

# Chapter 6

## Data Manipulation Instructions

### 6.1 Introduction

Data manipulation instructions transform data via an arithmetic, a shifting, or a boolean operation. This chapter deals with arithmetic instructions, shifting and rotation instructions, and boolean instructions for bit-level manipulation.

### 6.2 Arithmetic Instructions

#### 6.2.1 Increment and Decrement

These are single-operand instructions. The operation consists on incrementing, decrementing the operand by 1. The carry flag (CF) is not affected, however the remaining appropriate flags are set according to the result. The typical format of these instructions calls for a single explicit operand which specifies either a register or a memory reference.

$$\begin{array}{ll} \text{inc } r/m & ; r/m \leftarrow r/m + 1 \\ \text{dec } r/m & ; r/m \leftarrow r/m - 1 \end{array}$$

The  $r$  operand specifies any of either an 8-bit, a 16-bit, or a 32-bit register. The  $m$  operand specifies a byte, or a word, or a double word size operand. Such operand can be accessed directly or indirectly. For example, the instruction *inc*

*word [bx]* will increment the word-size contents at the location specified in *bx*. In a direct addressing mode the address is specified in the instruction; for example *dec byte [count]* will decrement the byte at the address associated with the label *count*.

### 6.2.2 Negate

The *negate* instruction returns the two's complement of the original contents of its single explicit operand that specifies a register or a memory reference. The format and the operation performed are the following:

$$\text{neg } r/m \quad ; r/m \leftarrow \overline{r/m} + 1$$

The *r* operand specifies the name of an 8-bit, a 16-bit, or a 32-bit operand. Again if the name is surrounded by square brackets it specifies an indirect fetch of the operand from memory. Direct memory references use the name of variable surrounded by square brackets. Recall that *nasm* does not remember the types of variables, so a type qualifier *byte*, *word*, or *dword* is used in front of the memory reference to specify the size of the operand to fetch. For example the execution of *neg dword [sum]* will result in the two's complement of the value stored at *sum*.

### 6.2.3 Addition and Subtraction

The *add* and *sub* instructions perform integer addition and subtraction, respectively. The contents of the two explicit operands specified in the instruction are added or subtracted and the result replaces the original contents of the destination operand. The format and operation of these instructions are the following:

$$\begin{aligned} \text{add } r/m, r/m/k & \quad ; r/m \leftarrow r/m + r/m/k \\ \text{sub } r/m, r/m/k & \quad ; r/m \leftarrow r/m - r/m/k \end{aligned}$$

Again the registers (*r*) or memory locations (*m*) specified contain 8-bit, 16-bit or 32-bit operands. For memory reference operands the appropriate qualifier will ensure that both operands are the same size. While only one operand can be fetched from memory, the other can be a constant (*k*) or a register. If the two operands require a memory reference then an extra instruction is needed to load one into a register first. The zero (*ZF*) and the sign (*SF*) flags are set according to the results of the operation. For example the execution of the following sequence of instructions:

```

mov ax,1
sub ax, 2

```

will result in a value *FFFFh* that represents a -1 stored in *ax*; therefore, the sign flag  $SF = 1$ . The overflow (*OF*) flag is checked in the context of signed operations. The result of an addition or a subtract operation may lead to an *overflow*, if the result is too big or too small (*underflow*) to be represented with the available number of bits.

*Example:* The addition of  $42 + 87 = 129$ . If 8 bits are available to hold the result then the fact that  $129 > 2^7 - 1 = 127$  indicates an *overflow*. The use of a binary representation is useful to illustrate the detection of overflow:

$$\begin{array}{r}
 00101010 \\
 + 01010111 \\
 \hline
 10000001 = -(127)
 \end{array}$$

Consider also the addition of negative numbers:  $(-42 - 87 = -129)$ . Note again that  $-129 < -2^7 = -128$  which indicates the occurrence of an *overflow*. In binary format:

$$\begin{array}{r}
 11010110 \\
 + 10101001 \\
 \hline
 01111111 = +(127)
 \end{array}$$

Observe that when adding two numbers with the same sign, overflow occurs if the sign of the result is different. While this is useful in paper and pencil a practical detection of overflow avoids the generation of the result altogether.

The instructions *add* and *sub* will also set the carry flag if an overflow occurs. Unlike the overflow flag, the carry flag is checked only for unsigned operations. Consider for example the following lines of code:

```

mov ax, 00FFh
add al, 1

```

The *add* operation will set the carry flag  $CF = 1$ ; the result in *ax* is zero because the unsigned 8-bit addition is performed only on the *al* register which contains the maximum 8-bit value before the *add* instruction is executed. However, the following code:

```

mov ax, 00FFh
add ax, 1

```

will carry out a 16-bit addition with a result in *ax* of 0100h with no overflow and the carry flag remains equal to zero.

### 6.2.4 Multiplication and Division

Multiplication and division instructions take a single operand to specify a multiplier or a divisor. Prior to the multiplication/division instruction, an additional implicit operand must be preloaded into the appropriate registers. The instruction set provides two different formats, signed and unsigned for multiplication and division:

Unsigned operations:

*mul reg/mem* ;  $product \leftarrow r/m \times al/ax/eax$   
*div reg/mem* ;  $remainder : quotient \leftarrow al/ax/eax \div r/m$

Signed operations:

*imul reg/mem* ;  $product \leftarrow reg/mem \times al/ax/eax$   
*idiv reg/mem* ;  $remainder : quotient \leftarrow al/ax/eax \div reg/mem$

A  $n$ -bit multiplication involves operands with  $n$  bits and will generate a product that requires  $2n$  bits. An  $n$ -bit division requires a  $2n$  dividend and the resulting quotient is an  $n$ -bit value. The execution of a division instruction generates a remainder and a quotient as indicated in the comment field. Table 1.1 illustrates which registers are used to preload implicit operands and which registers are assigned to hold results for both multiplication and division instructions.

Table 6.1: Multiplication and Division

Multiplication	Multiplicand	Multiplier	Product	
Byte	al	r/m	ax	
Word	ax	r/m	dx(high):ax(low)	
Dword	eax	r/m	edx(high):eax(low)	
Division	Dividend	Divisor	Quotient	Remainder
Byte	ax	r/m	al	ah
Word	dx:ax	r/m	ax	dx
Dword	edx:ax	r/m	eax	edx

**Sign Extension.** As shown in table 1.1, a multiplier and a multiplicand must be the same size, 8, 16, or 32 bits, then the product requires 16, 32, or 64 bits. Observe

that in the case of division, the dividend is 16, 32, or 64 bits wide and the quotient and remainder require 8, 16, or 32 bits. If the flow of computation renders and  $n$ -bit dividend, then for sign division operations it must be *sign-extended* to  $2n$  bits. In the following sign-extensions instructions the notation  $s_x$  indicates the sign bit repeated  $x$  times for the only purpose of describing the operation using a register transfer notation:

```
cbw    ;  $ax \leftarrow s_8 : al$ 
        ; converts the signed byte in  $al$  to a word in  $ax$ 
cwd    ;  $dx : ax \leftarrow s_{16} : ax$ 
        ; converts the signed word in  $ax$  to a double word in  $dx:ax$ 
cwde   ;  $eax \leftarrow s_{16} : ax$ 
        ; converts the signed word in  $ax$  into  $eax$ 
cdq    ;  $edx : eax \leftarrow s_{32} : eax$ 
        ; converts the signed double word in  $eax$  into  $edx:eax$ 
```

*Example 1:* The following code implements an 8-bit signed multiplication with a signed 16-bit product:

```
mov al, -4    ;  $al = 1111\ 1100$ 
mov bl, 8
imul bl       ;  $ax = 1111\ 1111\ 1110\ 0000 = -32$ 
```

*Example 2:* The following code illustrates the implementation of a 16-bit multiplication with a 32-bit product:

```
mov ax, 2000h ;  $ax = 0010\ 0000\ 0000\ 0000$ 
mov bx, 0025h
mul bx        ;  $dx:ax = 0000\ 0000\ 0000\ 0100:1010\ 0000\ 0000\ 0000$ 
               ;  $= 0004h:A000h$ 
```

*Example 3:* Illustration of a 32-bit multiplication requiring a 64-bit product:

```
mov eax, 12345678h
mov ebx, 100000h
mul ebx       ;  $edx:eax = 0001\ 2345h:6780\ 0000h$ 
```

*Example 4:* Illustration of an 8-bit signed division. Suppose  $al = -48$  then a sign-extension is required before the division is performed:

```
cbw          ;  $ax = FFD0h$ 
mov bl, 8
idiv bl      ;  $ax = 0000\ 0000\ 1111\ 1010$ 
               ;  $al = \text{quotient} = -6; ah = \text{remainder} = 0$ 
```

**Divide Overflow.** If the result of a division is too large for the number of bits required by the operation, then a divide overflow occurs. The smaller the divisor is the larger the quotient and the more likely to trigger an overflow. Note that a division by zero triggers an interrupt that hands control over to an exception procedure that displays a “divide overflow” message. When an overflow occurs a typical solution is to break an  $n$ -bit division into two  $\frac{n}{2}$  operations. The next two examples illustrate this procedure.

*Example 1:* This example illustrates the implementation of a 16-bit division where the quotient clearly requires 32 bits.

```

segment data
    dividend    dd    08010020h
    divisor     dw    10h

segment bss
    quotient    resd   1
    remainder   resw   1

segment code
    ...
    ...
    mov ax, word [dividend+2]    ;high part
    cwd                          ;extend sign
    mov cx, word [divisor]
    idiv cx                      ;ax=q (high), dx=r (high)
    mov bx,ax                    ;save q
    mov ax, word [dividend]      ;low part, now dx:ax = dividend
                                ;dx contains the remainder part
    idiv cx                      ;ax=q (low), dx=r (low)
    mov word [quotient], ax      ;save quotient (bx:ax)
    mov word [quotient+2], bx
    mov remainder, dx           ; save remainder
    ...

```

*Example 2:* While overflow is a potential problem for any  $n$ -bit division, the 16-bit implementation in the previous example is contrasted with the following 32-bit code:

```

segment .data
    dividend    dd    08010020h
    divisor     dd    10h

```

```

segment .bss
    quotient    resd    1
    remainder   resd    1

segment .code
    ...
    mov eax, dword [dividend]
    cdq
    mov ecx, dword [divisor]
    idiv ecx
    mov [quotient], eax
    mov [remainder], edx
    ...

```

### 6.2.5 Multiple addition and subtraction

To consider the carry flag set by the previous addition/subtraction instruction, the IA instruction set provides two instructions that basically expand the addition/subtraction operation to one more byte, word or double word. These instructions are *adc* and *sbb* for addition and subtraction with carry. The format and operation of these instructions are as follows:

$$\begin{array}{ll}
 \text{adc} & r/m, r/m/k \quad ; r/m \leftarrow r/m + r/m/k + C \\
 \text{sbb} & r/m, r/m/k \quad ; r/m \leftarrow r/m - r/m/k - C
 \end{array}$$

An illustration of how the carry affects multiple addition is shown in the 32-bit addition in Fig. 1.1.

Hexadecimal	Upper-half addition	Lower-half addition
$  \begin{array}{r}  1007 \text{ B104} \\  6002 \text{ E6F2} \\  \hline  700A \text{ 97F6}  \end{array}  $	$  \begin{array}{r}  0001 \text{ 0000 0000 0111} \\  0110 \text{ 0000 0000 0010} \\  \hline  0 \text{ 0111 0000 0000 1010}  \end{array}  $	$  \begin{array}{r}  1011 \text{ 0001 0000 0100} \\  1110 \text{ 0110 1111 0010} \\  \hline  1 \text{ 1001 0111 1111 0110}  \end{array}  $
	<p>Carry out of MSB</p>	

Figure 6.1: 32-bit addition operation

Likewise, the 32-bit subtraction operation in Fig. 1.2 illustrates the effect of a borrow bit:

Hexadecimal	Upper-half subtraction	Lower-half subtraction
1007 B104 - 6002 E6F2	0001 0000 0000 0111 - 0110 0000 0000 0010	1011 0001 0000 0100 1110 0110 1111 0010
1007 B104 +190E 0 CA12		1011 0001 0000 0100 + 0001 1001 0000 1110 0 1100 1010 0001 0010
Carry=0 then borrow = 1 1007 - 1	0001 0000 0000 0111 - 1	Carry=0 then borrow = 1
1007 +FFFF 1 1006 + 9FFE B005 CA12	0001 0000 0000 0111 + 1111 1111 1111 1111 1 0001 0000 0000 0110 + 1001 1111 1111 1110 1011 0000 0000 0101	
		1100 1010 0001 0010

Figure 6.2: 32-bit subtraction operation

*Example:* Implement  $x = ax + c$  assuming  $a$  and  $c$  are signed word-size variables.

```

segment data
    x    dd    1250
    a    dw    2
    c    dw    1000

segment code
    ...
    mov ax, data
    mov ds, ax
    mov ax, word [x]
    imul word [a]          ;dx:ax = [a]*[x]
    clc                    ;reset carry flag
    add ax, word [c]        ;ax = ax + [c]
    adc dx, 0               ;dx:ax = [a]*[x] + [c] + carry
    mov word [x], ax        ;store result: low part
    mov word [x+2], dx      ;high part
    ...

```



## 6.3 Shift instructions

Shift and rotate instructions allow the programmer to manipulate data at the bit level. A shift operation moves the position of bits of data in memory or in some register. Shifts can be either toward the left (i.e. toward the most significant bits) or toward the right (the least significant bits).

### 6.3.1 Logical shifts

A logical shift is the simplest type of shift. For each bit shifted there is an incoming bit equal to zero as shown in Fig. 1.3. The *shl* and *shr* instructions perform logical left and right shifts respectively. The number of bit positions to shift can either be a constant or can be stored in the *cl* register. The last bit shifted out of the data is stored in the carry flag. The formats and operation description of these shift instructions are as follows:

```

shl r/m, k    ; CF ← r/m(msb - k + 1)
               ; r/m(msb, ..., k) ← r/m(msb - k, ..., 0)
               ; r/m(k - 1, ..., 0) ← 0
shl r/m, cl   ; the shifting factor k is pre-loaded into cl
shr r/m, k    ; CF ← r/m(k - 1)
               ; r/m(msb - k, ..., 0) ← r/m(msb, ..., k)
               ; r/m(msb, ..., msb - k + 1) ← 0
shr r/m, cl   ; the shifting factor k is in cl

```

The source operand *k* refers to an 8-bit *shifting factor*, and can be optionally pre-loaded into *cl*. For the 8086/8088 processors *k* = 1. Note that this option can be used if *k* is known during programming, else *cl* is pre-loaded with a *k* generated during the flow of computation. As before, the comment line describes the operation at the register level. The notation *r/m*(\*) refers to the bit contents of a register or a memory reference; the arguments within parenthesis indicate the bit positions that are accessed during the transfer.

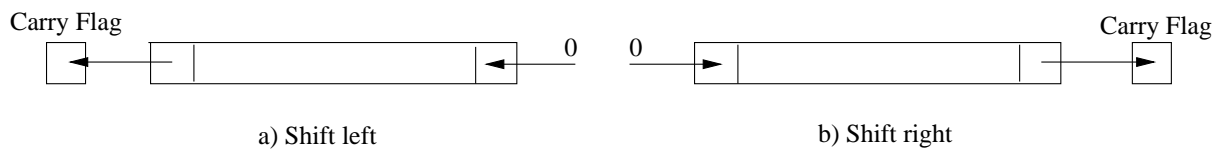


Figure 6.3: Logical shifts

Shift instructions provide fast multiplication and division. As in the decimal system where multiplication and division by a power of ten consists of shifting digits, the same is true for powers of two in binary. For example, to double the value represented by the binary number 01001 (9 in decimal), shift once to the left to get 10010 (18 in decimal). In general  $k$  shifts to the left are equivalent to multiplying data by  $2^k$ . The quotient of a division by a power of two is the result of a right shift. To divide by just 2, use a single right shift; given a shifting factor of  $k$ , a dividend is shifted  $k$  bit positions to the right resulting in a division by  $2^k$ . A typical use of logical shifts is to multiply or divide unsigned values. Shift instructions are very basic and are much faster than the corresponding *mul* and *div* instructions.

### Examples

1. Consider the use of left shift operations such that the operation  $2 \times 2^5$  is performed:

```
mov al, 2    ;al = 0000 0010
mov cl, 5    ;shifting factor
shl al, cl   ;al = 0100 0000 =  $2^6 = 64$ 
```

Note that an 8-bit precision is assumed; however, to anticipate a  $2n$ -bit product, an 8-bit multiplication requires a 16-bit result for which *ax* is used in place of *al*.

2. Consider the operation  $\frac{64}{2^5}$  using right shift operations:

```
mov ax, 64   ;ax = 0000 0000 0100 0000
mov cl, 5    ;shifting factor
shr ax, cl   ;ax = 0000 0000 0000 0010
```

IA 32-bit processors (386 and up) support double precision shifts using the *shld* and *shrd* instructions. The format and register-level and bit operations are described as follows:

```
shld r/m, r, k    ;  $CF \leftarrow r/m(msb - k + 1)$ 
                  ;  $r/m(msb, \dots, k) \leftarrow r/m(msb - k, \dots, 0)$ 
                  ;  $r/m(k - 1, \dots, 0) \leftarrow r(msb, \dots, msb - k + 1)$ 
shld r/m, r, cl   ;  $cl$  is pre-loaded with  $k$ 
shrd r/m, r, k    ;  $CF \leftarrow r/m(k - 1)$ 
                  ;  $r/m(msb - k, \dots, 0) \leftarrow r/m(msb, \dots, msb - k + 1)$ 
                  ;  $r/m(msb, \dots, msb - k + 1) \leftarrow r(k - 1, \dots, 0)$ 
shrd r/m, r, cl   ;  $cl$  is pre-loaded with  $k$ 
```

As shown in Fig. 1.4 and in the register-level bit operations description, *shld* places its second operand to the right of its first, then shifts the entire bit string thus generated to the left by  $k$  bits specified in the third operand. It then updates only the first operand according to the result of the shifts. The second operand is always a register ( $r$ ) and remains unchanged. *Shrd* performs the corresponding right shift by placing the second operand to the left of the first, shifts the whole bit string right, and updates only the first operand. For every shift the bit shifted out is moved into the carry flag.

### Examples

1. The instruction

$$shfrd\ ax, bx, 10$$

logically shifts the contents of  $ax$  right by 10 bit positions. The right most 10 bits of  $bx$  are right shifted into the leftmost bits of  $ax$ . The contents of  $bx$  remain unmodified. In terms of bit operations the following events take place:

- (a)  $CF \leftarrow ax(9)$
- (b)  $ax(5, \dots, 0) \leftarrow ax(15, \dots, 6)$
- (c)  $ax(15, \dots, 6) \leftarrow bx(9, \dots, 0)$

2. Suppose  $eax$  holds the value  $01234567h$ , and  $ebx$  holds the value  $89ABCDEFh$ , then the execution of

$$shld\ eax, ebx, 8$$

will update  $eax$  to a value  $23456789h$ , and make  $CF = 1$ . If the instruction

$$shrd\ eax, ebx, 8$$

is executed instead, then the new contents of  $eax$  will be  $EF012345h$ , and make  $CF = 0$ . The contents of  $ebx$  are not changed in neither case.

## 6.3.2 Arithmetic shifts

These shifts are designed for signed values to be quickly multiplied or divided by powers of 2. They insure that the sign bit is treated correctly.

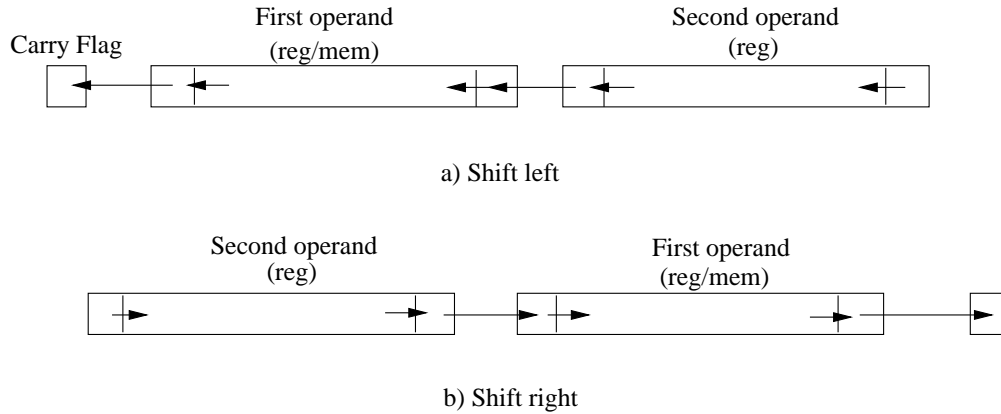


Figure 6.4: Double shift operations

```

sal r/m, k    ; similar to shl
sal r/m, cl    ; similar to shl
sar r/m, k    ;  $CF \leftarrow r/m(k-1)$ 
               ;  $r/m(msb-k, \dots, 0) \leftarrow r/m(msb, \dots, k)$ 
               ;  $r/m(msb, \dots, msb-k+1) \leftarrow r/m(msb)$ 
sar r/m, cl    ; cl is pre-loaded with k

```

The *sal* (shift arithmetic left) is functionally identical to *shl*. Furthermore, it is assembled into exactly the same binary code as *shl*. As long as the sign bit is not changed by the shift, the result will be correct. For each shift the most significant bit is shifted into the carry flag. On the other hand the *sar* (shift arithmetic right) instruction does not change the most significant bit (the sign bit) but for each shift a copy is transferred to the next bit on the right. Thus, the  $k$  rightmost bits are replaced by the most significant bit. All other bits are shifted to the right. The least significant bit is shifted into the carry flag as shown in Fig. 1.5.



Figure 6.5: Arithmetic right shift

*Example:* The following sequence of operations illustrate an arithmetic right shift with a shifting factor  $k = 3$ :

```

mov ax, 0A004h ; ax = 1010 0000 0000 0100
mov cl, 3
sar ax, cl      ; ax = 1111 0100 0000 0000, and C = 1

```

The bit-level operations can be described in the following steps:

1.  $CF = ax(2) = 1$ ,
2.  $ax(12, \dots, 0) \leftarrow ax(15, \dots, 3)$ ,
3.  $ax(15, \dots, 13) \leftarrow ax(15) = 1$ .

### 6.3.3 Rotate Instructions

The rotate shift instructions treat data as if it is a circular structure. The bit that is shifted out on one end is shifted in on the other side. The two simplest rotate instructions are *rol* and *ror* for left and right rotations, respectively. Each bit shifted around is also copied into the carry flag as shown in Fig. 1.6. The format and register-level operations are described as follows:

```

rol r/m, k ; CF ← r/m(msb - k + 1)
            ; r/m(k - 1, ..., 0) ← r/m(msb, ..., msb - k + 1)
            ; r/m(msb, ..., k) ← r/m(msb - k, ..., 0)
rol r/m, cl ; cl is pre-loaded with k
ror r/m, k ; CF ← r/m(k - 1)
            ; r/m(msb, ..., msb - k + 1) ← r/m(k - 1, ..., 0)
            ; r/m(msb - k, 0) ← r/m(msb, ..., k)
ror r/m, cl ; cl is pre-loaded with k

```



Figure 6.6: Rotate shifts

*Example:* The following code segment shows the effects of applying *ror* twice:

```

mov al, 01h ;al = 0000 0001
ror al, 1   ;al = 1000 0000, CF = 1
ror al, 1   ;al = 0100 0000, CF = 0

```

There are two additional rotate instructions, *rcl* and *rcr* that rotate data bits and the carry flag to the left, and to the right, respectively. For example, if the contents of the *ax* register are rotated with these instructions, the 17-bits made up of *ax* and the carry flag are rotated. Rotation with carry is illustrated in Fig. 1.7

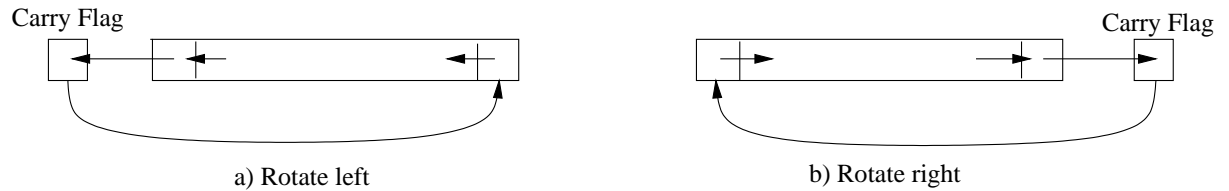


Figure 6.7: Rotation with carry

## 6.4 Boolean Instructions

There are three basic boolean instructions that every instruction set must include: *and*, *or* and *not*. More complex instructions such *xor*, *nor*, and *nand* can be implemented using the basic instructions and the instruction set may or may not provide them. However, the *xor* operation is so commonly used that it is found in all instruction sets of modern machines. Boolean operations are very useful for manipulating selected individual bits of data.

### 6.4.1 The AND instruction

The result of a logical *and* operation on two bits is 1 only if both bits are 1, else the result is 0. The *and* instruction performs a bitwise logical *and* between its two operands and stores the result in the destination operand. The format along with its register-level operation is described next:

$$\text{and } r/m, r/m/k \quad ; \quad r/m \leftarrow r/m \wedge r/m/k$$

The destination operand can be a register or a memory reference. The source operand can be a register, a memory reference, or an immediate value (*k*). In the forms with an 8-bit constant operand and a longer first operand, the constant operand is considered to be signed, and is sign-extended to the length of the first operand. A typical use of this instruction is to use a bit masking to clear or reset selected bits.

*Example:* Use a bit mask = 0000 1111 to clear the four most significant bits in register *al*:

```
mov al, 0011 1011b
and al, 0000 1111b    ;al = 0000 1011
```

### 6.4.2 The OR instruction

The inclusive logical *or* operation on 2 bits is 0 only if both bits are 0, else the result is 1. The *or* instruction executes the logical *or* operation on its two operands. The format and operation description follow:

$$\text{or } r/m, r/m/k \quad ; r/m \leftarrow r/m \vee r/m/k$$

The results of the *or* operation are placed in the destination operand. In the forms with an 8 bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign extended to the length of the first operand. Typical uses of the *or* instruction include setting selected bits, and oring and operand with itself to set flags.

*Example:* With the use of a bit mask set the most significant bits in register *al*:

```
mov al, 0011 1011b
or al, 1111 0000b    ;al = 1111 1011
```

### 6.4.3 The NOT instruction

The *not* instruction is a unary operation that returns the one's complements of the value in its single operand. No flag bit is affected.

$$\text{not } r/m \quad ; r/m \leftarrow \overline{r/m}$$

### 6.4.4 The XOR instruction

The *xor* instruction performs a bitwise *xor* operation between its two operands. Each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1. The result is stored in the destination operand. The format of the *xor* and the description of its register-level operation are the following:

`xor r/m, r/m/k ;  $r/m \leftarrow r/m \otimes r/m/k$`

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign extended to the length of the first operand. A typical use of the *xor* instruction is to toggle one or more selected bits.

*Example:* Toggle the first two least significant bits on the contents of the register *al*:

```
mov al, 0011 1010b
xor al, 0000 0011b ;al = 0011 1001
```

### 6.4.5 The TEST Instruction

The *test* instruction performs an *and* operation, but does not store the result. It only sets the *flags* register based on what the result would be. The *test* instruction is to logical operations what the *cmp* instruction is to arithmetic operations. The results do not alter the contents of the destination operand. The format and operation are described as follows:

`test r/m, r/m/k ;  $result = r/m \wedge r/m/k$`

*Example:* Set flags to inspect the nature of the value stored in *al*:

```
mov al, 0010, 0101b
test al, 0000 10001b ;result = 0000 0001, ZF = 0
```

The results of the test indicate that the contents of *al* correspond to a positive value.

## 6.5 Exercises

1. Suppose we have the following two instructions dealing with signed operations. Indicate the final results in *ax* and the state of the appropriate flags.

```
...
mov ax, FF00h
add ax, 1000h
...
```



2. Consider the assembly code shown below. Indicate in the spaces provided the results in *ax* and the carry flag.

```

...
mov ax, F00Fh
sar ax, 4           ;ax =          CF =

```

```

...
mov ax, F00Fh
shr ax, 4           ;ax =          CF =

```

3. Recall that an  $n$ -bit division requires a  $2n$ -bit dividend. Suppose  $X$  and  $Y$  are two 32-bit signed values stored in memory and the following operation is required  $Z = X/Y$ .
- Write the appropriate set of instructions to implement this division.
  - Indicate (in the comment line) where the results are returned, and extend your code to store these results at  $Z$ .
4. Following up from the previous problem, write a sequence of instructions to do the following: 1) if the remainder is an odd number jump to label "odd", 2) else continue with the next instruction.
5. Show the contents of the registers indicated and the carry flag as the code executes.

```

segment data
X    dw    0FB00h
B    dw    0F100h

```

```

segment code
...
mov dx, word [X]
mov ax, word [X]
clc
add ax, word [B]    ; dx:ax =          CF =

adc dx, 0           ; dx:ax =          CF =
...

```

6. Assume *eax* contains A23489ABh, determine the result after executing *cdq*.

7. The current contents of *eax* have to be modified based on the condition that it contains an even integer value. Write a sequence of instructions that tests that condition and if true increments the value in *eax* or decrements it otherwise.
8. Suppose *ax* and *bx* contain 1234h, and 5678h, respectively. Determine the contents of each register after executing each of the following instructions:
  - (a) `shld ax, bx, 4`
  - (b) `shrd bx, ax, 4`
9. Describe the difference between the *not* and the *neg* instructions.
10. A *palindrome* is a binary pattern that reads the same forward and backward. For example: 11100111 is a palindrome, and 11100101 is not. Write a sub-procedure that determines if *eax* contains a palindrome.
11. Write the appropriate *or* instruction to set bits 1,3,5,7,9, and 11 in register *ax*.
12. Show how the instruction *rol* can be used to rotate *ax* and *bx*, with *ax* containing the upper 16 bits.
13. Describe the difference between the *and* and *test* instructions.
14. Write a sequence of instructions needed to count the number of 0's in *eax*.
15. Write a sub-procedure to swap nibbles in *ax*.
16. Suppose we have some data currently stored in *ax*:
  - (a) Use an appropriate boolean instruction to clear the least significant bit,
  - (b) Repeat a) to toggle the least significant bit.