Chapter 8 – Bit Manipulation

A computer contains many integrated circuits that enable it to perform its functions. Each chip incorporates from a few to many thousand logic gates, each an elementary circuit that performs Boolean and, or, exclusive or, or not operations on bits that are represented by electronic states.

The CPU is usually the most complex integrated circuit in a PC.

Chapter 8 – Bit Manipulation

The 80x86 (and most other CPUs) can also execute instructions that perform Boolean operations on multiple pairs of bits at one time. This chapter defines the Boolean operations and describes the 80x86 instructions that implement them.

It also covers the instructions that cause bit patterns to shift or rotate in a byte, word, or doubleword, or to shift from one location to another.

Although bit manipulation instructions are very primitive, they are widely used in assembly language programming, often because they provide the sort of control that is rarely available in a high-level language.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

Many high-level languages allow variables of Boolean type; that is, variables that are capable of storing *true* or *false* values. In assembly language the Boolean value *true* is identified with the bit value 1 and the Boolean value *false* is identified with the bit value 0. Figure 8.1 gives the definitions of the Boolean operations using bit values as the operands.

bit1	bit2	Bit1 and bit2	(a) And Operation
0	0	0	
0	1	0	
1	0	0	
1	1	1	
bit1	**bit2**	**bit1 or bit2**	**(b) Or Operation**
0	0	0	
0	1	1	
1	0	1	
1	1	1	
bit1	**bit2**	**bit1 xor bit2**	**(c) Xor Operation**
0	0	0	
0	1	1	
1	0	1	
1	1	0	
Bit1		**not bit1**	**(d) not Operation**
0		1	
1		0	

Figure 8.1: Definitions of logical operations

8.1 Logical Operations

The **or** operation is sometimes called "inclusive or" to distinguish it from "exclusive or" (**xor**). The only difference between **or** and **xor** is for two 1 bits; 1 **or** 1 is 1, but 1 **xor** 1 is 0; that is, "exclusive" or corresponds to one operand or the other true, but not both.

The 80x86 has **`and`**, **`or`**, **`xor`**, and **`not`** instructions that implement the logical operations. The formats of these instructions are

```
and     destination, source
or      destination, source
xor     destination, source
not     destination
```

The first three instructions act on pairs of doublewords, words, or bytes, performing the logical operations on the bits in corresponding positions from the two operands.

8.1 Logical Operations

The not instruction does not affect any flag. However, each of the other three Boolean instructions affects CF, OF, PF, SF, ZF, and AF. The carry flag CF and overflow flag OF flags are both reset to 0; the value of the auxiliary carry flag AF may be changed but is undefined. The parity flag PF, the sign flag SF, and the zero flag ZF are set or reset according to the value of the result of the operation. For instance, if the result is a pattern of all 0 bits, then ZF will be set to 1; if any bit of the result is not 0, then ZF will be reset to 0.

The and, or, and xor instructions all accept the same types of operands, use the same number of clock cycles for execution, and require the same number of bytes of object code. They are summarized together in Fig. 8.2. Information about the not instruction is given in Fig. 8.3.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

They are summarized together in Fig. 8.2. Information about the not instruction is given in Fig. 8.3.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode		
		386	486	Pentium		and	or	xor
register 8	immediate 8	2	1	1	3	80	80	80
register 16	immediate 8	2	1	1	3	83	83	83
register 32	immediate 8	2	1	1	3	83	83	83
register 16	immediate 16	2	1	1	4	81	81	81
register 32	immediate 32	2	1	1	6	81	81	81
AL	immediate 8	2	1	1	2	24	0C	34
AX	immediate 16	2	1	1	3	25	0D	35
EAX	immediate 32	2	1	1	5	25	0D	35

8.1 Logical Operations

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
memory byte	immediate 8	5	2	2	3+	80
Memory word	immediate 8	5	2	2	3+	83
Memory doubleword	immediate 8	5	2	2	3+	83
Memory word	immediate 16	5	2	2	4+	81
Memory doubleword	immediate 32	5	2	2	6+	81
register 8	register 8	2	1	1	2	38
register 16	register 16	2	1	1	2	3B
register 32	register 32	2	1	1	2	3B

Figure 8.2 `and`, `or`, and `xor` instructions

8.1 Logical Operations

Destination Operand	Clock Cycles			Number of Bytes	opcode
	386	486	Pentium		
register 8	2	1	1	2	F6
register 16	2	1	1	2	F7
register 32	2	1	1	2	F7
memory byte	6	3	3	2+	F6
memory word	6	3	3	2+	F7
memory doubleword	6	3	3	2+	F7

Figure 8.3 not Instructions

Chapter 8 – Bit Manipulation

8.1 Logical Operations

Here are some examples showing how the logical instructions work.

Before	Instruction executed	Bitwise Operation	After
AX: E2 75 CX: A9 D7	and ax,cx	1110 0010 0111 0101 1010 1001 1101 0111 1010 0000 0101 0101	AX: A0 55 SF 1 ZF 0
DX: E2 75 Value: A9 D7	or dx,value	1110 0010 0111 0101 1010 1001 1101 0111 1110 1011 1111 0111	DX: EB F7 SF 1 ZF 0
BX: E2 75	xor bx,0a9d7h	1110 0010 0111 0101 1010 1001 1101 0111 0100 1011 1010 0010	BX: 4B A2 SF 0 ZF 0
AX: E2 75	not ax	1110 0010 0111 0101 0001 1101 1000 1010	AX: 1D 8A

Chapter 8 – Bit Manipulation
8.1 Logical Operations

Each of the logical instructions has a variety of uses.

- ✓ **One application of the and instruction is to clear selected bits in a destination.**

For example, to clear all but the last four bits in the EAX register, the following instruction can be used.

```
and     eax,0000000fh          ;clear first 28 bits of EAX
```

If EAX originally contained 4C881D7B, this and operation would yield 0000000B

```
0100 1100 1000 1000 0001 1101 0111 1011 4C881D7B
0000 0000 0000 0000 0000 0000 0000 1111 0000000F
0000 0000 0000 0000 0000 0000 0000 1011 0000000B
```

A value that is used with a logical instruction to alter bit values is often called a **mask**.

8.1 Logical Operations

- ✓ **The or instruction is useful when selected bits of a byte or word need to be set to 1 without changing other bits.**

Observe that if the value 1 is combined with either a 0 or 1 using the **or** operation, then the result is 1. However, if the value 0 is used as one operand, then the result of an **or** operation is the other operand.

- ✓ **The exclusive or instruction will complement selected bits of a byte or word without changing other bits.**

This works since 0 **xor** 1 is 1 and 1 **xor** 1 is 0; that is, combining any operand with 1 using an **xor** operation results in the opposite of the operand value.

8.1 Logical Operations

- **A second use of logical instructions is to implement high-level language Boolean operations.**

One byte in memory could be used to store eight Boolean values. If such a byte is at *flags*, then the statement

```
and     flags,11011101b     ;flag5 := false; flag1 := false
```

assigns value *false* to bits 1 and 5, leaving the other values unchanged.

If the byte in memory at *flags* is being used to store eight Boolean values, then an or instruction can assign *true* values to any selected bits. For instance, the instruction

```
or      flags,00001100b     ;flag3 := true; flag2 := true
```

assigns true values to bits 2 and 3 without changing the other bits.

8.1 Logical Operations

If the byte in memory at *flags* is being used to store eight Boolean values, then an xor instruction can negate selected values. For instance, the design statement

```
flag6 := NOT flag6;
```

can be implemented as

```
xor     flags,01000000b          ;flag6 := not flag6
```

- ✓ **A third application of logical instructions is to perform certain arithmetic operations.**

Suppose that the value in the EAX register is interpreted as an unsigned integer. The expression (*value* mod 32) could be computed using the following sequence of instructions.

```
mov     edx,0       ;extend value to quadword
mov     ebx,32      ;divisor
div     ebx         ;divide value by 32
```

8.1 Logical Operations

Following these instructions, the remainder (value mod 32) will be in the EDX register. The following alternative sequence leaves the same result in the EDX register without, however, putting the quotient in EAX.

```
mov     edx,eax          ;copy value to DX
and     edx,0000001fh    ;compute value mod 32
```

This choice is much more efficient than the first one It works because the value in EDX is a binary number; as a sum it is

$$bit31*2^{31} + bit30*2^{30} + \ldots + bit2*2^{2} + bit1*2 + bit0$$

Since each of these terms from $bit31*2^{31}$ down to $bit5*2^{5}$ is divisible by 32 (2^5), the remainder upon division by 32 is the bit pattern represented by the trailing five bits, those left after masking by 0000001F. Similar instructions will work whenever the second operand of mod is a power of 2.

8.1 Logical Operations

- ✓ **A fourth use of logical instructions is to manipulate ASCII codes.**

Recall that the ASCII codes for digits are 30_{16} for 0, 31_{16} for 1, and so forth, to 39_{16} for 9. Suppose that the AL register contains the ASCII code for a digit, and that the corresponding integer value is needed in EAX. If the value in the high-order 24 bits in EAX are known to be zero, then the instruction

```
sub     eax,00000030h     ;convert ASCII code to integer
```

will do the job. If the high-order bits in EAX are unknown, then the instruction

```
and     eax,0000000fh     ;convert ASCII code to integer
```

is a much safer choice. It ensures that all but the last four bits of EAX are cleared.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

The or instruction can be used to convert an integer value between 0 and 9 in a register to the corresponding ASCII character code.

For example, if the integer is in BL, then the following instruction changes the contents of BL to the ASCII code.

```
or      bl,30h                  ;convert digit to ASCII code
```

If BL contains 04, then the or instruction will yield 34.

With the 80x86 processors, the instruction

```
add     bl,30h
```

does the same job using the same number of clock cycles and object code bytes. However, the or operation is more efficient than addition with some CPUs.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

An **xor** instruction can be used to change the case of the ASCII code for a letter. Suppose that the CL register contains the ASCII code for some upper- or lowercase letter. The ASCII code for an uppercase letter and the ASCII code for the corresponding lowercase letter differ only in the value of bit 5. For example, the code for the uppercase letter S is 53_{16} (01010011_2) and the code for lowercase s is 73_{16} (01110011_2). The instruction

```
xor     cl,00100000b            ;change case of letter in CL
```

"flips" the value of bit 5 in the CL register, changing the value to the ASCII code for the other case letter.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

The 80x86 instruction set includes **`test`** instructions that function the same as and instructions except that destination operands are not changed. This means that the only job of a test instruction is to set flags. One application of a test instruction is to examine a particular bit of a byte or word. The following instruction tests bit 13 of the DX register.

```
test     dx,2000h          ;check bit 13
```

Note that 2000 in hex is the same as 0010 0000 0000 0000 in binary, with bit 13 equal to 1. Often this test instruction would be followed by a jz or jnz instruction, and the effect would be to jump to the destination if bit 13 were 0 or 1, respectively.

Chapter 8 – Bit Manipulation

8.1 Logical Operations

The test instruction can also be used to get information about a value in a register. For example,

```
test    cx,cx                ;set flags for value in CX
```

"ands" the value in the CX register with itself, resulting in the original value. ("Anding" any bit with itself gives the common value.) The flags are set according to the value in CX. The instruction

```
and     cx,cx                ;set flags for value in CX
```

will accomplish the same goal and is equally efficient. However, using test makes it clear that the only purpose of the instruction is testing.

8.1 Logical Operations

The various forms of the test instruction are listed in Fig. 8.4.

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
register 8	immediate 8	2	1	1	3	F6
register 16	immediate 16	2	1	1	4	F7
register 32	immediate 32	2	1	1	6	F7
AL	immediate 8	2	1	1	2	A8
AX	immediate 16	2	1	1	3	A9
EAX	immediate 32	2	1	1	5	A9
memory byte	immediate 8	5	2	2	3+	F6
memory word	immediate 16	5	2	2	4+	F7
memory doubleword	immediate 32	5	2	2	6+	F7

8.1 Logical Operations

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
register 8	register 8	2	1	1	3	84
register 16	register 16	2	1	1	4	85
register 32	register 32	2	1	1	6	85
memory byte	register 8	5	2	2	2+	84
memory word	register 16	5	2	2	2+	85
memory doubleword	register 32	5	2	2	2+	85

Figure 8.4 test instruction

8.2 Shift and Rotate Instructions

Shift and rotate instructions enable the programmer to change the position of bits within a doubleword, word, or byte.

Shift instructions slide the bits in a location given by the destination operand to the left or to the right. The direction of the shift can be determined from the last character of the mnemonic

- ✓ sal and shl are left shifts
- ✓ sar and shr are right shifts

Shifts are also categorized as logical or arithmetic

- ✓ shl and shr are logical shifts
- ✓ sal and sar are arithmetic shifts

The difference between arithmetic and logical shifts is explained below.

8.2 Shift and Rotate Instructions

The table in Fig. 8.5 summarizes the mnemonics.

	left	right
logical	shl	shr
arithmetic	sal	sar

Figure 8.5 Shift Instructions

The source code format of any shift instruction is

```
s-      destination,count
```

There are three versions of the count operand. This operand can be the number 1, another number serving as a byte-size immediate operand, or the register specification CL. The original 8086/8088 CPU had only the first and third of these options.

Chapter 8 – Bit Manipulation
8.2 Shift and Rotate Instructions

An instruction having the format

```
s-      destination,1
```

causes a shift of exactly one position within the destination location.

With the format

```
s-      destination,immediate8
```

an immediate operand of 0 to 255 can be coded. However, most of the 80x86 family mask this operand by 00011111_2; that is they reduce it mod 32 before performing the shift. This makes sense because you cannot do over 32 meaningful shift operations to an operand no longer than a doubleword.

Chapter 8 – Bit Manipulation
8.2 Shift and Rotate Instructions

In the final format,

```
s-      destination,cl
```

the unsigned count operand is in the CL register. Again, most 80x86 CPUs reduce it modulo 32 before beginning the shifts.

Arithmetic and logical left shifts are identical; the mnemonics sal and shl are synonyms that generate the same object code.

When a left shift is executed, the bits in the destination slide to the left and 0 bits fill in on the right. The bits that fall off the left are lost except for the very last one shifted off; it is saved in the carry flag CF. The sign flag SF, zero flag ZF, and parity flag PF are assigned values corresponding to the final value in the destination location. The overflow flag OF is undefined for a multiple-bit shift; for a single-bit shift (*count*=1) it is reset to 0 if the sign bit of the result is the same as the sign bit of the original operand value, and set to 1 if they are different. The auxiliary carry flag AF is undefined.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Arithmetic and logical right shifts are not the same. With both, the bits in the destination slide to the right and the bits that fall off the right are lost except for the very last one shifted off, which is saved in CF.

- ✓ For a logical right shift (shr) 0 bits fill in on the left.
- ✓ with an arithmetic right shift (sar) the original sign bit is used to fill in on the left.

As with left shifts, the values of SF, ZF, and PF depend on the result of the operation, and AF is undefined. The overflow flag OF is undefined for a multiple-bit shift. For a single-bit logical right shift shr, OF is reset to 0 if the sign bit in the result is the same as the sign bit in the original operand value, and set to 1 if they are different. (Notice that this is equivalent to assigning OF the sign bit of the original operand.) With a single-bit arithmetic right shift, sar, OF is always cleared—the sign bits of the original and new value are always the same.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Here are a few examples that illustrate execution of shift instructions; The bit(s) shifted off are separated by a line in the original value. The bit(s) added are in bold in the new value.

Before	Instruction executed	Bitwise Operation	After
CX: A9 D7	sal cx,1	1010 1001 1101 0111 0101 0011 1010 1110	CX: 53 AE SF 0 ZF 0 CF 1 OF 1
AX: A9 D7	shr ax,1	1010 1001 1101 0111 0101 0100 1110 1011	AX: 54 EB SF 0 ZF 0 CF 1 OF 1
BX: A9 D7	sar bx,1	1010 1001 1101 0111 1101 0100 1110 1011	BX: D4 EB SF 1 ZF 0 CF 1 OF 0

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Before	Instruction executed	Bitwise Operation	After
ace: A9 D7	sal ace,4	1010 1001 1101 0111 1001 1101 0111 0000	ace: 9D 70 SF 1 ZF 0 CF 0 OF ?
DX: A9 D7	shr dx,4	1010 1001 1101 0111 0000 1010 1001 1101	DX: 0A 9D SF 0 ZF 0 CF 0 OF ?
AX: A9 D7 CL: 04	sar ax,cl	1010 1001 1101 0111 1111 1010 1001 1101	AX: FA 9D SF 1 ZF 0 CF 0 OF ?

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Figure 8.6 gives the number of clock cycles and number of bytes required using various operand types in shift instructions. Notice that the single-bit shifts are faster than the multiple-bit shifts—often it is more time-efficient to use several single-bit shifts than one multiple-bit shift.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode
		386	486	Pentium		
register 8	1	3	3	1	2	D0
register 16/32	1	3	3	1	2	D1
memory byte	1	7	4	3	2+	D0
memory word/double word	1	7	4	3	2+	D1
register 8	immediate 8	3	2	1	3	C0
register 16/32	immediate 8	3	2	1	3	C1
memory byte	immediate 8	7	4	3	3+	C0

8.2 Shift and Rotate Instructions

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
memory word/double word	immediate 8	7	4	3	3+	C1
register 8	CL	3	3	4	2	D2
register 16/32	CL	3	3	4	2	D3
memory byte	CL	7	4	4	2+	D2
memory word/double word	CL	7	4	4	2+	D3

Figure 8.6 Shift and rotate instructions

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

The shift instructions have many applications.

- ✓ **One of these is to do some multiplication and division operations.**

In fact, for processors without multiplication instructions, shift instructions are a crucial part of routines to do multiplication. Even with the 80x86 architecture, some products are computed more rapidly with shift operations than with multiplication instructions.

In a multiplication operation where the multiplier is 2, a single-bit left shift of the multiplicand results in the product in the original location. The product will be correct unless the overflow flag OF is set. It is easy to see why this works for unsigned numbers; shifting each bit to the left one position makes it the coefficient of the next higher power of two in the binary representation of the number. A single-bit left shift also correctly doubles a signed operand. In fact, one can use multiplication by 2 on a hex calculator to find the result of any single-bit left shift.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

A single-bit right shift can be used to efficiently divide an unsigned operand by 2. Suppose, for example, that the EBX register contains an unsigned operand. Then the logical right shift shr ebx,1 shifts each bit in EBX to the position corresponding to the next lower power of two, resulting in half the original value. The original units bit is copied into the carry flag CF, and is the remainder for the division.

Chapter 8 – Bit Manipulation
8.2 Shift and Rotate Instructions

If EBX contains a signed operand, then the arithmetic right shift

```
sar    ebx,1
```

does almost the same job as an idiv instruction with a divisor of 2. The difference is that if the dividend is an odd negative number, then the quotient is rounded down; that is, it is one smaller than it would be using an idiv instruction.

For a concrete example, suppose that the DX register contains FFFF and the AX register contains FFF7, so that DX-AX has the doubleword size 2's complement representation for 9. Assume also that CX contains 0002. Then idiv cx gives a result of FFFC in AX and FFFF in DX; that is, a quotient of 4 and a remainder of 1. However, if FFFFFF7 is in EBX, then sar ebx,1 gives a result of FFFFFFFB in EBX and 1 in CF, a quotient of −5 and a remainder of +1. Both quotient-remainder pairs satisfy the equation

dividend = quotient*divisor + remainder

8.2 Shift and Rotate Instructions

but with the −5 and +1 combination, the sign of the remainder differs from the sign of the dividend, contrary to the rule followed by **`idiv`**.

Instead of multiplying an operand by 2, it can be doubled by either adding it to itself or by using a left shift. A shift is sometimes slightly more efficient than addition and either is much more efficient than multiplication.

To divide an operand by 2, a right shift is the only alternative to division and is much faster; however, the right shift is not quite the same as division by 2 for a negative dividend.

To multiply or divide an operand by 4, 8, or some other small power of two, either repeated single-bit shifts or one multiple-bit shift can be used.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Shifts can be used in combination with other logical instructions to combine distinct groups of bits into a byte or a word or to separate the bits in a byte or word into different groups. The program shown in Fig. 8.7 prompts for an integer, uses the atod macro to convert it to 2's complement form in the EAX register, and then displays the word in the EAX register as eight hexadecimal digits. To accomplish this display, eight groups of four bits must be extracted from the value in EAX. Each group of four bits represents a decimal value from 0 to 15, and each group must be converted to a character for display. This character is a digit 0 through 9 for integer value 0 (0000_2) through 9 (1001_2) or a letter A through F for integer value 10 (1010_2) through 15 (1111_2).

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

The eight characters are stored right to left in contiguous bytes of memory as they are generated; the EBX register is used to point at the destination byte for each character. The design for the middle of the program is

```
-------------
for count := 8 downto 1 loop
    copy EAX to EDX;
    mask off all but last 4 bits in EDX;
    if value in EDX ≤ 9
    then
        convert value in EDX to a character 0 through 9;
    else
        convert value in EDX to a letter A through F;
    end if;
    store character in memory at address in EBX;
    decrement EBX to point at next position to the left;
    shift value in EAX right four bits;
end for;
 --------------
```

Chapter 8 – Bit Manipulation
8.2 Shift and Rotate Instructions

To implement this design, the instruction

```
        and     edx,0000000fh       ;zero all but last hex digit
```

masks off all but the last four bits in EDX. The if is implemented by

```
        cmp     edx,9                       ;digit?
        jnle    elseLetter                  ;letter if not
        or      edx,30h                     ;convert to character
        jmp     endifDigit
elseLetter:
        add     edx,'A'-10                  ;convert to letter
endifDigit:
```

A value from 0 to 9 is converted to the ASCII code for a digit using the or instruction; add edx,30h would work just as well here. To convert numbers 0A to 0F to the corresponding ASCII codes 41 to 46 for letters A to F, the value 'A'–10 is added to the number. This actually adds the decimal number 55, but the code used is clearer than add edx,55. The shr instruction shifts the value in EAX right four bits, discarding the hex digit that was just converted to a character.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

The shift instructions discussed above shift the bits of an operand in place, except that one bit affects the carry flag. The 80x86 architecture has two additional double shift instructions, shld and shrd. Each of these instructions has the format

```
sh-d      destination, source, count
```

where the destination may be a word or a doubleword in a register or memory, the source is a word or doubleword in a register, and the count is either immediate or in CL. A shld instruction shifts the destination left exactly like a shl instruction, except that the bits shifted in come from the left end of the source operand. The source operand is not changed. A shrd instruction shifts the destination right exactly like a shr instruction, except that the bits shifted in come from the right end of the source operand. For both double shifts, the last bit shifted out goes to CF, and SF, ZF, and PF are given values corresponding to the result in the destination location. The overflow flag OF is left undefined by a double shift.

8.2 Shift and Rotate Instructions

The following two examples illustrate double shift instructions. The one with shld shifts off the leading three hex digits (12 bits) of ECX, filling from the right with the leftmost three hex digits from EAX. The carry flag CF is 1 since the last bit shifted off was the rightmost bit of 3 ($001\mathbf{1}_2$). The example using shrd shifts off the trailing two hex digits (8 bits) of ECX, filling from the left with the rightmost two hex digits from EAX. The carry flag CF is 0 since the last bit shifted off was the leftmost bit of 7 ($\mathbf{0}111_2$).

Before	Instruction executed	After
ECX: 12 34 56 78 EAX: 90 AB CD EF	shld ecx,eax,12	ECX: 45 67 89 0A EAX: 90 AB CD EF CF 1 ZF 0 SF 0
ECX: 12 34 56 78 EAX: 90 AB CD EF CL: 08	Shrd ecx,eax,CL	ECX: EF 12 34 56 EAX: 90 AB CD EF CF 0 ZF 0 SF 1

8.2 Shift and Rotate Instructions

Figure 8.8 lists the various double shift instructions. The source operand is not shown since it is always a register 16 or register 32, the same size as the destination.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode	
		386	486	Pentium		shld	shrd
register 16/32	immediate 8	3	2	4	4	0F 04	0F AC
memory word/double word	immediate 8	7	4	4	4+	0F 04	0F AC
register 16/32	CL	3	3	4	3	0F 05	0F AD
memory word/double word	CL	7	4	5	3+	0F 05	0F AD

Figure 8.8 Double shift instructions

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

A double shift instruction can be used to get a slightly cleaner version of the program in Fig. 8.7. The following code generates the hex digits left-to-right instead of right to-left. Each time through the loop, a shld copies the leading hex digit from EAX into EDX.

```
                lea     ebx,hexOut          ;address for first character
                mov     ecx,8               ;number of characters
forCount:       shld    edx,eax,4           ;get leading hex digit
                and     edx,0000000fh       ;zero all but last hex digit
                cmp     edx,9               ;digit?
                jnle    elseLetter          ;letter if not
                or      edx,30h             ;convert to character
                jmp     endifDigit
elseLetter:     add     edx,'A'-10          ;convert to letter
endifDigit:
                mov     BYTE PTR [ebx],dl   ;copy character to memory
                inc     ebx                 ;point at next character
                shl     eax,4               ;shift one hex digit left
                loop    forCount            ;repeat
```

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Rotate instructions

Rotate instructions are very similar to single shift instructions. With shift instructions the bits that are shifted off one end are discarded while vacated space at the other end is filled by 0s (or 1s for a right arithmetic shift of a negative number). With rotate instructions the bits that are shifted off one end of the destination are used to fill in the vacated space at the other end.

Rotate instruction formats are the same as single shift instruction formats. A single-bit rotate instruction has the format

```
r-   destination,1
```

and there are two multiple-bit versions

```
r-   destination, immediate8
r-   destination, cl
```

Chapter 8 – Bit Manipulation
8.2 Shift and Rotate Instructions

The instructions **rol** (rotate left) and **ror** (rotate right) can be used for byte, word, or doubleword operands in a register or in memory. As each bit "falls off" one end, it is copied to the other end of the destination. In addition, the last bit copied to the other end is also copied to the carry flag CF. The overflow flag OF is the only other flag affected by rotate instructions. It is undefined for multibit rotates.

As an example, suppose that the DX register contains D25E and the instruction

```
rol    dx, 1
```

is executed. In binary, the operation looks like

resulting in 1010 0100 1011 1101 or A4BD. The carry flag CF is set to 1 since a 1 bit rotated from the left end to the right.

Chapter 8 – Bit Manipulation

8.2 Shift and Rotate Instructions

Timings and opcodes for rotate instructions are identical to those for shift instructions. They are given in Fig. 8.6.

A rotate instruction can be used to give yet another version of the program in Fig. 8.7. This one produces the hex digits in a left-to-right order and has the advantage of leaving the value in EAX unchanged at the end since eight rotations, four bits each time, result in all bits being rotated back to their original positions.

8.2 Shift and Rotate Instructions

```
          lea   ebx,hexOut          ;address for first character
          mov   ecx,8               ;number of characters
forCount: rol   eax,4               ;rotate first hex digit to end
          mov   edx,eax             ;copy all digits
          and   edx,0000000fh       ;zero all but last hex digit
          cmp   edx,9               ;digit?
          jnle  elseLetter          ;letter if not
          or    edx,30h             ;convert to character
          jmp   endifDigit
elseLetter: add edx,'A'-10          ;convert to letter
endifDigit:
          mov   BYTE PTR[ebx],dl    ;copy character to memory
          inc   ebx                 ;point at next character
          loop  forCount            ;repeat
```

8.2 Shift and Rotate Instructions

There is an additional pair of rotate instructions, **`rcl`** (rotate through carry left) and **`rcr`** (rotate through carry right).

Each of these instructions treats the carry flag CF as if it were part of the destination.

This means that

```
rcl    eax,1
```

shifts bits 0 through 30 of EAX left one position, copies the old value of bit 31 into CF and copies the old value of CF into bit 0 of EAX. The rotate through carry instructions obviously alter CF; they also affect OF, but no other flag. The opcodes for rotate through carry instructions are the same as the corresponding shift instructions and can be found in Fig. 8.6.

Chapter 8 – Bit Manipulation

8.3 Converting an ASCII String to a 2's Complement Integer

The atoi and atod macros have been used to scan an area of memory containing an ASCII representation of an integer, producing the corresponding word-length 2's complement integer in the EAX register. These macros and the procedures they call are very similar. This section uses atod as an example.

The atod macro expands into the following sequence of instructions.

```
lea     eax,source      ;source address to EAX
push    eax             ;source parameter on stack
call    atodproc        ;call atodproc(source)
```

These instructions simply call procedure *atodproc* using a single parameter, the address of the string of ASCII characters to be scanned. The EAX register is not saved by the macro code since the result is to be returned in EAX. The actual source identifier is used in the expanded macro, not the name *source*.

Chapter 8 – Bit Manipulation

8.3 Converting an ASCII String to a 2's Complement Integer

The actual ASCII to 2's complement integer conversion is done by the procedure *atodproc.*

The assembled version of this procedure is contained in the file IO.OBJ. Source code for *atodproc* is shown in Fig. 8.9.

The procedure begins with standard entry code. The flags are saved so that flag values that are not explicitly set or reset as promised in the comments can be returned unchanged.

The popf and pop instructions at AToDExit restore these values; however, the word on the stack that is popped by popf will have been altered by the body of the procedure, as discussed below.

Chapter 8 – Bit Manipulation

8.4 The Hardware Level—Logic Gates

The first job of *atodproc* is to skip leading spaces, if any. This is implemented with a straightforward while loop. Note that BYTE PTR [esi] uses register indirect addressing to reference a byte of the source string. Following the while loop, ESI points at some nonblank character.

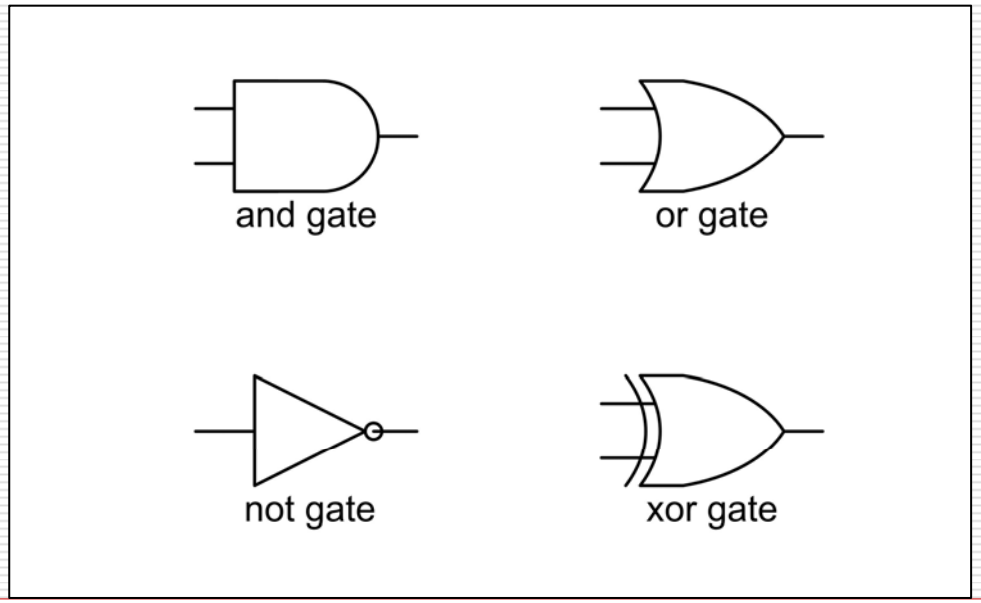
The main idea of the procedure is to compute the value of the integer by implementing the following left-to-right scanning algorithm.

```
value :=0;
while pointing at code for a digit loop
    multiply value by 10;
    convert ASCII character code to integer;
    add integer to value;
    point at next byte in memory;
end while;
```

Chapter 8 – Bit Manipulation

8.4 The Hardware Level—Logic Gates

Digital computers contain many integrated circuits and many of the components on these circuits are *logic gates*. A logic gate performs one of the elementary logical operations described in Section 8.1: **and**, **or**, **xor**, or **not**. Each type of gate has a simple diagram that represents its function. These diagrams are pictured in Fig. 8.10, with inputs shown on the left and output on the right.



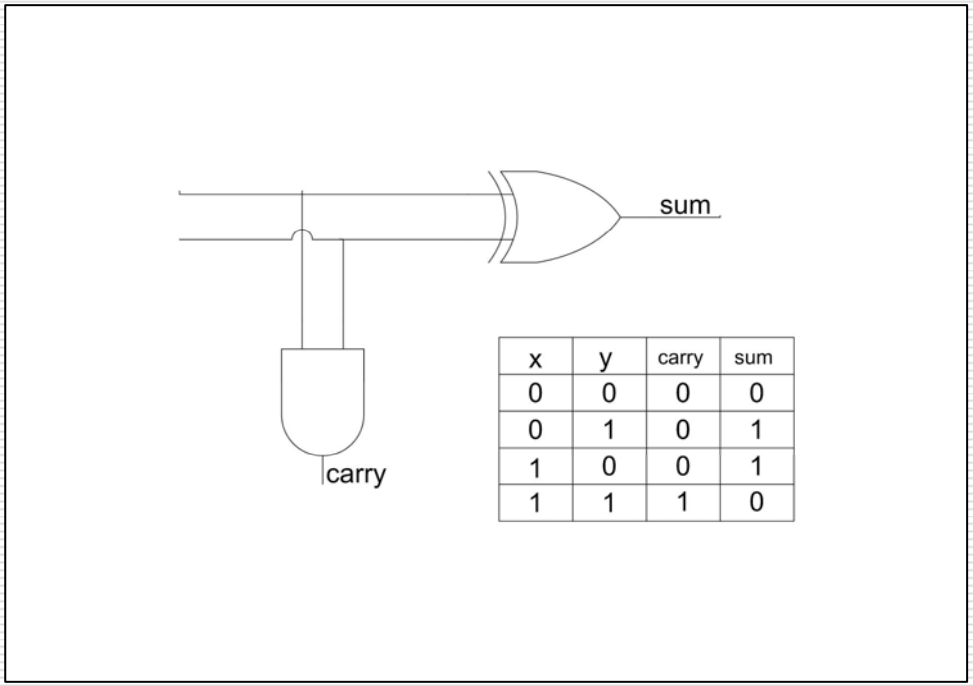
Chapter 8 – Bit Manipulation

8.4 The Hardware Level—Logic Gates

These simple circuits operate by getting logic 0 or 1 inputs and putting the correct value on the output. For example, if the two inputs of the **or** circuit are 0 and 1, then the output will be 1. Logic values 0 and 1 are often represented by two distinct voltage levels.

These simple circuits are combined to make the complex circuits that perform a computer's operations. For example, Fig. 8.11 pictures a *half* adder circuit. The logic values at inputs x and y of this circuit can be thought of as two bits to add. The desired results are 0+0=0, 1+0=1, and 0+1=1, each with a carry of 0, and 1+1=0 with a carry of 1. These are exactly the results given by a half adder circuit.

Chapter 8 – Bit Manipulation
8.4 The Hardware Level—Logic Gates



x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Chapter 8 – Bit Manipulation

Chapter Summary

This chapter has explored the various 80x86 instructions that allow bits in a byte, word, or doubleword destination to be manipulated. The logical instructions and, or, and xor perform Boolean operations using pairs of bits from a source and destination. Applications of these instructions include setting or clearing selected bits in a destination. The not instruction takes the one's complement of each bit in its destination operand, changing each 0 to a 1 and each 1 to a 0. The test instruction is the same as the and instruction except that it only affects flags; the destination operand is unchanged.

Shift instructions move bits left or right within a destination operand. These instructions come in single-bit and multiple-bit versions. Single-bit shifts use 1 for the second operand; multiple-bit versions use CL or an immediate value for the second operand and shift the destination the number of positions specified. Vacated positions are filled by 0 bits in all single shift operations except for the arithmetic right shift of a negative number, for which 1 bits are used. Shift instructions can be used for efficient, convenient multiplication or division by 2, 4, 8 or some higher power of two. Double shift instructions get bits to shift in from a source register.

Chapter 8 – Bit Manipulation

Chapter Summary

Rotate instructions are similar to shift instructions. However, the bit that falls off one end of the destination fills the void on the other end. Shift or rotate instructions can be used in combination with logical instructions to extract groups of bits from a location or to pack multiple values into a single byte or word.

The atod macro generates code that calls the procedure *atodproc*. This procedure scans a string in memory, skipping leading blanks, noting a sign (if any), and accumulating a doubleword integer value as ASCII codes for digits are encountered. Logical instructions are used in several places in the procedure.

Logic gates are the primitive building blocks for digital computer circuits. Each gate performs one of the elementary Boolean operations.