必考

3. After the code is executed, what are the values of the registers? (4%)

```
mov ax, 1234h
mov dx, 5678h
shr dx, 1
rcr ax, 1
```

registers	ax	dx
value	091A h	2B3C h

Status Flags The following CPU status flags are affected by arithmetic operations:

- The Sign flag is set when the outcome of an arithmetic operation is negative.
- The Carry flag is set when the result of an unsigned arithmetic operation is too large for the destination operand.
- 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 carry or borrow occurs in bit position 3 of the destination operand.
- The Zero flag is set when the outcome of an arithmetic operation is zero.
- The Overflow flag is set when the result of an signed arithmetic operation is out of range for the destination operand.

必考
2020考過

6. The table shows the initial values saved in the registers before the code is executed (16%, 2% for each blank)

ebx	45cba7ddh
esp	0019ff74h

```
.data
Char BYTE 'c'
.code
Procedure PROC uses bx
        and bl, 11011111b
        mov al, bl
        L2:
                ret
Procedure ENDP
main PROC
        movzx bx, Char
        call Procedure
        L1:
                exit
Main ENDP
```

(1) When the code is executed to L2 position, the stack data are:

0019ff6eh	0063h
0019ff70h	L1
0019ff74h	75e13c55h

(2) Following Problem (1), what are the ~~IG~~ values saved in the registers?

ebx	45cb0043h
esp	0019ff6eh

(3) What are the register values when the program stops at L1?

ebx	45cb0063h
esp	0019ff74h

(4) What is the purpose of the program?

將小寫英文字母轉成大寫
並將大寫英文字母放進 al 中
小寫依然存放小寫英文字母

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

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

必考
2021考過

6. The table shows the initial values saved in the registers before the code is executed. (16%, 2% for each blank)

ebx	45cba7ddh
esp	0019ff74h

```
.data
Char BYTE 'c'
.code
Procedure PROC uses bx
        and bl, 11011111b
        mov al, bl
        L2:
                ret
Procedure ENDP
main PROC
        movzx bx, Char
        call Procedure
        L1:
          exit
main ENDP
```

(1) When the code is executed to L2 p:osition, the stack data are:

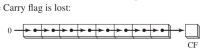
0019ff6eh	0063h
0019ff70h	L1
0019ff74h	75e13c55h

(2) Following Problem (1), what are the values saved in the registers?

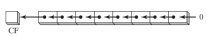
ebx	45cb0043h
esp	0019ff6eh

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:

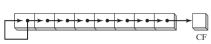
7.1.3 SHR Instruction
The SHR (shift right) instruction performs a logical right shift on the destination operand, replacing the highest bit with a 0. The lowest bit is copied into the Carry flag, and the bit that was previously in the Carry flag is lost:



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:



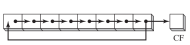
The SAR (shift arithmetic right) instruction performs a right arithmetic shift on its destination operand:



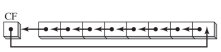
7.1.5 ROL Instruction
Bitwise rotation occurs when you move the bits in a circular fashion. In some versions, the bit leaving one end of the number is immediately copied into the other end. Another type of rotation uses the Carry flag as an intermediate point for shifted bits.

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:

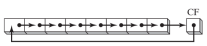
7.1.6 ROR Instruction
The ROR (rotate right) instruction shifts each bit to the right and copies the lowest bit into the Carry flag and the highest bit position. The instruction format is the same as for SHL:

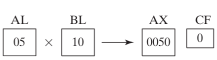


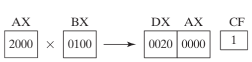
7.1.7 RCL and RCR Instructions
The RCL (rotate carry left) instruction shifts each bit to the left, copies the Carry flag to the LSB, and copies the MSB into the Carry flag:



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:



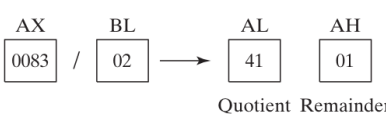


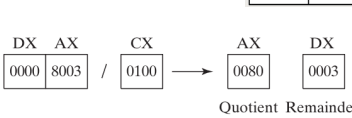


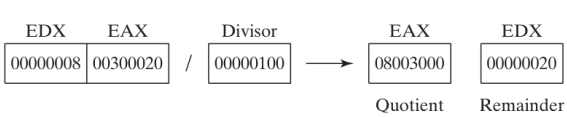
Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	r/m8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX

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







1. Translate 8-bit binary 00101101 (in 2's complement format) to the hexadecimal and Decimal. (4%)

Hexadecimal	8-bit binary	Decimal
2D h	00101101	45

2. After the code is executed, what are the values of the registers? (8%, 2% for each blank)

```
mov al, 52
mov bl, 4
imul bl
```

OF	CF	ah	al
1	0	00h	b0 h

4. Explain the difference between the two instructions in each of the following sets of instructions. (6%)

```
mov ax, bx
```
將 bx 的值移入 ax

```
mov ax, [bx]
```
bx 存放一位址
將此位址指向的值移入 ax 中

5. Suppose AX contains —32,768 and apply NEG to it. Will the result be valid and why? (6%)

```
neg  AX    32768
```
AX 最多存放 16 bits , -32768 以 hexadecimal 表示為 ffffh
neg 後我們會得到 0001h , 為十進位的 1 , 而不是 32768
因在有符號數的情況下 , AX 的範圍為 32767 ~ -32768

7. Please find the executing sequence of the instruction that match the flags and registers value.

OF	SF	ZF	CF	ah	al	
0	1	0	0	abh	00h	Initial value
0	0	1	0	00h	00h	xor ax, ax
0	0	1	0	00h	e0h	mov al, e0h
0	1	0	0	a0 h	e0h	or ah, 10100000b
1	0	0	1	a0 h	90 h	add al, 90h

8. We want to find out the largest value and put it in ax. Fill in t the code. (12%, 4% for each blank)

```
.data
        Val1 WORD 1234h
        Val2 WORD 8765h
        Val3 WORD 1111h
.code
        mov ax, Val1
        cmp ax, Val2
        jae  L1        ; jump L1
        mov ax, Val2
        L1:
        cmp  ax , Val3  compare ax and Val3
        jae  L2        ; jump L2
        mov ax, Val3
        Exit
```

9. Array summation and write the values of the registers(12%, 2% for each blank)

```
.data
intarray DWORD 10000h, 20000h, 30000h, 40000h
.code
main PROC
        mov ecx , LENGTHOF  intarray      ; Set the number for LOOP
executions
        mov eax, 0
        mov esi, 0
L1:
        add  eax , [ intarray + esi ]      ; Add an integer
L2:
        inc esi
        loop  L1
main ENDP
END main
```

registers	eax	ecx	esi
value	0000 0000 h	0000 0000 h	0000 0004 h

1. Translate hexadecimal 7Dh to the 8-bit binary (in 2's complement format) and Decimal. (4%, 2% for each blank)

Hexadecima	8-bit binary	Decimal
7D	0111 1101	125

10. Mr. Hsieh was the student the ASM course. Someday, he wrote down the multiplication code only using shift and add instructions, but it was erased accidently. (12% , 3% for each blank)

```
.data
        Datal BYTE 12     ; multiplicand
        Data2 BYTE 14     ; multiplier
        Data3 WORD ?      ; product a
.code
multiplication MACRO multiplicand, multiplier, product
        movzx ax, multiplicand
        mov bl, multiplier
        mov dx, product

        mov ecx, SIZEOF multiplier   ; determine the loop number by sizeof instruction
        shl ecx, 3                   ; shift loop number
L1:
        test bl, 1
        jz L2                        ; jump if bl's bit0 is 0
        add dx, ax
L2:
        shl ax, 1
        shr bl, 1
        LOOP LI
        mov product, dx
ENDM

main PROC
        multiplication Data1, Data 2, Data 3
                                     ; call macro function
exit
main ENDP
END main
        exit
main ENDP
```

. Please find the executing sequence of the instruction that match the registers value. (20%, 1% for each blank)

OF	SF	ZF	CF	ah	al	
0	1	0	0	ddh	00h	(Initial value)
0	0	1	0	00h	00h	sub ah,ddh
0	1	0	0	a0h	00h	or ah, 10100000b
0	1	0	0	a0h	e0h	mov al, e0h
1	0	0	1	a0h	90h	add al, 90h

注意CF

7. What are the values saved in the registers afier Line 3,4,5 is each executed? (16%,2% for each blank)

Line

1 mov ax, 77Bh
2 mov bx, 1000h
3 mul bx
4 rol ah, 3
5 ror ah, 4

	ax		bx			注意
	ah	al	bh	bl	dx	CF
Line 3	B0 h	00h	00h	13h	0077h	1
Line 4	85h	X	X	X	X	1
Line 5	58h	X	X	X	X	0

4. The following table shows the original state in each registers (9%, each register 3%)

registers	value
eax	00307546h
ebx	0035ee81h
esp	0019ff68h

```
mov eax, 0012a000h
mov ebx, 00133131h
push eax
push bx
pop eax
```

After the code is executed, what are the values of the registers?

registers	value
eax	0012 3131h
ebx	0013 3131h
esp	0019ff66h

1. Translate

Hexadecimal	8-bit binary number	unsigned	Signed (2's complement)
0AD h	1010 1101 b	173	−83

5. We want to add the even numbers from 20 to 29 and store the final result into *al*.

Fill in the blank to finish the code. (16%, 4% for each blank)

```
.data
  Num BYTE 20,21,22,23,24,25,26,27,28,29
.code
      mov ecx, LENGTHOF Num      (only accept Data Related Operators)
      shr ecx, 1
      ;Above two lines set the number of LOOP execution
      mov esi, OFFSET Num ; esi is pointing to Num[0]
      mov al,0
L1:
      mov bl,[esi]
      add al,bl
      inc esi
      inc esi
      loop L1
```

8. Complete the following code which contains MACRO: (12%, 4% for each blank)

```
PrintChar MACRO char
      IFB<char>          ; check parameter is empty or not
            EXITM
      ENDIF

      push eax           ; preserve the value of register
      mov al, char

      call WriteChar     ; to print out char
      pop eax            ; restore register
ENDM

main PROC
      mov bl, 'J'

      PrintChar bl       ; invoke MACRO with argument bl
      exit
main ENDP
```

2. The purpose of this program is to summon the values pointed by the variable myData in the memory, nothing that we handle the values in bytes. Fill in the result of al in the blank.

```
.data
      myData WORD 2019h, 108h, 11h
.code
      mov ax, 0
      mov ecx, SIZEOF myData
      mov esi, OFFSET myData
L1:
      add al, BYTE PTR [esi]
      inc esi
L:
      loop L1
```

At label L	al
1	19h
2	39h
3	41h
4	42h
5	53h
6	53h

3. What are the values of the registers after the code is executed?

```
mov ax, 1104h
mov bx, 100h
mov cx, 20h
mov dx, 19h
div bx
sar cx, 4
```

ax	bx	cx	dx
1911 h	0100 h	0002h	0004 h

4.

```
.data
      Val1 WORD 20, 40, 60 DUP(80)
.code
      add al, 2ah
      add al, LENGTHOF Val1
      mov ax, 0123 decimal
      sub ax, SIZEOF Val1
```

OF	SF	ZF	CF	ah	al
0	0	0	0	20h	19h
0	0	0	0	20h	43h
1	1	0	0	20h	81h
1	1	0	0	00h	7Eh
0	1	0	1	ffh	ffh

5. What are the values of the register al and the CF when the code is executed in the loop L1?

```
.data
      myArray BYTE 1, 2, 3
.code
main PROC
      mov ecx, LENGTHOF myArray
      mov esi, 0
L1:
      mov al, myArray[esi]
      rcr al, 1
      inc esi
L:
      LOOP L1
exit
main ENDP
```

Stop at label L the i-th time	al	CF
First time	00h	1
Second time	81h	0
Third time	01h	1

6.

```
.code
UseStack PROC
      push bx
      pop cx
      cmp cx, 10
      jae L1
      ror ax, 2
L1:
      push ax
L2:
      pop bx
      ret
      pop bx
UseStack ENDP
main PROC
      mov eax, 11h
      mov ebx, 04h
      mov ecx, 15h
      push cx
      call UseStack
L3:
      exit
main ENDP
```

A	B	A XOR B
true	true	false
true	false	true
false	true	true
false	false	false

7.

(a)

```
.data
      myData WORD 10h
.code
Procedure MACRO data
      IFB<data>          ; if the parameter is empty, exit macro
            EXITM
      ENDIF
      mov bx, [data]
      test bx, TYPE[data]
      je L1
      inc bx
      mov [data], bx
L1:
      dec bx
      mov [data], bx
ENDM
main PROC
      mov esi, OFFSET myData     ; store the offset of myData tp esi
      Procedure esi              ; MACRO call procedure with parameter esi
      exit
      main ENDP
```

(b)

The final value of myData is 04 h after the program is completed

Hexadecimal	8-bit binary	Decimal
6C	0110 1100	108

P	~P	Q	~Q	(~P OR Q) AND (P XOR ~Q)
T	F	T	F	T
T	F	F	T	F
F	T	T	F	F
F	T	F	T	T

8.

```
.data
      myString BYTE "abcde"
.code
      mov ecx, LENGTHOF myString
      mov eax, OFFSET myString
      mov ebx, OFFSET myString
      add ebx, ecx
      dec ebx
      shr ecx, 1
L:
      mov dh, [eax]
      mov dl, [ebx]
      cmp dh, dl
      jz L1
      xor dh, dl
      xor dl, dh
      xor dh, dl
      mov [eax], dh
      mov [ebx], dl
      inc eax
      dec ebx
L1:
      LOOP L
```

What is the purpose of the program?

將字串順序反轉

Stop at Label L1 the i-th time	myString value
Before the loop	abcde
First loop	ebcda
Second loop	edcba

The values saved in the registers before the code is executed

eax	F12E8F74h
ebx	003AB000h
ecx	00400100h
esp	0019FF74h

(2) What are the register values when programs stops at L3?

ebx	0000 4004h
esp	0019ff72h

(1) What are the register values when programs stops at L2?

0019ff6ch	4004h	eax	0000400 4h
0019ff6eh	L3	ebx	00000004h
0019ff72h	0015h	ecx	0000 0004h
0019ff74h	75e13c20h	esp	0019ff6ch

Stack

3. In the following data definition, assume that List begins at offset 100Ah.What is the offset of the third value (5): 1012 h. (5%)

List DWORD 3, 4, 5, 6, 7

4. Each of the following MOV statements is invalid connect its reason by line.

```
.data
bVal    BYTE    100
bVal2   BYTE    ?
wVal    WORD    2
dVal    DWORD   5
.code
mov ds, 45
mov esi, wVal
mov eip, dVal
mov 25, bVal
mov bVa12,bVal
```

- mov ds, 45 → immediate value cannot be destination
- mov esi, wVal → size mismatch
- mov eip, dVal → EIP cannot be the destination
- mov 25, bVal → memory-to-memory move not permitted
- mov bVa12,bVal → immediate move to DS not permitted

5. Based the code below and Val1 = 0ffh, when is not any command executed, registers values will be as follows: (10%)

```
movsx ax, Val1
mov dx, 0
add dx, ax
shl ax, 4
add dx, ax
```

EAX	00000000h
EBX	00000000h
ECX	0012fb08h
EDX	772070b4h

When all the command have been executed, what are registers values?

And DX = 17 * Val1 (write a number)

EAX	0000 ffff0 h
EBX	00000000h
ECX	0012fb08h
EDX	7720 ffef h

6. Fill the form below to show changes Carry Flag, Zero Flag, Sign Flag and Overflow Flag respectively in every following sequence. (16%)

```
.data
myArray WORD 00ffh
.code
    mov ax, [myArray]
    add al, 1
    mov al, 7fh
    add al, TYPE myArray
```

OF	SF	ZF	CF	ah	al
0	0	0	0	00	00h
0	0	0	0	00	ffh
0	0	1	1	00	00h
0	0	1	1	00	7fh
1	1	0	0	00	81h

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:

8. Before executing any instruction, the values of registers are as below:

EBP	0012ff94h
ESP	0012ff8ch
ESI	00000000h
EDI	00000000h

EAX	00000001h
EBX	00000003h
ECX	00000000h
EDX	00401000h

What will be the value of EAX • EBX • ESP when the code has executed? (10%)

```
mov eax, 5
push eax
mov eax, 10
push eax
pop ebx
```

EAX	0000000A h
EBX	0000000A h
ECX	00000000h
EDX	00401000h

EBP	0012ff94h
ESP	0012ff88h
ESI	00000000h
EDI	00000000h

1. Translate the decimal number 21 to 8-bit binary: 00010101
2. Translate the decimal number -21 to 8-bit binary (in 2's complement format): 11101011
3. What is the value (true or false) of the boolean expression X ∨ (Y ∧ Z), when X=false, Y=true, and Z=false? false
4. Which one of the following addresses is on a paragraph boundary? (a) 010101h, (b) 0101Ah, (c) 01010h, (d) 0101Bh. c. A paragraph = 16 bytes in size so the segment address is always divisible by 16 or 10h
5. The smallest signed integer stored in 4-bit binary is: 1000. Its decimal value is -8.
6. What decimal value is equivalent to hexadecimal 1Ah ? 26
7. What is the name of the lowest 16 bits of the EBX register?
   BH, BL, BX, or EX Answer: BX.
8. The actual address (or linear address) in real-address mode represented by the base-offset address 01A0:0200 is 01C00 h. 01A0 + 0200 = 01C00
9. In the following data definition, assume that **List2** begins at offset 1000h. What is the offset of the 4th value (6)? 100C h. 1000 + 4×3 = 100C

```
List2 DWORD 3, 4, 5, 6, 7
```

10. Which utility program reads an assembly language source file and produces an object file?
    Assembler
11. The basic parts of an instruction, in order from left to right, are:
    a. mnemonic, operand(s), comment
    b. label, mnemonic, comment
    c. comment, label, mnemonic, operand(s)
    d. label, mnemonic, operand(s), comment
    Answer: d
12. Which shows the memory byte order, from low to high address, of the following data definition?

```
Val WORD 0012h,0034h
```

    a. 12h,34h, 00h,00h
    b. 34h,00h,12h,00h
    c. 12h,00h,34h,00h
    d. 00h,00h,12h,34h
    Answer: C
13. Which of the following is a valid data definition statement that creates an array of bytes containing hexadecimal 1Ah, 2Ah, and 3Ah, named NS?
    a. NS BYTE DUP (3) 1Ah,2Ah,3Ah
    b. NS Array[3]: 1Ah, 2Ah,3Ah
    c. BYTE NS 1Ah, 2Ah, 3Ah
    d. NS BYTE 1Ah, 2Ah, 3Ah
    Answer: d.
14. Which two of the following are true about assembly language instructions and directives?
    a. a directive is executed at runtime
    b. an instruction can be translated into machine code
    c. an instruction is executed at assembly time
    d. a directive is interpreted by assembler at assembly time
    Answer: b and d.
15. Select a data definition statement that creates an array of 10 signed words named **myList** and initializes the first five elements with −1 and last five with 0.
    a. SWORD myList 5 DUP (−1), 5 DUP (0)
    b. myList 10 SWORD DUP (−1,0)
    c. myList SWORD 5 DUP (−1), 5 DUP (0)
    d. myList SWORD 5 DUP (−1,0)
    Answer: C.
16. Complete the following instructions in which **ArrayCount** automatically calculates the number of elements in the array **newArray**.:

```
DWORDSIZE = 4
newArray DWORD 10,20,30
ArrayCount = ($ - newArray)/DWORDSIZE
```

7. After executed the code below what are register values in EAX ; EBX ; EDX ; CF, complete the table below.

(1)

```
mov eax, 123h
mov ebx, 100h
mov edx, 0
mul bx
```

AX		BX		DX	
AH	AL	BH	BL	DX	CF
23h	00h	01h	00h	0001h	1

(2)

```
mov dx, 0
mov ax, 1206h
mov bx, 100h
div bx
```

AX		BX		DX	
AH	AL	BH	BL	DX	CF
00h	12h	01h	00h	0006h	0

9.

(1) At the beginning, we want the code below can loop inner loop 20times and outer loop 100times to make ax=2005, but we forget to add the line to restore outer loop count. So what will we get when running the code without restoring outer loop count, please explain what happened? (5%)

We will get ax=100 because the code will only run the inner loop. When cx decreases to 0, it will pass through both loop L2 and loop L1

```
.386
.model flat
Teacher Macro Name
    Local L1, L2
    IFNB <Name>
L1  byte "Who?"
    EXITM
    ELSE
L2  byte "Teacher &Name."
    ENDIF
    EndM
.data
    Teacher Huang
    Teacher
.code
    _start PROC
    _start ENDP

    END _start
```

Hint: The two macro calls will be expanded as:

```
??0001 byte "Teacher Huang."
??0002 byte "Who?"
```

(2) Trying to fix the code with your knowledge learned from AL class and filling the right code in the blank. (5%)

```
.data
count DWORD ?
.code
    mov ax, 5
    mov ecx, 100            ;set outer loop count
L1:
    mov count, ecx          ;save outer loop count
    mov ecx, 20             ;set inner loop count
L2:
    inc ax
    loop L2
    mov ecx, count          ;restore outer loop count
    loop L1
```

11. Complete the following code: (10%)

```
mWriteChar MACRO char
    IFB <char>.             ; check parameter is empty or not
        EXITM
    ENDIF

    push eax                ; preserve the value of register
                            ; whose value will be changed.
    mov al, char
    call "WriteChar"        ; to print out char

    pop edx                 ; restore register
ENDM

main PROC
    mov b1, 'A'
    mWrite.Char b1          ; invoke MACR O "mWriteChar"
                            ; with argument bl
    exit
```

1. Translate the 8-bit binary (in 2's complement format) 1111 1001 to decimal: -7
2. In the following data definition, assume that List begins at offset 100Ah.
   What is the offset of the third value (5)? _100E_h

```
List WORD 3, 4, 5, 6, 7
```

3. Complete the instruction that moves the higher 8 bits (11h) of Var1 into the AH register (hint: use PTR).

```
Var1  DWORD  11223344h
mov ah,  Var1          BYTE PTR [Var1+3]
```

4. What is the value of AX when the code has executed? 22____h

```
bytes BYTE 0h,11h,22h,33h
mov ebx,OFFSET bytes
mov ax,word PTR [ebx+2]
```

5. What will be the value of EBX when the code has executed? _____10

```
mov eax,5
push eax
mov eax,10
push eax
pop ebx
pop eax
```

6. Consider the instruction: push eax
   Before executing the instruction, esp: 0000ff04h
   After executing it, esp: 0000ff00____h.
7. What will be the final value of ESI when the following code has executed? ____6h

```
.data
array  word 8,2,3          2×3=6
.code
       mov esi,0
       mov ecx, LENGTHOF array
L1:
       add esi,TYPE array
       loop L1
```

8. What will be the final value of EAX when the following code has executed? __10110__b

```
       mov eax,10110b
       xor eax,10101b
       jz L1               ;jmp if zero flag set …
       xor eax,10101b
L1:
```

Complete the following block of instructions which multiply the value in EAX register by 20? (Do not worry about overflow.)

```
mov ebx,eax
shl eax,4        ;shift left …      2^4 + 2^2 = 16+4 = 20
shl ebx,__2
add eax,ebx
```

10. What will be the value of AX after the following code has executed? FFFF h

```
    mov  ax,1
    shr  ax,1
    mov  ax,0FFFFh
    rcr  ax,6          ;rotate with carry right …
    jc   L1            ;jmp if carry …
    xor  ax, 0FFFFh
L1: or   ax,ax
```

11. Complete the instruction in the following code inside the procedure to reserve space on the stack for three word local variables?

```
ArraySum PROC
    push ebp
    mov  ebp,esp
    add  ebp , -6      ; reserve space for local variables
    ...
    ...
ArraySum ENDP
```

12. Complete the following code.

```
[1]   TITLE Calculating a Factorial                (Fact.asm)
[2]   ; This program uses the recursion( n! = n*(n-1)!; 0!=1) to calculate 6!
[3]   ; Note: we ignore edx when using MUL instruction.
[4]   .386
[5]   .model flat
[6]   ExitProcess PROTO stdcall, dwExitCode:DWORD
[7]   .stack 4096
[8]   .code
[9]   _start PROC
[10]        push 6                    ; calculate 6 factorial
[11]        call Factorial            ; calculate factorial
[12]        call ExitProcess(0)
[13]  _start ENDP
[14]
[15]  Factorial PROC
[16]        push ebp                  ; save caller's stack frame base
[17]        mov  ebp , esp            ; establish the current stack frame base
[18]        mov  eax,[ebp+8]          ; get input
[19]        cmp eax,0                 ; n > 0?
[20]        ja  L1                    ; yes: continue
[21]        mov eax,1                 ; no: return 1
[22]        jmp L2
[23]
[24]  L1: dec  eax
[25]        push eax                  ; Factorial(n-1)
[26]        call Factorial
[27]
[28]
[29]  ; ReturnFact
[30]        mov ebx,[ebp+8]           ; get n
[31]        mul  ebx                  ; return eax = n * (n-1)!
[32]  ; ignore edx
[33]  L2: pop ebp                     ; restore the caller's stack frame base
[34]        ret 4                     ; ret and clean up stack
[35]  Factorial ENDP
[36]
[37]  END _start
```

13. Complete the following code.

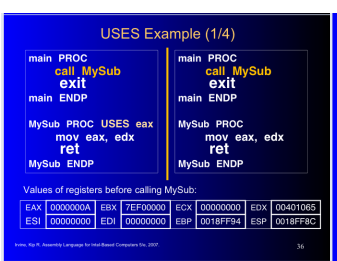
```
[1]   .386
[2]   .model flat
[3]   MyName Macro Name
[4]       Local      msg1, msg2
[5]       IFB <Name>
[6]   Msg1 byte "I am nobody."
[7]       EXITM
[8]       ELSE
[9]   Msg2 byte "My name is &Name
[10]      ENDIF
[11]  EndM
[12]  .data
[13]      MyName Huang
[14]      MyName
[15]  .code
[16]  _start PROC
[17]
[18]  _start ENDP
[19]  END _start
```

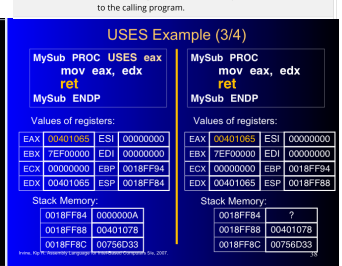
Hint- The two macro calls will be expanded as

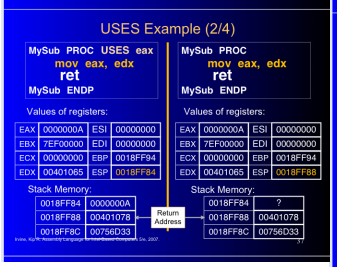
```
??0001 byte "My name is Huang."
??0002 byte "I am nobody."
```

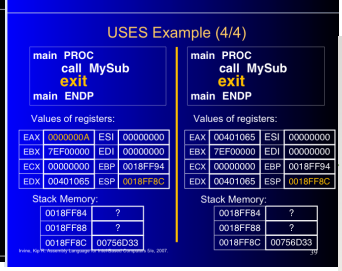
Differences between Procedures and Macros

Characteristic	Procedure	Macro
Number of Instructions that can be effectively handled by the microprocessor	It is better to use Procedures for a set of a large number of instructions. Hence, it is optimal to use Procedures when the number of instructions is more than 10.	Macros are useful over Procedures when the number of instructions in the set is less. Therefore, when the subprogram contains less than 10 instructions, Macros are more efficient to use in such cases.
Assembler Directives used	The assembler directive - PROC is used to define a Procedure. And the assembler directive - ENDP is used to indicate that the body of the procedure has ended.	The assembler directive- MACRO is used to define a Macro, And to indicate that the body of the procedure has ended, the assembler directive- ENDM is used.
Execution Process	Every time a procedure is called, the CALL and RET instructions are required for shifting the control of instruction execution.	Every time a Macro is called, the assembler of the microprocessor places the entire set of instructions of the Macros in the mainline program form where the call to the macro is made.
Execution Time	The Procedures execute slower than the Macros because every time a procedure is called, it is necessary to integrate and link it with the calling program and this takes time.	The execution of macros is faster as compared to procedures because there is no need to integrate or link the macros with the calling program. It is simply loaded into the main memory every time it is called.
Overhead time	Overhead time occurs while calling the procedure and returning the control to the calling program.	Overhead time is avoided as calling and returning does not take place.









1. Translate decimal -10 to the 8-bit binary (in 2's complement format) : 11110110.
2. In the following data definition, assume that List begins at offset 101Ah. What is the offset of the second value (4)? 101C h.

```
List WORD 3, 4, 5, 6, 7
```

3. What is the value of ax when the code has executed?: FF8F h

```
mov bl,8Fh
movsx ax,bl
```

MOVSX
An operand is sign-extended by taking the smaller operand's highest bit and repeating (replicating) the bit throughout the extended bits in the destination operand. The following example

4. What is the value of AL when the code has executed? 1 h

```
word1 DWORD  0,1,2,3
mov ebx,OFFSET word1
mov al,byte ptr [ebx+4]
```

5. What will be the value of EAX when the code has executed? 6

```
mov ebx,3
push ebx
mov eax,6
push eax
pop eax
pop ebx
```

6. Consider the instruction: push eax
   Before executing the instruction, esp: 00FF0010h
   After executing it, esp: 00FF000Ch h.
7. What will be the final value of ESI when the following code has executed? 6

```
.data
array word 8,2,3
.code
    mov esi,0
    mov ecx, LENGTHOF array
L1:
    add esi, TYPE array
    loop L1
```

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:

2+2+2=6

8. What will be the final value of EAX when the following code has executed? 00FF00FF h.

```
mov eax,00FF00FFh
xor eax,0F0F0F0Fh
xor eax,0F0F0F0Fh
```

9. Complete the following block of instructions which multiply the value in EAX register by 6? (Do not worry about overflow.)

```
mov ebx,eax
shl eax,1          ;shift left…
shl ebx,  2
add eax,ebx
```

10. What will be the value of AX after the following code has executed? F000 h.

```
    mov ax,0FFF0h
    ror ax,4       ;rotate right …
    jc L1          ;jmp if carry …
```

0FFF 0FFF FFOF FFFO

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:


```
    xor ax, 0FFFFh
L1: or ax,ax
```

11. Complete the instruction in the following code inside the procedure to reserve space on the stack for two doubleword local variables?

```
ArraySum PROC
push ebp
mov ebp,esp
add  esp , -8      ; reserve space for local variables
.....
ArraySum ENDP
```

2x4=8

12. Complete the following code.

```
[1]   TITLE Calculating a sum        (Sum.asm)
[2]   ; This program uses the recursion(Sum(n)=n+Sum(n-1); Sum(1)=1) to
      calculate Sum(6).
[3]   .386
[4]   .model flat
[5]   ExitProcess PROTO stdcall, dwExitCode:DWORD
[6]   .stack 4096
[7]   .code
[8]   _start  PROC
[9]         push 6                          ; calculate Sum(6)
[10]        call Sum                        ; calculate sum in eax
[11]        call ExitProcess(0)
[12]  _start ENDP
[13]
[14]  Sum PROC
[15]        push ebp                        ;save caller's stack frame base
[16]        mov  ebp , esp                  ;establish the current stack frame
      base
[17]        mov eax, [ebp+8]                ; get input
[18]        cmp eax, 1                      ; n > 1?
[19]        ja L1                           ;yes: continue
[20]        mov eax, 1                      ;no: return
[21]        jmp L2
[22]
[23]  L1: dec eax                           ;calculate Sum(n-1)
[24]        push eax
[25]        call Sum
[26]
[27]  ; ReturnSum
[28]        mov edx,[ ebp+8 ]               ; get n
[29]        add eax , edx                   ; return Sum(n)=n+Sum(n-1)
[30]
[31]  L2:  pop  ebp                         ;restore the caller's stack frame base
[32]        ret 4                           ; ret and clean up stack
[33]  Sum ENDP
[34]
[35]  END _start
```

Q2. What are the values of the registers after the code is executed?

```
mov eax, 00011001h
mov ebx, 00000110h
mov ecx, 01010000h
push eax
push ebx
push ecx
pop eax
pop ecx
pop bx
```

eax	0001 1001h
ebx	0000 0110h
ecx	0101 0000h

Register

0101 0000h
0000 0110h
0001 1001h

Stack

1001h
0001h

eax	0101 0000 h
ebx	0000 1001 h
ecx	0000 0110 h

Q1. Translate hexadecimal 9Dh to the binary(in **2's complement format**) and Decimal

Hexadecimal	9D
8-bit binary	1001 1101
Decimal	-99

符號								
0	1	1	1	1	1	1	1	= 127
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= -1
1	1	1	1	1	1	1	0	= -2
1	0	0	0	0	0	0	1	= -127
1	0	0	0	0	0	0	0	= -128

Q3. What are the values of the registers after the code is executed?

```
mov   ax, 20h
mov   bx, 139h
mulbx
shl  bx, 5
```

ax	27 20 h
bx	27 20 h
dx	00 00 h
CF	0

0139 h
x 0020 h
2720 h

0000 0001 0011 1001

0010 0111 0010 0000

Q4. Carry Flag, Zero Flag, Sign Flag and Overflow Flag

```
.data
  Val1 WORD 20 DUP(?)
.code
  mov ax, 6Ch
  add al, LENGTHOF Val1
  mov al, 0d8h
  add al, SIEZEOF Val1
```

ax 00 6C h

0110 1100
+ 0001 0100
1000 0000

1101 1000
+ 0010 1000
1 0000 0000

OF	SF	ZF	CF
1	1	0	0
1	1	0	0
0	0	1	1

Q5. The table shows the values saved in the registers before the code is executed.

```
UseStack PROC USES ax
    pop  cx
    push bx
    L2:
    ret
UseStack ENDP
main PROC
    mov ax, Val1
    mov bx, 1234h
    call UseStack
  L1:
    exit
main ENDP
```

ax	12 34 h
bx	12 34 h
cx	00 9D h
esp	0019ff84 h

Register

0019ff7eh	12 34 h
0019ff80h	L1
0019ff84h	75e13c20h

Stack

Q6. What are the values of the registers after the code is executed.

ESI

```
.data
    myData byte 1, 4, 8, 16, 64
.code
    mov  eax, 0
    mov  esi, OFFSET myData
    mov  ecx, LENGTHOF myData
L1:
    add  al, [esi]
    inc  esi
    loop L1
```

eax	0000 0000 h	eax	0000 000d h
ecx	0000 0005 h	ecx	0000 0002 h
eax	0000 0001 h	eax	0000 001d h
ecx	0000 0004 h	ecx	0000 0001 h
eax	0000 0005 h	eax	0000 005d h
ecx	0000 0003 h	ecx	0000 0000 h

Q7. According to the following program.

```
    mov  eax, 3Ch
    test eax, 195
L1:
    jz   L2
    or eax, 7Fh
    exit
L2:
    xor eax, eax
```

eax 0000 003C h

0011 1100
& 1100 0011
0000 0000 b

0011 1100
xor 0011 1100
0000 0000 b

ZF = 1

eax 0000 003C h | eax 0000 0000 h

Q8. Fill in the blank to finish the code, and the value of myData is ___48___ (d) after the code is executed.

```
.data
        myData DWORD 12
.code
Quadruple  MACRO data
    IFB <__data__>                ;If the parameter is empty, exit macro
        EXITM
    ENDIF
    mov eax, [data]
    shl eax, __2                  ; Make the value become quadruple
    mov [data], eax
ENDM

main PROC
    mov esi, OFFSET myData
    Quadruple  __esi              ; MACRO Call: Quadruple, with parameter esi
    exit
main ENDP
```

Q9. This is a assembly language code, please answer the following questions.

```
.code
    mov ax, 42
    mov bx, 27
L1:
    mov dx, 0
    div bx
    cmp dx,0
    je L2
    mov ax, bx
    mov bx, dx
    jmp L1
L2:
    mov ax, bx
    exit
```

27 | 42 (1), 27, 15

15 | 27 (1), 15, 12

12 | 15 (1), 12, 3

3 | 12 (4), 12, 0

ax	bx	dx
002A h	001B h	1005 h
001B h	000F h	000F h
000F h	000C h	000C h
000C h	0003 h	0003 h
		0000 h

Assembly Language – Procedures (2)

5-2 Defining and Using Procedures

Objective: Understanding the use of Procedure and its relevant directives.

Procedure "Convert" This Procedure will store an uppercase letter in ESI. After the invocation, register values will be the same as before.

```
.data
ChStr1 BYTE 'A'41h, 'B', 'C'
ChStr2 BYTE 3 dup(?
.code
Convert PROC USES ax
        mov al, [esi]
        add al, 20h
        mov [edi], al
L2:     ret
Convert ENDP

main PROC
        mov esi, OFFSET ChStr1
        mov edi, OFFSET ChStr2
L1:     call Convert
L3:     exit
main ENDP
END main
```

a. Suppose the values in the registers at L1 are:

EAX	75e13c33h	EBP	0012ff94h
EBX	7ffdf000h	ESP	0012ff8ch
ECX	00000005h	ESI	00404000h
EDX	00401000h	EDI	00404003h

At L2 position, what are register values?

EAX	75e13c61h	EBP	0012ff94h3
EBX	7ffdf000h	ESP	0012ff86h
ECX	00000005h	ESI	00404000h
EDX	00401000h	EDI	00404003h

b. Following Problem a, the stack data are:

0012ff86h	3c33h
0012ff88h	L3
0012ff8ch	75e13c20h

c. What are the register values when the program stops at L3?

EAX	75e13c33h	EBP	0012ff94h
EBX	7ffdf000h	ESP	0012ff86h
ECX	00000005h	ESI	00404000h
EDX	00401000h	EDI	00404003h

Symbol	hex	Symbol	Hex
A	41h	a	61h

4-1 Integer Arithmetic, Data Transfer Instructions

Objective : Familiar with the instructions MOV and SHIFT.

a. The following codes intend to set Rval = 19*Val1. Please complete the codes.

```
.data
Val1    SBYTE   -1          ; Val1 = 0ffh
Rval    SWORD   ?
.code
main PROC
L1:
movsx ax, Val1    ;BaseValue
movzx bx, Val1
mov cl, Val1
mov ch, 10h
L2:
    mov dx,  0                ; 1. dx = 0
    add dx,  ax               ;2. dx=dx+ax
    shl ax,  ____1____        ;3. ax*2
    add dx,  ax               ;4.dx=2Val1+Val1=3Val1
    shl ax,  ____3____        ;5. ax*8, ax = 16Val1
    add dx,  ax               ;6. dx=19* Val1
L3:     exit
main ENDP
END main
```

	AX	DX
1	0ffffh	0000h
2	0ffffh	0ffffh
3	0fffeh	0ffffh
4	0fffeh	0fffdh
5	0fff0h	0fffdh
6	0fff0h	0ffedh

b. Based the code above, when is not any command executed at L1 position, registers values will be as follows:

EAX	00000000h	EBP	0012fb50h
EBX	00000000h	ESP	0012fb24h
ECX	0012fb08h	ESI	ffffffeh
EDX	772070b4h	EDI	00000000h

When all the commands have been executed before L3 position, what is the register's values?

EAX	0000fff0h	EBP	0012fb50h
EBX	000000ffh	ESP	0012fb24h
ECX	0012fb10h	ESI	ffffffeh
EDX	77120ffedh	EDI	00000000h

PUSH Instruction

The PUSH instruction first decrements ESP and then copies a source operand into the stack. A 16-bit operand causes ESP to be decremented by 2. A 32-bit operand causes ESP to be decremented by 4. There are three instruction formats:

```
PUSH reg/mem16
PUSH reg/mem32
PUSH imm32
```

POP Instruction

The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

7-1 Shift and Rotate Instructions, Multiplication and Division Instructions
Objective: Understanding the rotate instructions, "MUL" instruction, and "IMUL" instruction.

(1)

```
.data
myArray BYTE 33, 3, 17
myArray_v2 WORD 256
.code
main PROC
   movzx eax, myArray[0]
   movzx ebx, myArray[1]
```

a. Based on code in the left, What are the register values at L1?

the registers values at L1 are:

EAX	00000021h	EBP	0018ff94h
EBX	00000003h	ESP	0018ff8ch
ECX	00000000h	ESI	00000000h
EDX	00400100h	EDI	00000000h

b. What are the register values when the program run each steps at L1

	BX		BL(bit)	CF
	BH	BL		
L1:	00h	03h	00000011b	00
ror bl, 3	00h	60h	01100000b	0
or bl, 110b	00h	66h	01100110b	0
ror bl, 2	00h	99h	10011001b	1
mov bl, 10h	00h	10h	00010000b	1

c. What are the register values when the program run each steps at L2?

	DX	AX		BX		CF
		AH	AL	BH	BL	
L2:	0100h	00h	21h	00h	10h	0
mul bl ;ax=Product	0100h	02h	10h	00h	10h	1
mov bx,myArray_v2[0]	0100h	02h	10h	01h	00h	1
mul bx ;dx:ax=Product	0002h	10h	00h	01h	00h	1

```
L3:
   exit
main ENDP
END main
```

256 = 16

• The Carry flag is set when the result of an unsigned arithmetic operation is too large for the destination operand.

(2)

```
.code
main PROC
mov al, 19h
mov bl, 08h
imul bl
main ENDP
END main
```

What are the register values at each position?

AX		BL	OF
AH	AL		
00h	19h	00h	0
00h	19h	08h	0
00h	98h	08h	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

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

What Are Assemblers and Linkers? An *assembler* is a utility program that converts source code programs from assembly language into machine language. A *linker* is a utility program that combines individual files created by an assembler into a single executable program. A related utility, called a *debugger*, lets you to step through a program while it's running and examine registers and memory.

How Does Assembly Language Relate to Machine Language? *Machine language* is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language. *Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* relationship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

1.3.3 Integer Storage Sizes

The basic storage unit for all data in an x86 computer is a *byte*, containing 8 bits. Other storage sizes are *word* (2 bytes), *doubleword* (4 bytes), and *quadword* (8 bytes). In the following figure, the number of bits is shown for each size:

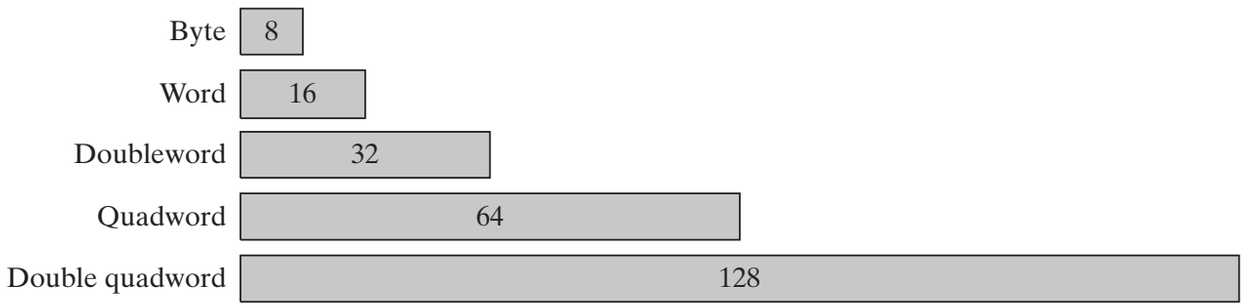


Table 1-4 shows the range of possible values for each type of unsigned integer.

Large Measurements A number of large measurements are used when referring to both memory and disk space:

- One *kilobyte* is equal to 2^{10}, or 1024 bytes.
- One *megabyte* (1 MByte) is equal to 2^{20}, or 1,048,576 bytes.
- One *gigabyte* (1 GByte) is equal to 2^{30}, or 1024^3, or 1,073,741,824 bytes.
- One *terabyte* (1 TByte) is equal to 2^{40}, or 1024^4, or 1,099,511,627,776 bytes.
- One *petabyte* is equal to 2^{50}, or 1,125,899,906,842,624 bytes.
- One *exabyte* is equal to 2^{60}, or 1,152,921,504,606,846,976 bytes.
- One *zettabyte* is equal to 2^{70} bytes.
- One *yottabyte* is equal to 2^{80} bytes.

Table 1-4 Ranges and Sizes of Unsigned Integer Types.

Type	Range	Storage Size in Bits
Unsigned byte	0 to $2^8 - 1$	8
Unsigned word	0 to $2^{16} - 1$	16
Unsigned doubleword	0 to $2^{32} - 1$	32
Unsigned quadword	0 to $2^{64} - 1$	64
Unsigned double quadword	0 to $2^{128} - 1$	128

1.3.4 Hexadecimal Integers

Large binary numbers are cumbersome to read, so hexadecimal digits offer a convenient way to represent binary data. Each digit in a hexadecimal integer represents four binary bits, and two hexadecimal digits together represent a byte. A single hexadecimal digit represents decimal 0 to 15, so letters A to F represent decimal values in the range 10 through 15. Table 1-5 shows how each sequence of four binary bits translates into a decimal or hexadecimal value.

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

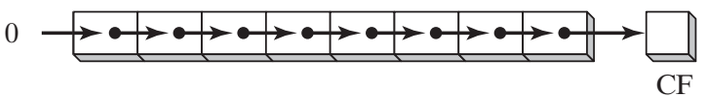
Along with bitwise instructions introduced in Chapter 6, shift instructions are among the most characteristic of assembly language. *Bit shifting* means to move bits right and left inside an operand. x86 processors provide a particularly rich set of instructions in this area (Table 7-1), all affecting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

Table 7-1 Shift and Rotate Instructions.

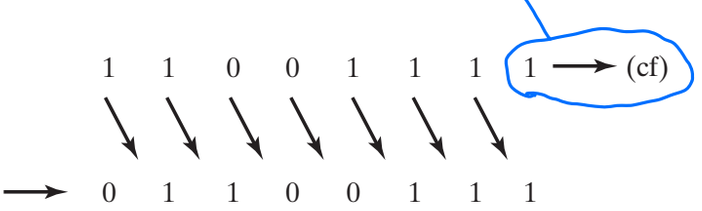
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

7.1.1 Logical Shifts and Arithmetic Shifts

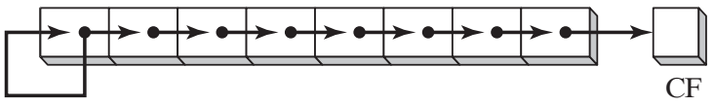
There are two ways to shift an operand's bits. The first, *logical shift*, fills the newly created bit position with zero. In the following illustration, a byte is logically shifted one position to the right. In other words, each bit is moved to the next lowest bit position. Note that bit 7 is assigned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



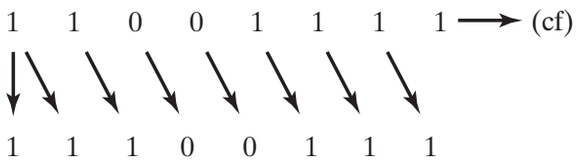
The following illustration shows a single logical right shift on the binary value 11001111, producing 01100111. The lowest bit is shifted into the Carry flag:



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:

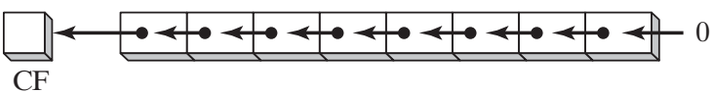


Binary 11001111, for example, has a 1 in the sign bit. When shifted arithmetically 1 bit to the right, it becomes 11100111:

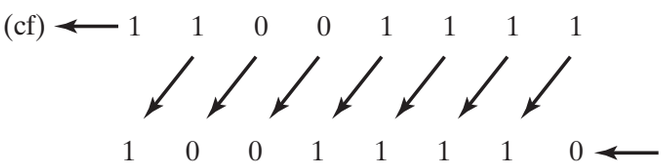


7.1.2 SHL Instruction

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 $5 \times 2^1 = 10$:

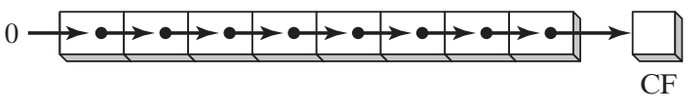
```
mov  dl,5        Before:  0 0 0 0 0 1 0 1  = 5
shl  dl,1        After:   0 0 0 0 1 0 1 0  = 10
```

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
```

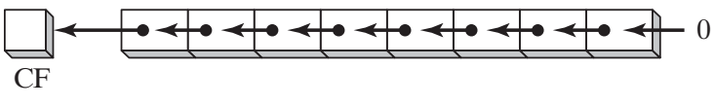
In the following example, 64 is divided by 2^3:

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

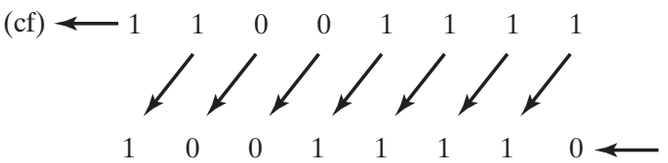
Division of signed numbers by shifting is accomplished using the SAR instruction because it preserves the number's sign bit.

7.1.4 SAL and SAR Instructions

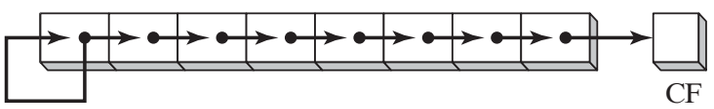
The SAL (shift arithmetic left) instruction works the same as the SHL instruction. For each shift count, SAL shifts each bit in the destination operand to the next highest bit position. The lowest bit is assigned 0. The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is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

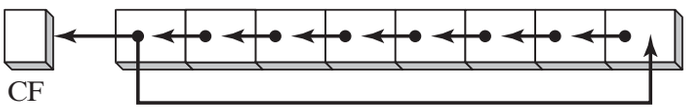
Sign-Extend AX into EAX Suppose AX contains a signed integer and you want to extend its sign into EAX. First shift EAX 16 bits to the left, then shift it arithmetically 16 bits to the right:

```
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:

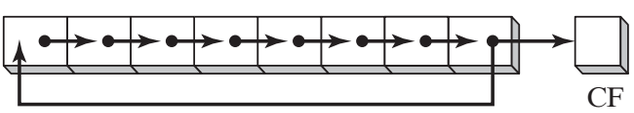
```
mov  al,26h
rol  al,4                      ; AL = 62h
```

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:

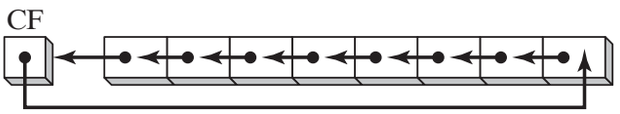
```
mov  al,01h                    ; AL = 00000001b
ror  al,1                      ; AL = 10000000b, CF = 1
ror  al,1                      ; AL = 01000000b, CF = 0
```

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:



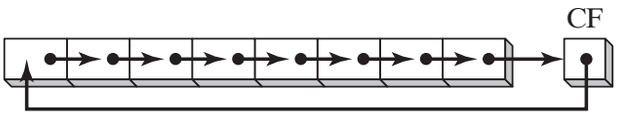
If we imagine the Carry flag as an extra bit added to the high end of the operand, RCL looks like a rotate left operation. In the following example, the CLC instruction clears the Carry flag. The first RCL instruction moves the high bit of BL into the Carry flag and shifts the other bits left. The second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

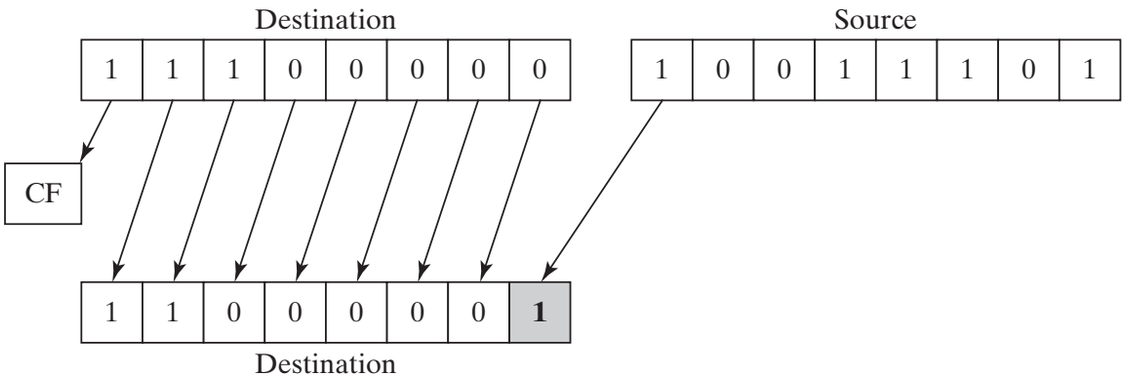
The value of the Overflow flag is undefined when the shift or rotation count is greater than 1.

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
```

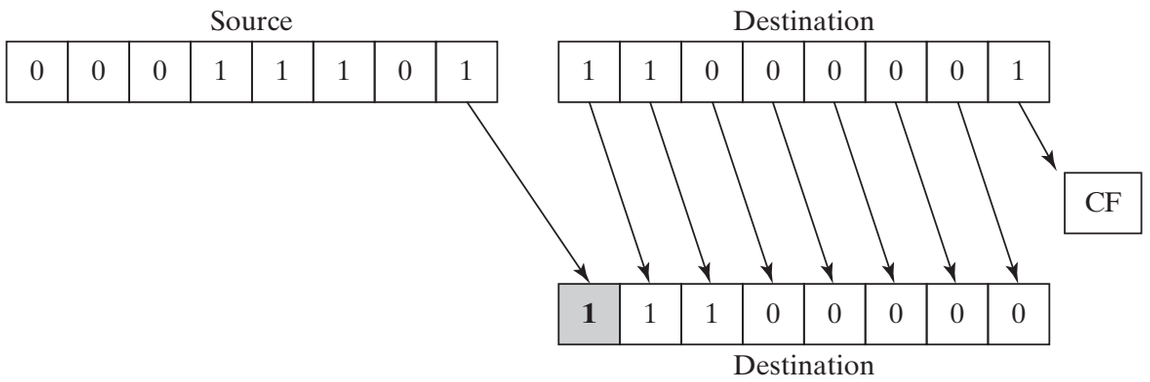
The following illustration shows the execution of SHLD with a shift count of 1. The highest bit of the source operand is copied into the lowest bit of the destination operand. All the destination operand bits are shifted left:



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:



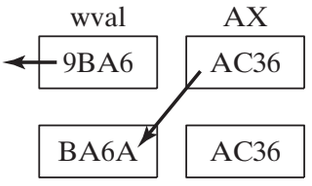
The following instruction formats apply to both SHLD and SHRD. The destination operand can be a register or memory operand, and the source operand must be a register. The count operand can be the CL register or an 8-bit immediate operand:

```
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
```

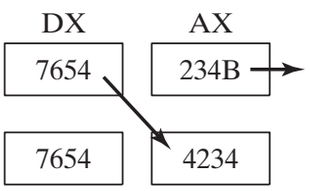
The data movement is shown in the following figure:



9BA6 shld BA6A

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
```



shrd
234B → 4234

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

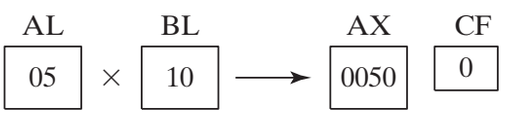
A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.

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:

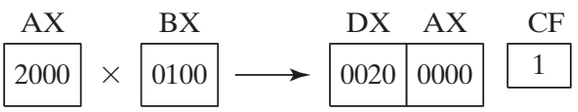
```
mov  al,5h
mov  bl,10h
mul  bl                          ; AX = 0050h, CF = 0
```

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:

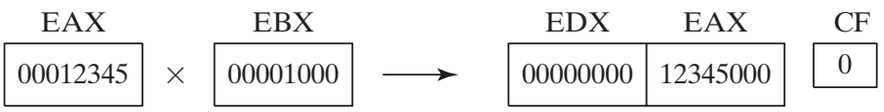
```
.data
val1  WORD  2000h
val2  WORD  0100h
.code
mov  ax,val1                     ; AX = 2000h
mul  val2                        ; DX:AX = 00200000h, CF = 1
```



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
```