply unsigned EAX by 36, we can write 36 as $2^5 + 2^2$ and use the distributive property of multiplication:

$$\begin{aligned} EAX * 36 &= EAX * (2^5 + 2^2) \\ &= EAX * (32 + 4) \\ &= (EAX * 32) + (EAX * 4) \end{aligned}$$

The following figure shows the multiplication 123 * 36, producing 4428, the product:

```
          01111011      123
    ×     00100100      36
    --------------
          01111011      123 SHL 2
+   01111011            123 SHL 5
--------------------
0001000101001100        4428
```

It is interesting to note that bits 2 and 5 are set in the multiplier (36), and the integers 2 and 5 are also the required shift counters. Using this information, the following code snippet multiplies 123 by 36, using SHL and ADD instructions:

```
mov  eax,123
mov  ebx,eax
shl  eax,5                      ; multiply by 2^5
shl  ebx,2                      ; multiply by 2^2
add  eax,ebx                    ; add the products
```

As a chapter programming exercise, you will be asked to generalize this example and create a procedure that multiplies any two 32-bit unsigned integers using shifting and addition.

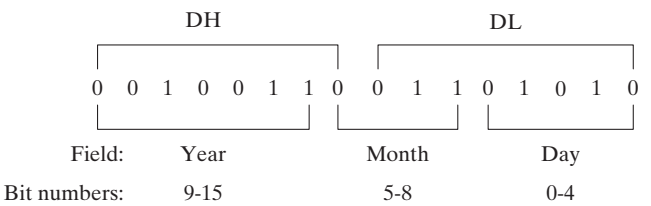
7.2.3 Displaying Binary Bits

A common programming task is converting a binary integer to an ASCII binary string, allowing the latter to be displayed. The SHL instruction is useful in this regard because it copies the highest bit of an operand into the Carry flag each time the operand is shifted left. The following **BinToAsc** procedure is a simple implementation:

```
;-----------------------------------------------------
BinToAsc PROC
;
; Converts 32-bit binary integer to ASCII binary.
; Receives: EAX = binary integer, ESI points to buffer
; Returns: buffer filled with ASCII binary digits
;-----------------------------------------------------
    push  ecx
    push  esi

    mov   ecx,32              ; number of bits in EAX

L1: shl   eax,1               ; shift high bit into Carry flag
    mov   BYTE PTR [esi],'0'  ; choose 0 as default digit
    jnc   L2                  ; if no Carry, jump to L2
    mov   BYTE PTR [esi],'1'  ; else move 1 to buffer

L2: inc   esi                 ; next buffer position
    loop  L1                  ; shift another bit to left

    pop   esi
    pop   ecx
    ret
BinToAsc ENDP
```

7.2.4 Extracting File Date Fields

When storage space is at a premium, system-level software often packs multiple data fields into a single integer. To uncover this data, applications often need to extract sequences of bits called *bit strings*. For example, in real-address mode, MS-DOS function 57h returns the date stamp of a file in DX. (The date stamp shows the date on which the file was last modified.) Bits 0 through 4 represent a day number between 1 and 31, bits 5 through 8 are the month number, and bits 9 through 15 hold the year number. If a file was last modified on March 10, 1999, the file's date stamp would appear as follows in the DX register (the year number is relative to 1980):



To extract a single bit string, shift its bits into the lowest part of a register and clear the irrelevant bit positions. The following code example extracts the day number field of a date stamp integer by making a copy of DL and masking off bits not belonging to the field:

```
mov  al,dl                    ; make a copy of DL
and  al,00011111b             ; clear bits 5-7
mov  day,al                   ; save in day
```

To extract the month number field, we shift bits 5 through 8 into the low part of AL before masking off all other bits. AL is then copied into a variable:

```
mov  ax,dx                    ; make a copy of DX
shr  ax,5                     ; shift right 5 bits
and  al,00001111b             ; clear bits 4-7
mov  month,al                 ; save in month
```

The year number (bits 9 through 15) field is completely within the DH register. We copy it to AL and shift right by 1 bit:

```
mov  al,dh                    ; make a copy of DH
shr  al,1                     ; shift right one position
mov  ah,0                     ; clear AH to zeros
add  ax,1980                  ; year is relative to 1980
mov  year,ax                  ; save in year
```

7.2.5 Section Review

1. Write assembly language instructions that calculate EAX * 24 using binary multiplication.
2. Write assembly language instructions that calculate EAX * 21 using binary multiplication. *Hint:* $21 = 2^4 + 2^2 + 2^0$.
3. What change would you make to the BinToAsc procedure in Section 7.2.3 in order to display the binary bits in reverse order?
4. The time stamp field of a file directory entry uses bits 0 through 4 for the seconds, bits 5 through 10 for the minutes, and bits 11 through 15 for the hours. Write instructions that extract the minutes and copy the value to a byte variable named **bMinutes**.

7.3 Multiplication and Division Instructions

In 32-bit mode, integer multiplication can be performed as a 32-bit, 16-bit, or 8-bit operation. In 64-bit mode, you can also use 64-bit operands. The MUL and IMUL instructions perform unsigned and signed integer multiplication, respectively. The DIV instruction performs unsigned integer division, and IDIV performs signed integer division.

7.3.1 MUL Instruction

In 32-bit mode, the MUL (unsigned multiply) instruction comes in three versions: The first version multiplies an 8-bit operand by the AL register. The second version multiplies a 16-bit operand by the AX register, and the third version multiplies a 32-bit operand by the EAX register. The multiplier and multiplicand must always be the same size, and the product is twice their size. The three formats accept register and memory operands, but not immediate operands:

```
MUL  reg/mem8
MUL  reg/mem16
MUL  reg/mem32
```

The single operand in the MUL instruction is the multiplier. Table 7-2 shows the default multiplicand and product, depending on the size of the multiplier. Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur. MUL sets the Carry and Overflow flags if the upper half of the product is not equal to zero. The Carry flag is ordinarily used for unsigned arithmetic, so we'll focus on it here. When AX is multiplied by a 16-bit operand, for example, the product is stored in the combined DX and AX registers. That is, the high 16 bits of the product are stored in DX, and the low 16 bits are stored in AX. The Carry flag is set if DX is not equal to zero, which lets us know that the product will not fit into the lower half of the implied destination operand.

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

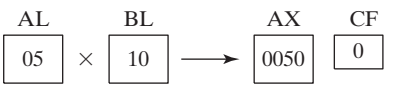
A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.

MUL Examples

The following statements multiply AL by BL, storing the product in AX. The Carry flag is clear (CF = 0) because AH (the upper half of the product) equals zero:

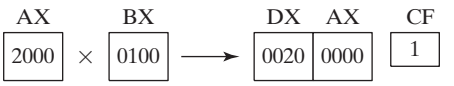
```
mov  al,5h
mov  bl,10h
mul  bl                          ; AX = 0050h, CF = 0
```

The following diagram illustrates the movement between registers:



The following statements multiply the 16-bit value 2000h by 0100h. The Carry flag is set because the upper part of the product (located in DX) is not equal to zero:

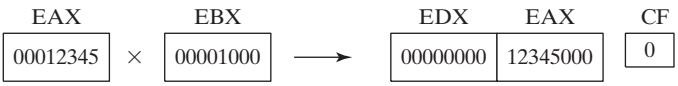
```
.data
val1  WORD  2000h
val2  WORD  0100h
.code
mov  ax,val1                     ; AX = 2000h
mul  val2                        ; DX:AX = 00200000h, CF = 1
```



The following statements multiply 12345h by 1000h, producing a 64-bit product in the combined EDX and EAX registers. The Carry flag is clear because the upper half of the product in EDX equals zero:

```
mov  eax,12345h
mov  ebx,1000h
mul  ebx                          ; EDX:EAX = 0000000012345000h, CF = 0
```

The following diagram illustrates the movement between registers:



Using MUL in 64-Bit Mode

In 64-bit mode, you can use 64-bit operands with the MUL instruction. A 64-bit register or memory operand is multiplied against RDX, producing a 128-bit product in RDX:RAX. In the following example, each bit in RAX is shifted one position to the left when RAX is multiplied by 2. The highest bit of RAX spills over into the RDX register, which equals 0000000000000001 hexadecimal:

```
mov rax,0FFFF0000FFFF0000h
mov rbx,2
mul rbx                ; RDX:RAX = 0000000000000001FFFE0001FFFE0000
```

In the next example, we multiply RAX by a 64-bit memory operand. The value is being multiplied by 16, so each hexadecimal digit is shifted one position to the left (a 4-bit shift is the same as multiplying by 16).

```
.data
multiplier QWORD 10h
.code
mov rax,0AABBBBCCCCDDDDh
mul multiplier       ; RDX:RAX = 00000000000000000AABBBBCCCCDDDD0h
```

7.3.2 IMUL Instruction

The IMUL (signed multiply) instruction performs signed integer multiplication. Unlike the MUL instruction, IMUL preserves the sign of the product. It does this by sign extending the highest bit of the lower half of the product into the upper bits of the product. The x86 instruction set supports three formats for the IMUL instruction: one operand, two operands, and three operands. In the one-operand format, the multiplier and multiplicand are the same size and the product is twice their size.

Single-Operand Formats The one-operand formats store the product in AX, DX:AX, or EDX:EAX:

```
IMUL  reg/mem8                ; AX = AL * reg/mem8
IMUL  reg/mem16               ; DX:AX = AX * reg/mem16
IMUL  reg/mem32               ; EDX:EAX = EAX * reg/mem32
```

As in the case of MUL, the storage size of the product makes overflow impossible. Also, the Carry and Overflow flags are set if the upper half of the product is not a sign extension of the lower half. You can use this information to decide whether to ignore the upper half of the product.

Two-Operand Formats (32-Bit Mode) The two-operand version of the IMUL instruction in 32-bit mode stores the product in the first operand, which must be a register. The second operand (the multiplier) can be a register, a memory operand, or an immediate value. Following are the 16-bit formats:

```
IMUL  reg16,reg/mem16
IMUL  reg16,imm8
IMUL  reg16,imm16
```

Following are the 32-bit operand types showing that the multiplier can be a 32-bit register, a 32-bit memory operand, or an immediate value (8 or 32 bits):

```
IMUL  reg32,reg/mem32
IMUL  reg32,imm8
IMUL  reg32,imm32
```

The two-operand formats truncate the product to the length of the destination. If significant digits are lost, the Overflow and Carry flags are set. Be sure to check one of these flags after performing an IMUL operation with two operands.

Three-Operand Formats The three-operand formats in 32-bit mode store the product in the first operand. The second operand can be a 16-bit register or memory operand, which is multiplied by the third operand, an 8- or 16-bit immediate value:

```
IMUL  reg16,reg/mem16,imm8
IMUL  reg16,reg/mem16,imm16
```

A 32-bit register or memory operand can be multiplied by an 8- or 32-bit immediate value:

```
IMUL  reg32,reg/mem32,imm8
IMUL  reg32,reg/mem32,imm32
```

If significant digits are lost when IMUL executes, the Overflow and Carry flags are set. Be sure to check one of these flags after performing an IMUL operation with three operands.

Using IMUL in 64-Bit Mode

In 64-bit mode, you can use 64-bit operands with the MUL instruction. In the two-operand format, a 64-bit register or memory operand is multiplied against RDX, producing a 128-bit sign-extended product in RDX:RAX. In the next example, RBX is multiplied by RAX, producing a 128-bit product of −16.

```
mov  rax,-4
mov  rbx,4
imul rb             ; RDX = 0FFFFFFFFFFFFFFFFh, RAX = -16
```

In other words, decimal −16 is represented as FFFFFFFFFFF0 hexadecimal in RAX, and RDX just contains an extension of RAX's high-order bit, also known as its sign bit.

The three-operand format is also available in 64-bit mode. In the next example, we multiply the multiplicand (−16) by 4, producing −64 in the RAX register:

```
.data
multiplicand QWORD -16
.code
imul rax, multiplicand, 4           ; RAX = FFFFFFFFFFFFFFC0 (-64)
```

Unsigned Multiplication The two-operand and three-operand IMUL formats may also be used for unsigned multiplication because the lower half of the product is the same for signed and unsigned numbers. There is a small disadvantage to doing so: The Carry and Overflow flags will not indicate whether the upper half of the product equals zero.

IMUL Examples

The following instructions multiply 48 by 4, producing +192 in AX. Although the product is correct, AH is not a sign extension of AL, so the Overflow flag is set:

```
mov   al,48
mov   bl,4
imul  bl                        ; AX = 00C0h, OF = 1
```

The following instructions multiply −4 by 4, producing −16 in AX. AH is a sign extension of AL, so the Overflow flag is clear:

```
mov   al,-4
mov   bl,4
imul  bl                        ; AX = FFF0h, OF = 0
```

The following instructions multiply 48 by 4, producing +192 in DX:AX. DX is a sign extension of AX, so the Overflow flag is clear:

```
mov   ax,48
mov   bx,4
imul  bx                        ; DX:AX = 000000C0h, OF = 0
```

The following instructions perform 32-bit signed multiplication (4,823,424 * −423), producing −2,040,308,352 in EDX:EAX. The Overflow flag is clear because EDX is a sign extension of EAX:

```
mov   eax,+4823424
mov   ebx,-423
imul  ebx                       ; EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

The following instructions demonstrate two-operand formats:

```
.data
word1   SWORD 4
dword1  SDWORD 4
.code
mov   ax,-16                      ; AX = -16
mov   bx,2                        ; BX = 2
imul  bx,ax                       ; BX = -32
imul  bx,2                        ; BX = -64
```

```
imul  bx,word1                  ; BX = -256
mov   eax,-16                   ; EAX = -16
mov   ebx,2                     ; EBX = 2
imul  ebx,eax                   ; EBX = -32
imul  ebx,2                     ; EBX = -64
imul  ebx,dword1                ; EBX = -256
```

The two-operand and three-operand IMUL instructions use a destination operand that is the same size as the multiplier. Therefore, it is possible for signed overflow to occur. Always check the Overflow flag after executing these types of IMUL instructions. The following two-operand instructions demonstrate signed overflow because −64,000 cannot fit within the 16-bit destination operand:

```
mov   ax,-32000
imul  ax,2                      ; OF = 1
```

The following instructions demonstrate three-operand formats, including an example of signed overflow:

```
.data
word1  SWORD 4
dword1 SDWORD 4
.code
imul  bx,word1,-16              ; BX = word1 * -16
imul  ebx,dword1,-16            ; EBX = dword1 * -16
imul  ebx,dword1,-2000000000    ; signed overflow!
```

7.3.3 Measuring Program Execution Times

Programmers often find it useful to compare the performance of one code implementation to another by measuring their performance times. The Microsoft Windows API provides the necessary tools to do this, which we have made even more accessible with the GetMseconds procedure in the Irvine32 library. The procedure gets the number of system milliseconds that have elapsed since midnight. In the following code example, GetMseconds is called first, so we can record the system starting time. Then we call the procedure whose execution time we wish to measure (*FirstProcedureToTest*). Finally, GetMseconds is called a second time, and the difference between the current milliseconds value and the starting time is calculated:

```
.data
startTime DWORD ?
procTime1 DWORD ?
procTime2 DWORD ?
.code
call GetMseconds                ; get start time
mov  startTime,eax
.
call FirstProcedureToTest
.
call GetMseconds                ; get stop time
sub  eax,startTime              ; calculate the elapsed time
mov  procTime1,eax              ; save the elapsed time
```

There is, of course, a small amount of execution time used up by calling GetMseconds twice. But this overhead is insignificant when we measure the ratio of performance times between one code implementation and another. Here, we call the other procedure we wish to test, and save its execution time (*procTime2*):

```
call GetMseconds                ; get start time
mov  startTime,eax
.
call SecondProcedureToTest
.
call GetMseconds                ; get stop time
sub  eax,startTime              ; calculate the elapsed time
mov  procTime2,eax              ; save the elapsed time
```

Now, the ratio of *procTime1* to *procTime2* indicates the relative performance of the two procedures.

Comparing MUL and IMUL to Bit Shifting

In older x86 processors, there was a significant difference in performance between multiplication by bit shifting versus multiplication using the MUL and IMUL instructions. We can use the GetMseconds procedure to compare the execution time of the two types of multiplication. The following two procedures perform multiplication repeatedly using a LOOP_COUNT constant to determine the amount of repetition:

```
mult_by_shifting PROC
;
; Multiplies EAX by 36 using SHL, LOOP_COUNT times.
;
     mov  ecx,LOOP_COUNT
L1:  push eax                   ; save original EAX
     mov  ebx,eax
     shl  eax,5
     shl  ebx,2
     add  eax,ebx
     pop  eax                   ; restore EAX
     loop L1
     ret
mult_by_shifting ENDP

mult_by_MUL PROC
;
; Multiplies EAX by 36 using MUL, LOOP_COUNT times.
;
     mov  ecx,LOOP_COUNT
L1:  push eax                   ; save original EAX
     mov  ebx,36
     mul  ebx
     pop  eax                   ; restore EAX
     loop L1
     ret
mult_by_MUL ENDP
```

The following code calls *mult_by_shifting* and displays the timing results. See the *CompareMult.asm* program from the book's Chapter 7 examples for the complete implementation:

```
.data
LOOP_COUNT = 0FFFFFFFFh
.data
intval DWORD 5
startTime DWORD ?
.code
call  GetMseconds                    ; get start time
mov   startTime,eax
mov   eax,intval                     ; multiply now
call  mult_by_shifting
call  GetMseconds                    ; get stop time
sub   eax,startTime
call  WriteDec                       ; display elapsed time
```

After calling *mult_by_MUL* in the same manner, the resulting timings on a legacy 4-GHz Pentium 4 showed that the SHL approach executed in 6.078 seconds and the MUL approach executed in 20.718 seconds. In other words, using MUL instruction was 241 percent slower. However, when running the same program on a more recent processor, the timings of both function calls were exactly the same. This example shows that Intel has managed to greatly optimize the MUL and IMUL instructions in recent processors.

7.3.4 DIV Instruction

In 32-bit mode, the DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit unsigned integer division. The single register or memory operand is the divisor. The formats are

```
DIV  reg/mem8
DIV  reg/mem16
DIV  reg/mem32
```

The following table shows the relationship between the dividend, divisor, quotient, and remainder:

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

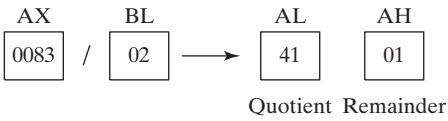
In 64-bit mode, the DIV instruction uses RDX:RAX as the dividend, and it permits the divisor to be a 64-bit register or memory operand. The quotient is stored in RAX, and the remainder in RDX.

DIV Examples

The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:

```
mov  ax,0083h                ; dividend
mov  bl,2                    ; divisor
div  bl                      ; AL = 41h,  AH = 01h
```

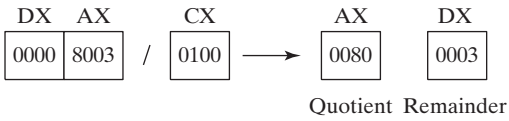
The following diagram illustrates the movement between registers:



The following instructions perform 16-bit unsigned division (8003h/100h), producing a quotient of 80h and a remainder of 3. DX contains the high part of the dividend, so it must be cleared before the DIV instruction executes:

```
mov  dx,0                       ; clear dividend, high
mov  ax,8003h                   ; dividend, low
mov  cx,100h                    ; divisor
div  cx                         ; AX = 0080h,  DX = 0003h
```

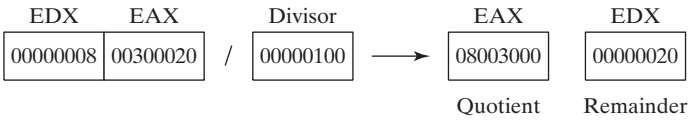
The following diagram illustrates the movement between registers:



The following instructions perform 32-bit unsigned division using a memory operand as the divisor:

```
.data
dividend QWORD 0000000800300020h
divisor  DWORD 00000100h
.code
mov  edx,DWORD PTR dividend + 4 ; high doubleword
mov  eax,DWORD PTR dividend     ; low doubleword
div  divisor                    ; EAX = 08003000h, EDX = 00000020h
```

The following diagram illustrates the movement between registers:



The following 64-bit division produces the quotient (0108000000003330h) in RAX and the remainder (0000000000000020h) in RDX:

```
.data
dividend_hi  QWORD 0000000000000108h
dividend_lo  QWORD 0000000033300020h
divisor      QWORD 0000000000010000h
.code
mov  rdx, dividend_hi
mov  rax, dividend_lo
div  divisor                    ; RAX = 0108000000003330
                                ; RDX = 0000000000000020
```

Notice how each hexadecimal digit in the dividend was shifted 4 positions to the right, because it was divided by 64. (Division by 16 would have moved each digit only one position to the right.)

7.3.5 Signed Integer Division

Signed integer division is nearly identical to unsigned division, with one important difference: The dividend must be sign-extended before the division takes place. *Sign extension* is the term used for copying the highest bit of a number into all of the upper bits of its enclosing variable or register. To show why this is necessary, let's try leaving it out. The following code uses MOV to assign −101 to AX, which is the lower half of EAX:

```
.data
wordVal SWORD -101     ; 009Bh
.code
mov  eax,0             ; EAX = 00000000h
mov  ax,wordVal        ; EAX = 0000009Bh  (+155)
mov  bx,2              ; EBX is the divisor
idiv bx                ; divide EAX by BX (signed operation)
```

Unfortunately, the 009Bh in EAX is not really equal to −101. It is equal to +155, so the quotient produced by the division is +77, which is not what we wanted. Instead, the correct way to set up the problem is to use the CWD instruction (convert word to doubleword), which sign-extends AX into EAX before performing the division:

```
.data
wordVal SWORD -101     ; 009Bh
.code
mov  eax,0             ; EAX = 00000000h
mov  ax,wordVal        ; EAX = 0000009Bh  (+155)
cwd                    ; EAX = FFFFFF9Bh  (-101)
mov  bx,2              ; EBX is the divisor
idiv bx                ; divide EAX by BX
```

We introduced the concept of sign extension in Chapter 4 along with the MOVSX instruction. The x86 instruction set includes several instructions for sign extension. First, we will look at these instructions, and then we will apply them to the signed integer division instruction, IDIV.

Sign Extension Instructions (CBW, CWD, CDQ)

Intel provides three sign extension instructions: CBW, CWD, and CDQ. The CBW instruction (convert byte to word) extends the sign bit of AL into AH, preserving the number's sign. In the next example, 9Bh (in AL) and FF9Bh (in AX) both equal −101 decimal:

```
.data
byteVal SBYTE -101             ; 9Bh
.code
mov  al,byteVal                ; AL = 9Bh
cbw                            ; AX = FF9Bh
```

The CWD (convert word to doubleword) instruction extends the sign bit of AX into DX:

```
.data
wordVal SWORD -101              ; FF9Bh
.code
mov  ax,wordVal                 ; AX = FF9Bh
cwd                             ; DX:AX = FFFFFF9Bh
```

The CDQ (convert doubleword to quadword) instruction extends the sign bit of EAX into EDX:

```
.data
dwordVal SDWORD -101            ; FFFFFF9Bh
.code
mov  eax,dwordVal
cdq                             ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

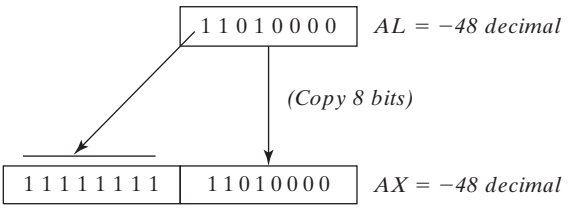
The IDIV Instruction

The IDIV (signed divide) instruction performs signed integer division, using the same operands as DIV. Before executing 8-bit division, the dividend (AX) must be completely sign-extended. The remainder always has the same sign as the dividend.

Example 1 The following instructions divide −48 by 5. After IDIV executes, the quotient in AL is −9 and the remainder in AH is −3:

```
.data
byteVal SBYTE -48               ; D0 hexadecimal
.code
mov  al,byteVal                 ; lower half of dividend
cbw                             ; extend AL into AH
mov  bl,+5                      ; divisor
idiv bl                         ; AL = -9, AH = -3
```

The following illustration shows how AL is sign-extended into AX by the CBW instruction:



To understand why sign extension of the dividend is necessary, let's repeat the previous example without using sign extension. The following code initializes AH to zero so it has a known value, and then divides without using CBW to prepare the dividend:

```
.data
byteVal SBYTE -48               ; D0 hexadecimal
.code
mov  ah,0                       ; upper half of dividend
mov  al,byteVal                 ; lower half of dividend
mov  bl,+5                      ; divisor
idiv bl                         ; AL = 41, AH = 3
```

Before the division, AX = 00D0h (208 decimal). IDIV divides this by 5, producing a quotient of 41 decimal, and a remainder of 3. That is certainly not the correct answer.

Example 2 16-Bit division requires AX to be sign-extended into DX. The next example divides −5000 by 256:

```
.data
wordVal SWORD -5000
.code
mov   ax,wordVal              ; dividend, low
cwd                           ; extend AX into DX
mov   bx,+256                 ; divisor
idiv  bx                      ; quotient AX = -19, rem DX = -136
```

Example 3 32-Bit division requires EAX to be sign-extended into EDX. The next example divides 50,000 by −256:

```
.data
dwordVal SDWORD +50000
.code
mov   eax,dwordVal            ; dividend, low
cdq                           ; extend EAX into EDX
mov   ebx,-256                ; divisor
idiv  ebx                     ; quotient EAX = -195, rem EDX = +80
```

All arithmetic status flag values are undefined after executing DIV and IDIV.

Divide Overflow

If a division operand produces a quotient that will not fit into the destination operand, a *divide overflow* condition results. This causes a processor exception and halts the current program. The following instructions, for example, generate a divide overflow because the quotient (100h) is too large for the 8-bit AL destination register:

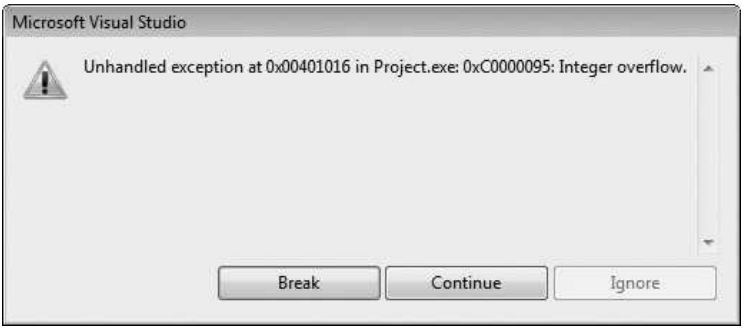
```
mov  ax,1000h
mov  bl,10h
div  bl                       ; AL cannot hold 100h
```

When this code executes, Fig. 7-1 shows the resulting error dialog produced by Visual Studio. A similar dialog window appears when you execute code that attempts to divide by zero.

Here's a suggestion: use a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition. In the following code, the divisor is EBX, and the dividend is placed in the 64-bit combined EDX and EAX registers:

```
mov  eax,1000h
cdq
mov  ebx,10h
div  ebx                      ; EAX = 00000100h
```

Figure 7–1 Divide overflow error example.



To prevent division by zero, test the divisor before dividing:

```
mov  ax,dividend
mov  bl,divisor
cmp  bl,0                  ; check the divisor
je   NoDivideZero          ; zero? display error
div  bl                    ; not zero: continue
.
.
NoDivideZero:              ;(display "Attempt to divide by zero")
```

7.3.6 Implementing Arithmetic Expressions

Chapter 4 showed how to implement arithmetic expressions using addition and subtraction. We can now include multiplication and division. Implementing arithmetic expressions at first seems to be an activity best left for compiler writers, but there is much to be gained by hands-on study. You can learn how compilers optimize code. Also, you can implement better error checking than a typical compiler by checking the size of the product following multiplication operations. Most high-level language compilers ignore the upper 32 bits of the product when multiplying two 32-bit operands. In assembly language, however, you can use the Carry and Overflow flags to tell you when the product does not fit into 32 bits. The use of these flags was explained in Sections 7.4.1 and 7.4.2.

> ***Tip:*** There are two easy ways to view assembly code generated by a C++ compiler: While debugging in Visual Studio, right-click in the debug window and select *Go to Disassembly.* Alternatively, to generate a listing file, select *Properties* from the Project menu. Under *Configuration Properties,* select *Microsoft Macro Assembler.* Then select *Listing File*. In the dialog window, set *Generate Preprocessed Source Listing* to *Yes,* and set *List All Available Information* to *Yes.*

Example 1 Implement the following C++ statement in assembly language, using unsigned 32-bit integers:

```
var4 = (var1 + var2) * var3;
```

This is a straightforward problem because we can work from left to right (addition, then multiplication). After the second instruction, EAX contains the sum of **var1** and **var2**. In the third instruction, EAX is multiplied by **var3** and the product is stored in EAX:

```
    mov  eax,var1
    add  eax,var2
    mul  var3                  ; EAX = EAX * var3
    jc   tooBig                ; unsigned overflow?
    mov  var4,eax
    jmp  next
tooBig:                        ; display error message
```

If the MUL instruction generates a product larger than 32 bits, the JC instruction jumps to a label that handles the error.

Example 2 Implement the following C++ statement, using unsigned 32-bit integers:

```
var4 = (var1 * 5) / (var2 - 3);
```

In this example, there are two subexpressions within parentheses. The left side can be assigned to EDX:EAX, so it is not necessary to check for overflow. The right side is assigned to EBX, and the final division completes the expression:

```
mov  eax,var1                  ; left side
mov  ebx,5
mul  ebx                       ; EDX:EAX = product
mov  ebx,var2                  ; right side
sub  ebx,3
div  ebx                       ; final division
mov  var4,eax
```

Example 3 Implement the following C++ statement, using signed 32-bit integers:

```
var4 = (var1 * -5) / (-var2 % var3);
```

This example is a little trickier than the previous ones. We can begin with the expression on the right side and store its value in EBX. Because the operands are signed, it is important to sign-extend the dividend into EDX and use the IDIV instruction:

```
mov   eax,var2                 ; begin right side
neg   eax
cdq                            ; sign-extend dividend
idiv  var3                     ; EDX = remainder
mov   ebx,edx                  ; EBX = right side
```

Next, we calculate the expression on the left side, storing the product in EDX:EAX:

```
mov   eax,-5                   ; begin left side
imul  var1                     ; EDX:EAX = left side
```

Finally, the left side (EDX:EAX) is divided by the right side (EBX):

```
idiv  ebx                      ; final division
mov   var4,eax                 ; quotient
```

7.3.7 Section Review

1. Explain why overflow cannot occur when the MUL and one-operand IMUL instructions execute.
2. How is the one-operand IMUL instruction different from MUL in the way it generates a multiplication product?
3. What has to happen in order for the one-operand IMUL to set the Carry and Overflow flags?
4. When EBX is the operand in a DIV instruction, which register holds the quotient?
5. When BX is the operand in a DIV instruction, which register holds the quotient?
6. When BL is the operand in a MUL instruction, which registers hold the product?
7. Show an example of sign extension before calling the IDIV instruction with a 16-bit operand.

7.4 Extended Addition and Subtraction

Extended precision addition and subtraction is the technique of adding and subtracting numbers having an almost unlimited size. In C++, for example, no standard operator permits you to add two 1024-bit integers. But in assembly language, the ADC (add with carry) and SBB (subtract with borrow) instructions are well suited to this type of operation.

7.4.1 ADC Instruction

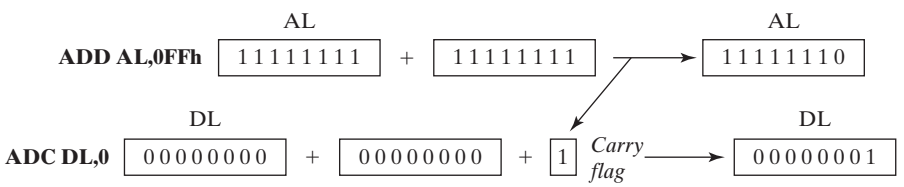
The ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand. The instruction formats are the same as for the ADD instruction, and the operands must be the same size:

```
ADC  reg,reg
ADC  mem,reg
ADC  reg,mem
ADC  mem,imm
ADC  reg,imm
```

For example, the following instructions add two 8-bit integers (FFh + FFh), producing a 16-bit sum in DL:AL, which is 01FEh:

```
mov  dl,0
mov  al,0FFh
add  al,0FFh                    ; AL = FEh
adc  dl,0                       ; DL/AL = 01FEh
```

The following illustration shows the movement of data during the two addition steps. First, FFh is added to AL, producing FEh in the AL register and setting the Carry flag. Next, both 0 and the contents of the Carry flag are added to the DL register:



Similarly, the following instructions add two 32-bit integers (FFFFFFFFh + FFFFFFFFh), producing a 64-bit sum in EDX:EAX: 00000001FFFFFFFEh:

```
mov  edx,0
mov  eax,0FFFFFFFFh
add  eax,0FFFFFFFFh
adc  edx,0
```

7.4.2 Extended Addition Example

Next, we would like to demonstrate a procedure named **Extended_Add** that adds two extended integers of the same size. Using a loop, it works its way through the two extended integers as if they were parallel arrays. As it adds each matching pair of values in the arrays, it includes the value of the carry from the addition that was performed during the previous iteration of the loop. Our implementation assumes that the integers are stored as arrays of bytes, but the example could easily be modified to add arrays of doublewords.

The procedure receives two pointers in ESI and EDI that point to the integers to be added. The EBX register points to a buffer in which the bytes of the sum will be stored, with the precondition that this buffer must be one byte longer than the two integers. Also, the procedure receives the length of the longest integer in ECX. The numbers must be stored in little-endian order, with the lowest order byte at each array's starting offset. Here's the code, with line numbers added so we can discuss it in detail:

```
 1:    ;-----------------------------------------------------
 2:    Extended_Add PROC
 3:    ;
 4:    ; Calculates the sum of two extended integers stored
 5:    ; as arrays of bytes.
 6:    ; Receives: ESI and EDI point to the two integers,
 7:    ;      EBX points to a variable that will hold the sum,
 8:    ;      and ECX indicates the number of bytes to be added.
 9:    ; Storage for the sum must be one byte longer than the
10:    ;      input operands.
11:    ; Returns: nothing
12:    ;-----------------------------------------------------
13:        pushad
14:        clc                         ; clear the Carry flag
15:
16:    L1: mov   al,[esi]              ; get the first integer
17:        adc   al,[edi]              ; add the second integer
18:        pushfd                      ; save the Carry flag
19:        mov   [ebx],al              ; store partial sum
20:        add   esi,1                 ; advance all three pointers
21:        add   edi,1
22:        add   ebx,1
23:        popfd                       ; restore the Carry flag
24:        loop  L1                    ; repeat the loop
25:
26:        mov   byte ptr [ebx],0      ; clear high byte of sum
```

```
27:         adc   byte ptr [ebx],0     ; add any leftover carry
28:         popad
29:         ret
30:     Extended_Add ENDP
```

When lines 16 and 17 add the first two low-order bytes, the addition might set the Carry flag. Therefore, it's important to save the Carry flag by pushing it on the stack on line 18, because we will need it when the loop repeats. Line 19 saves the first byte of the sum, and lines 20–22 advance all three pointers (for the two operands and the sum array). Line 23 restores the Carry flag, and line 24 continues the loop back to line 16. (The LOOP instruction never modifies the CPU status flags.) As the loop repeats, line 17 adds the next pair of bytes, and includes the value of the Carry flag. So if a Carry had been generated during the first pass through the loop, that Carry would be included during the second pass through the loop. The loop continues this way until all bytes have been added. Then, finally lines 26 and 27 look for any Carry that was generated when the two highest bytes of the operand were added, and adds this Carry to the extra byte in the sum operand.

The following sample code calls **Extended_Add**, passing it two 8-byte integers. We are careful to allocate an extra byte for the sum:

```
.data
op1 BYTE 34h,12h,98h,74h,06h,0A4h,0B2h,0A2h
op2 BYTE 02h,45h,23h,00h,00h,87h,10h,80h
sum BYTE 9 dup(0)

.code
main PROC
     mov   esi,OFFSET op1              ; first operand
     mov   edi,OFFSET op2              ; second operand
     mov   ebx,OFFSET sum              ; sum operand
     mov   ecx,LENGTHOF op1            ; number of bytes
     call  Extended_Add

; Display the sum.

     mov   esi,OFFSET sum
     mov   ecx,LENGTHOF sum
     call  Display_Sum
     call  Crlf
```

The following output is produced by the program. The addition produces a carry:

```
0122C32B0674BB5736
```

The **Display_Sum** procedure (from the same program) displays the sum in its proper order, starting with the high-order byte, and working its way down to the low-order byte:

```
Display_Sum PROC
     pushad
     ; point to the last array element
     add  esi,ecx
     sub  esi,TYPE BYTE
     mov  ebx,TYPE BYTE
```

```
L1: mov  al,[esi]              ; get an array byte
    call WriteHexB             ; display it
    sub  esi,TYPE BYTE         ; point to previous byte
    loop L1

    popad
    ret
Display_Sum ENDP
```

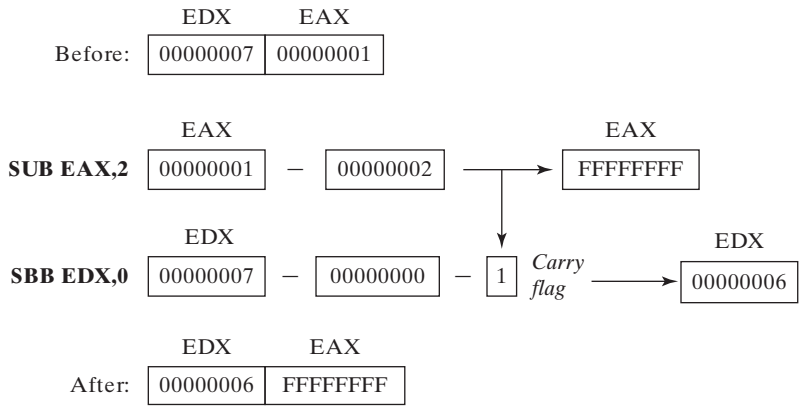
7.4.3 SBB Instruction

The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand. The possible operands are the same as for the ADC instruction. The following example code carries out 64-bit subtraction with 32-bit operands. It sets EDX:EAX to 0000000700000001h and subtracts 2 from this value. The lower 32 bits are subtracted first, setting the Carry flag. Then the upper 32 bits are subtracted, including the Carry flag:

```
mov  edx,7                     ; upper half
mov  eax,1                     ; lower half
sub  eax,2                     ; subtract 2
sbb  edx,0                     ; subtract upper half
```

Figure 7-2 demonstrates the movement of data during the two subtraction steps. First, the value 2 is subtracted from EAX, producing FFFFFFFFh in EAX. The Carry flag is set because a borrow is required when subtracting a larger number from a smaller one. Next the SBB instruction subtracts both 0 and the contents of the Carry flag from EDX.

Figure 7–2 Subtracting from a 64-bit integer using SBB.



7.4.4 Section Review

1. Describe the ADC instruction.
2. Describe the SBB instruction.

3. What will be the values of EDX:EAX after the following instructions execute?

```
mov  edx,10h
mov  eax,0A0000000h
add  eax,20000000h
adc  edx,0
```

4. What will be the values of EDX:EAX after the following instructions execute?

```
mov  edx,100h
mov  eax,80000000h
sub  eax,90000000h
sbb  edx,0
```

5. What will be the contents of DX after the following instructions execute (STC sets the Carry flag)?

```
mov  dx,5
stc                              ; set Carry flag
mov  ax,10h
adc  dx,ax
```

7.5 ASCII and Unpacked Decimal Arithmetic

(The instructions discussed in Section 7.5 apply only to programming in 32-bit mode.) The integer arithmetic shown so far in this book has dealt only with binary values. The CPU calculates in binary, but is also able to perform arithmetic on ASCII decimal strings. The latter can be conveniently entered by the user and displayed in the console window, without requiring them to be converted to binary. Suppose a program is to input two numbers from the user and add them together. The following is a sample of the output, in which the user has entered 3402 and 1256:

```
Enter first number:   3402
Enter second number:  1256
The sum is:           4658
```

We have two options when calculating and displaying the sum:

1. Convert both operands to binary, add the binary values, and convert the sum from binary to ASCII digit strings.
2. Add the digit strings directly by successively adding each pair of ASCII digits (2 + 6, 0 + 5, 4 + 2, and 3 + 1). The sum is an ASCII digit string, so it can be directly displayed on the screen.

The second option requires the use of specialized instructions that adjust the sum after adding each pair of ASCII digits. Four instructions that deal with ASCII addition, subtraction, multiplication, and division are as follows:

AAA	(ASCII adjust after addition)
AAS	(ASCII adjust after subtraction)
AAM	(ASCII adjust after multiplication)
AAD	(ASCII adjust before division)

ASCII Decimal and Unpacked Decimal The high 4 bits of an unpacked decimal integer are always zeros, whereas the same bits in an ASCII decimal number are equal to 0011b. In any case, both types of integers store one digit per byte. The following example shows how 3402 would be stored in both formats:

(All values are in hexadecimal)

Although ASCII arithmetic executes more slowly than binary arithmetic, it has two distinct advantages:

- Conversion from string format before performing arithmetic is not necessary.
- Using an assumed decimal point permits operations on real numbers without danger of the roundoff errors that occur with floating-point numbers.

ASCII addition and subtraction permit operands to be in ASCII format or unpacked decimal format. Only unpacked decimal numbers can be used for multiplication and division.

7.5.1 AAA Instruction

In 32-bit Mode, the AAA (ASCII adjust after addition) instruction adjusts the binary result of an ADD or ADC instruction. Assuming that AL contains a binary value produced by adding two ASCII digits, AAA converts AL to two unpacked decimal digits and stores them in AH and AL. Once in unpacked format, AH and AL can easily be converted to ASCII by ORing them with 30h.

The following example shows how to add the ASCII digits 8 and 2 correctly, using the AAA instruction. You must clear AH to zero before performing the addition or it will influence the result returned by AAA. The last instruction converts AH and AL to ASCII digits:

```
mov  ah,0
mov  al,'8'                ; AX = 0038h
add  al,'2'                ; AX = 006Ah
aaa                        ; AX = 0100h (ASCII adjust result)
or   ax,3030h              ; AX = 3130h = '10' (convert to ASCII)
```

Multibyte Addition Using AAA

Let's look at a procedure that adds ASCII decimal values with implied decimal points. The implementation is a bit more complex than one would imagine because the carry from each digit addition must be propagated to the next highest position. In the following pseudocode, the name *acc* refers to an 8-bit accumulator register:

```
esi (index) = length of first_number - 1
edi (index) = length of first_number
ecx = length of first_number
set carry value to 0
Loop
     acc = first_number[esi]
     add previous carry to acc
     save carry in carry1
     acc += second_number[esi]
```

```
    OR the carry with carry1
    sum[edi] = acc
    dec edi
Until ecx == 0
Store last carry digit in sum
```

The carry digit must always be converted to ASCII. When you add the carry digit to the first operand, you must adjust the result with AAA. Here is the listing:

```
; ASCII Addition                          (ASCII_add.asm)

; Perform ASCII arithmetic on strings having
; an implied fixed decimal point.

INCLUDE Irvine32.inc

DECIMAL_OFFSET = 5              ; offset from right of string
.data
decimal_one BYTE "100123456789765"    ; 1001234567.89765
decimal_two BYTE "900402076502015"    ; 9004020765.02015
sum BYTE (SIZEOF decimal_one + 1) DUP(0),0

.code
main PROC
; Start at the last digit position.
    mov   esi,SIZEOF decimal_one - 1
    mov   edi,SIZEOF decimal_one
    mov   ecx,SIZEOF decimal_one
    mov   bh,0                  ; set carry value to zero

L1: mov   ah,0                  ; clear AH before addition
    mov   al,decimal_one[esi]   ; get the first digit
    add   al,bh                 ; add the previous carry
    aaa                         ; adjust the sum (AH = carry)
    mov   bh,ah                 ; save the carry in carry1
    or    bh,30h                ; convert it to ASCII
    add   al,decimal_two[esi]   ; add the second digit
    aaa                         ; adjust the sum (AH = carry)
    or    bh,ah                 ; OR the carry with carry1
    or    bh,30h                ; convert it to ASCII
    or    al,30h                ; convert AL back to ASCII
    mov   sum[edi],al           ; save it in the sum
    dec   esi                   ; back up one digit
    dec   edi
    loop  L1
    mov   sum[edi],bh           ; save last carry digit

; Display the sum as a string.
    mov   edx,OFFSET sum
    call  WriteString
    call  Crlf

    exit
main ENDP
END main
```

Here is the program's output, showing the sum without a decimal point:

```
1000525533291780
```

7.5.2 AAS Instruction

In 32-bit Mode, the AAS (ASCII adjust after subtraction) instruction follows a SUB or SBB instruction that has subtracted one unpacked decimal value from another and stored the result in AL. It makes the result in AL consistent with ASCII digit representation. Adjustment is necessary only when the subtraction generates a negative result. For example, the following statements subtract ASCII 9 from 8:

```
.data
val1 BYTE '8'
val2 BYTE '9'
.code
mov  ah,0
mov  al,val1                    ; AX = 0038h
sub  al,val2                    ; AX = 00FFh
aas                             ; AX = FF09h
pushf                           ; save the Carry flag
or   al,30h                     ; AX = FF39h
popf                            ; restore the Carry flag
```

After the SUB instruction, AX equals 00FFh. The AAS instruction converts AL to 09h and subtracts 1 from AH, setting it to FFh and setting the Carry flag.

7.5.3 AAM Instruction

In 32-bit Mode, the AAM (ASCII adjust after multiplication) instruction converts the binary product produced by MUL to unpacked decimal. The multiplication can only use unpacked decimals. In the following example, we multiply 5 by 6 and adjust the result in AX. After adjustment, AX = 0300h, the unpacked decimal representation of 30:

```
.data
AscVal BYTE 05h,06h
.code
mov  bl,ascVal                  ; first operand
mov  al,[ascVal+1]              ; second operand
mul  bl                         ; AX = 001Eh
aam                             ; AX = 0300h
```

7.5.4 AAD Instruction

In 32-Bit Mode, the AAD (ASCII adjust before division) instruction converts an unpacked decimal dividend in AX to binary in preparation for executing the DIV instruction. The following example converts unpacked 0307h to binary, then divides it by 5. DIV produces a quotient of 07h in AL and a remainder of 02h in AH:

```
.data
quotient  BYTE ?
remainder BYTE ?
.code
```

```
mov  ax,0307h                    ; dividend
aad                              ; AX = 0025h
mov  bl,5                        ; divisor
div  bl                          ; AX = 0207h
mov  quotient,al
mov  remainder,ah
```

7.5.5 Section Review

1. Write a single instruction that converts a two-digit unpacked decimal integer in AX to ASCII decimal.
2. Write a single instruction that converts a two-digit ASCII decimal integer in AX to unpacked decimal format.
3. Write a two-instruction sequence that converts a two-digit ASCII decimal number in AX to binary.
4. Write a single instruction that converts an unsigned binary integer in AX to unpacked decimal.

7.6 Packed Decimal Arithmetic

(The instructions discussed in Section 7.6 apply only to programming in 32-bit mode.) Packed decimal integers store two decimal digits per byte. Each digit is represented by 4 bits. If there is an odd number of digits, the highest nybble is filled with a zero. Storage sizes may vary:

```
bcd1 QWORD 2345673928737285h     ; 2,345,673,928,737,285 decimal
bcd2 DWORD 12345678h             ; 12,345,678 decimal
bcd3 DWORD 08723654h             ; 8,723,654 decimal
bcd4 WORD 9345h                  ; 9,345 decimal
bcd5 WORD 0237h                  ; 237 decimal
bcd6 BYTE 34h                    ; 34 decimal
```

Packed decimal storage has at least two strengths:

- The numbers can have almost any number of significant digits. This makes it possible to perform calculations with a great deal of accuracy.
- Conversion of packed decimal numbers to ASCII (and vice versa) is relatively simple.

Two instructions, DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction), adjust the result of an addition or subtraction operation on packed decimals. Unfortunately, no such instructions exist for multiplication and division. In those cases, the number must be unpacked, multiplied or divided, and repacked.

7.6.1 DAA Instruction

In 32-bit Mode, the DAA (decimal adjust after addition) instruction converts a binary sum produced by ADD or ADC in AL to packed decimal format. For example, the following instructions add packed decimals 35 and 48. The binary sum (7Dh) is adjusted to 83h, the packed decimal sum of 35 and 48.

```
mov  al,35h
add  al,48h                      ; AL = 7Dh
daa                              ; AL = 83h (adjusted result)
```

The internal logic of DAA is documented in the Intel Instruction Set Reference Manual.

Example The following program adds two 16-bit packed decimal integers and stores the sum in a packed doubleword. Addition requires the sum variable to contain space for one more digit than the operands:

```
; Packed Decimal Example        (AddPacked.asm)

; Demonstrate packed decimal addition.
INCLUDE Irvine32.inc

.data
packed_1 WORD 4536h
packed_2 WORD 7207h
sum DWORD ?

.code
main PROC
; Initialize sum and index.
    mov   sum,0
    mov   esi,0

; Add low bytes.
    mov   al,BYTE PTR packed_1[esi]
    add   al,BYTE PTR packed_2[esi]
    daa
    mov   BYTE PTR sum[esi],al

; Add high bytes, include carry.
    inc   esi
    mov   al,BYTE PTR packed_1[esi]
    adc   al,BYTE PTR packed_2[esi]
    daa
    mov   BYTE PTR sum[esi],al

; Add final carry, if any.
    inc   esi
    mov   al,0
    adc   al,0
    mov   BYTE PTR sum[esi],al

; Display the sum in hexadecimal.
    mov   eax,sum
    call  WriteHex
    call  Crlf
    exit
main ENDP
END main
```

Needless to say, the program contains repetitive code that suggests using a loop. One of the chapter exercises will ask you to create a procedure that adds packed decimal integers of any size.

7.6.2 DAS Instruction

In 32-bit Mode, the DAS (decimal adjust after subtraction) instruction converts the binary result of a SUB or SBB instruction in AL to packed decimal format. For example, the following statements subtract packed decimal 48 from 85 and adjust the result:

```
mov  bl,48h
mov  al,85h
sub  al,bl                      ; AL = 3Dh
das                             ; AL = 37h  (adjusted result)
```

The internal logic of DAS is documented in the Intel Instruction Set Reference Manual.

7.6.3 Section Review

1. Under what circumstances does DAA instruction set the Carry flag? Give an example.
2. Under what circumstances does DAS instruction set the Carry flag? Give an example.
3. When adding two packed decimal integers of length *n* bytes, how many storage bytes must be reserved for the sum?

7.7 Chapter Summary

Along with the bitwise instructions from the preceding chapter, shift instructions are among the most characteristic of assembly language. To *shift* a number means to move its bits right or left.

The SHL (shift left) instruction shifts each bit in a destination operand to the left, filling the lowest bit with 0. One of the best uses of SHL is for performing high-speed multiplication by powers of 2. Shifting any operand left by *n* bits multiplies the operand by 2^n. The SHR (shift right) instruction shifts each bit to the right, replacing the highest bit with a 0. Shifting any operand right by *n* bits divides the operand by 2^n.

SAL (shift arithmetic left) and SAR (shift arithmetic right) are shift instructions specifically designed for shifting signed numbers.

The ROL (rotate left) instruction shifts each bit to the left and copies the highest bit to both the Carry flag and the lowest bit position. The ROR (rotate right) instruction shifts each bit to the right and copies the lowest bit to both the Carry flag and the highest bit position.

The RCL (rotate carry left) instruction shifts each bit to the left and copies the highest bit into the Carry flag, which is first copied into the lowest bit of the result. The RCR (rotate carry right) instruction shifts each bit to the right and copies the lowest bit into the Carry flag. The Carry flag is copied into the highest bit of the result.

The SHLD (shift left double) and SHRD (shift right double) instructions, available on x86 processors, are particularly effective for shifting bits in large integers.

In 32-bit mode, the MUL instruction multiplies an 8-, 16-, or 32-bit operand by AL, AX, or EAX. In 64-bit mode, a value can also be multiplied by the RAX register. The IMUL instruction performs signed integer multiplication. It has three formats: single operand, double operand, and three operand.

In 32-bit mode, the DIV instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers. In 64-bit mode, you can also perform 64-bit division. The IDIV instruction performs signed integer division, using the same operands as the DIV instruction.

The CBW (convert byte to word) instruction extends the sign bit of AL into the AH register. The CDQ (convert doubleword to quadword) instruction extends the sign bit of EAX into the EDX register. The CWD (convert word to doubleword) instruction extends the sign bit of AX into the DX register.

Extended addition and subtraction refers to adding and subtracting integers of arbitrary size. The ADC and SBB instructions can be used to implement such addition and subtraction. The ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand. The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.

ASCII decimal integers store one digit per byte, encoded as an ASCII digit. The AAA (ASCII adjust after addition) instruction converts the binary result of an ADD or ADC instruction to ASCII decimal. The AAS (ASCII adjust after subtraction) instruction converts the binary result of a SUB or SBB instruction to ASCII decimal. All of these instructions are available only in 32-bit mode.

Unpacked decimal integers store one decimal digit per byte, as a binary value. The AAM (ASCII adjust after multiplication) instruction converts the binary product of a MUL instruction to unpacked decimal. The AAD (ASCII adjust before division) instruction converts an unpacked decimal dividend to binary in preparation for the DIV instruction. All of these instructions are available only in 32-bit mode.

Packed decimal integers store two decimal digits per byte. The DAA (decimal adjust after addition) instruction converts the binary result of an ADD or ADC instruction to packed decimal. The DAS (decimal adjust after subtraction) instruction converts the binary result of a SUB or SBB instruction to packed decimal. All of these instructions are available only in 32-bit mode.

7.8 Key Terms

7.8.1 Terms

arithmetic shift
binary multiplication
bit rotation
bit shifting
bit strings
bitwise division
bitwise multiplication
bitwise rotation
divide overflow
little-endian order
logical shift
sign extension
signed division
signed multiplication
signed overflow
unsigned multiplication

7.8.2 Instructions, Operators, and Directives

AAA
AAD
AAM
AAS
ADC
CBQ
CBW
DAA
DAS

DIV
IDIV
IMUL
MUL
RCL
RCR
ROL
ROR
SAL
SAR
SBB
SHL
SHLD
SHR
SHRD