C Bit-level Programming

C.1 INTRODUCTION

One of the unique features of Java language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. Bit-level manipulations are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster. As pointed out in Chapter 5, Java supports the following operators:

1. Bitwise logical operators
2. Bitwise shift operators
3. One's complement operator

All these operators work only on integer-type operands.

C.2 BITWISE LOGICAL OPERATORS

There are three logical Bitwise operators. They are:

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise *exclusive* OR (^)

These are binary operators and require two integer-type operands. These operators work on their operands bit by bit starting from the least significant (i.e., the rightmost) bit, setting each bit in the result as shown in Table C.1.

Table C.1 *Java milestones*

op1	*op2*	*op1 & op2*	*op1 | op2*	*op1 ^ op2*
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Bitwise AND

The bitwise AND operator is represented by a single ampersand (&) and is surrounded on both sides by integer expressions. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Let us consider two variables **x** and **y** whose values are 13 and 25. The binary representation of these two variables are

```
x → 0000 0000 0000 1101
y → 0000 0000 0001 1001
```

If we execute statement

z = x & y;

then the result would be

```
z → 0000 0000 0000 1001
```

Although the resulting bit pattern represents the decimal number 9, there is no apparent connection between the decimal values of these three variables. Program C.1 shows how to use the bitwise operators.

Program C.1 *Demonstration of bitwise operators*

```
Class Bitwise
{
    public static void main (String args[])
    {
        int a=13, b=25;
        System.out.println ("a = " + a);
        System.out.println ("b = " + b);
        System.out.println ("a & b = " + (a & b) );
        System.out.println ("a | b = " + (a | b) );
        System.out.println ("a ^ b = " + (a ^ b) );
    }
}
```

The output would be

```
a = 13
b = 25
a & b = 9
a | b = 29
a ^ b = 20
```

Bitwise ANDing is often used to test whether a particular bit is 1 or 0. For example, the following program tests whether the fourth bit of the variable **flag** is 1 or 0.

```
Class Bit1
{
    Static final TEST = 8;              /* represents 00....01000 */
    public static void main (String args[])
    {
        int flag;
        ..........
        ..........
        if ( (flag & TEST) ! = 0)       /* test 4th bit */
        {
```

```
                    System.out.println ("Fourth bit is set \n") ;
                }
                ..........
                ..........
            }
        }
```

Note that the bitwise logical operators have lower precedence than the relational operators and therefore additional parentheses are necessary as shown above.

The following program tests whether a given number is odd or even.

```
        Class Bit2
        {
            public static void main (String args [])
            {
                int test = 1;
                int number;
                // Input a number here
                    ..........
                    ..........
                while (number ! = —1)
                {
                    if (number & test)
                        System.out.println ("Number is odd\n\n");
                else
                        System.out.println ("Number is even\n\n");
                // Input a number here
                ..........
                ..........
                }
            }
        }
```

Output:

```
Input a number
20
Number is even
Input a number
9
Number is odd
Input a number
—1
```

Bitwise OR

The bitwise OR is represented by the symbol | (vertical bar) and is surrounded by two integer operands. The result of OR operation is 1 if *at least* one of the bits has a value of 1; otherwise it is zero. Consider the variables **x** and **y** discussed above.

```
x      →   0000   0000   0000   1101
y      →   0000   0000   0001   1001
x | y  →   0000   0000   0001   1101
```

The bitwise inclusion OR operation is often used to set a particular bit to 1 in a flag. Example:

```
Class Bit3
{
     final static SET = 8;
     public static void main (String args[])
     {
          int flag;
          ..........
          ..........
          flag = flag | SET;
          if ((flag & SET) ! = 0)
          {
               System.out.println ("flag is set \n");
          }
          ..........
          ..........
     }
}
```

The statement

```
flag = flag | SET;
```

causes the fourth bit of flag to set 1 if it is 0 and does not change it if it is already 1.

Bitwise Exclusive OR

The bitwise *exclusive* OR is represented by the symbol ^. The result of exclusive OR is 1 if *only one* of the bits is 1; otherwise it is 0. Consider again the same variables **x** and **y** discussed above.

```
x      →    0000  0000  0000  1101
y      →    0000  0000  0001  1001
           ------------------------
x^y    →    0000  0000  0001  1101
           ------------------------
```

C.3 BITWISE SHIFT OPERATORS

The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols < and > and are used in the following form:

```
Left shift: op << n
Right shift:      op >> n
```

op is the integer expression that is to be shifted and *n* is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand *op* to be shifted to the left by *n* positions. The leftmost *n* bits in the original bit pattern will be lost and rightmost *n* bits positions that are vacated will be filled with 0s.

Similarly, the right-shift operation causes all the bits in the operand *op* to be shifted to the right by *n* positions. The rightmost *n* bits will be lost. The leftmost *n* bit positions that are vacated will be filled with zero, if the *op* is a *positive integer*. If the variable to be shifted is *negative*, then the operation preserves the high-order bit of 1 and shifts only the lower 31 bits to the right.

Both the operands *op* and *n* can be constants or variables. There are two restrictions on the value of *n*. It may not be negative and it may not exceed the number of bits used to represent the left operand *op*.

Let us suppose **x** is a positive integer whose bit pattern is

```
0100     1001     1100     1011
```

then,

```
                                                  Vacated
                                                ↙ positions
x << 3   =  0100   1110   0101   1000
x >> 3   =  0000   1001   0011   1001

                 ↑
              Vacated
             positions
```

Shift operators are often used for multiplication and division by powers of two. Consider the following statement:

```
x = y << 1;
```

This statement shifts one bit to the left in **y** and then the result is assigned to **x**. The decimal value of **x** will be the value of y multiplied by 2. Similarly, the statement

```
x = y >> 1;
```

shifts **y** one bit to the right and assigns the result to **x**. In this case, the value of **x** will be the value of **y** divided by 2.

Java supports another shift operator >>> known as zero-fill-right-shift operator. When dealing with positive numbers, there is no difference between this operator and the right-shift operator. They both shift zeros into the upper bits of a number. The difference arises when dealing with negative numbers. Note that negative numbers have the high-order bit set to 1. The right-shift operator preserves the high-order bit as 1. The zero-fill-right-shift operator shifts zeros into all the upper bits, including the high-order bit, thus making a negative number into positive. Program C.2 demonstrates the use of shift operators.

Program C.2 *Demonstration of Shift operators*

```
Class Shift
{
    public static void main (String args[])
    {
        int a=8, b=–8;
        System.out.println (“a = ” + a + “ b = ” + b);
        System.out.println (“a >> 2 = ” + (a >> 2));
        System.out.println (“a << 1 = ” + (a << 1));
        System.out.println (“a >>> 1 = ” + (a >>> 1));
        System.out.println (“b >> 1 = ” + (b >> 1));
        System.out.println (“b >>> 1 = ” + (b >>> 1));
    }
}
```

The output of Program C.2 would be:

```
a = 8    b = – 8
a >> 2     = 2
a << 1     = 16
```

```
a >>> 1     = 4
b >> 1      = - 4
b >>> 1     = 2147483644
```

C.4 BITWISE COMPLEMENT OPERATORS

The complement operator **~** (also called the one's complement operator) is an unary operator and inverts all the bits represented by its operand. That is, 0s become 1s and 1s become zero. Example:

```
x    =    1001 0110 1100 1011
~x   =    0110 1001 0011 0100
```

This operator is often combined with the bitwise AND operator to turn off a particular bit. For example, the statements

```
x    =  8;         /* 0000 0000 00000 1000 */
flag  =  flag & ~x;
```

would turn off the fourth bit in the variable **flag.**