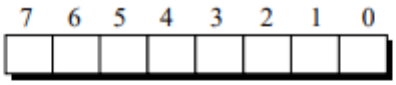
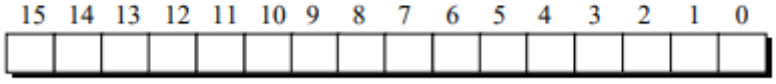
A bit (short of binary digit) is most basic unit of information that can store 1 or 0.

4 bits together form a nibble, 8 bits form a byte, 16 bits form a word and 32 bits form a double word.

Bits are numbered from zero onwards, increasing from right to left.



Character



16-bit Integer

Suppose we wish to store binary values 10110110 in a byte, we can't do that with the help of a C program directly. Because C understands integers, so we should convert this binary into decimal first and then store that decimal number.

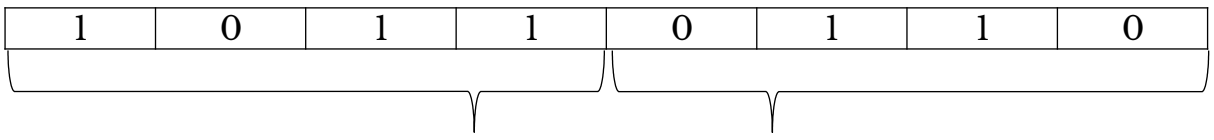
10110110 = 1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 182

As you can see converting from binary to decimal is a tedious task, it is better it should be converted into a hexadecimal system.

Hex	Binary	Hex	Binary	Hex	Binary
0	0000	5	0101	A	1010
1	0001	6	0110	B	1011
2	0010	7	0111	C	1100
3	0011	8	1000	D	1101
4	0100	9	1001	E	1110
				F	1111

Following depicts the method of conversion:

Write the binary in groups of 4 bits. If the binary is 110110, this is equal to 0011 0110 (two zeros are prefixed for making groups of 4 bits). Start from left and write above corresponding Hex digit for that group. The combined hex number formed is the required conversion.



So, 10110110 = 0xB6.

a) Bitwise NOT (~)

This operator is also known as One's complement operator as all the 1's and 0's are interchanged in binary. The symbol ~ is called 'Tilde'.

~ 1011 1110 = 0100 0001

Consider char ch = 32;

In binary 32 = 0010 0000. Now ~ch = 1101 1111.

This number is now a negative number. The reason of this is, if this number is converted back to decimal, we get –

1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 223.

Now this 223 is out of the range of char i.e. -128 to 127. So the number overflows and cycles back to give a negative number. You can predict that negative number as –

223 – 127 – 1 (This additional minus 1 is because zero is included) = 95;

-128 + 95 = -33;

So the number 1101 1111 is -33.

Now if you see, the leftmost bit is actually the bit which determines the sign of the number. The maximum possible number which the 7 bits can store (ignoring the leftmost bit) is 0111 1111, which is equal to 2^8 – 1, because

1 + 2^1 + 2^2 + 2^3 + … + 2^n = 2^ (n+1) – 1.

The minimum possible number which uses that leftmost bit is 1000 0000, which is 2^8 = 128. So if there is a 1 in the leftmost bit, the number has to be negative as it is out of the range, and if there is a 0, the number is positive.

Likewise, computer stores negative numbers by storing 1 in the leftmost bit. Now computer's approach is not to find first a number, then deduct the range from it, cycle the remaining part and give the negative number. Instead, computer follows a method known as 'Two's complement'. In this method, computer checks if there is a 1 in leftmost bit, if yes, it puts a negative sign. Now its computer's job to reach the magnitude of the negative number. Consider the same binary number mentioned above:

To convert that 2's complement to 1's complement, deduct 1 from the binary number. So

1101 1111 – 0000 0001 = 1101 1110

To reach to original number from 1's complement, again take its 1's complement.

~ 1101 1110 = 0010 0001 = 33.

So the number 1101 1111 is -33.

Now let's do reverse of it. Say I have to store – *55* in memory.

55 in binary is 0011 0111.

We take its 1's complement: ~ 0011 0111 = 1100 1000.

To convert 1's complement to 2's complement, add 1 in the 1's complement.

1100 1000 + 0000 0001 = 1100 1001.

So -*55* in memory will be stored as 1100 1001.

This all could be verified with the following program:

```
#include <stdio.h>
int main()
{
    char ch = 32, x;
    unsigned char y;
    x = ~ ch;
    y = ~ ch;
```

```
    printf("%d %d\n", x, y);
    return 0;
}
```

Output: -33 223

Utility of 1's complement/ Bitwise NOT operator:

With a piece of code, we can scan each and every character from a file and replace it with its 1's complement. With every character different, this new file could act as a code file. Thus 1's complement will help in encryption. Now doing 1's complement of that file again, we get the same file. This is decryption.

b) Left shift (<<) and Right shift (>>) operators

Consider again, char ch = 32; and say we want to print ch >> 3. This means shift all bits in ch 3 places to right. The binary of 32 = 0010 0000. Shifting by 3 gives, 0000 0100. Note that the new places are filled with zeros. So the new number is 4.

```
#include <stdio.h>
int main()
{
    unsigned char ch = 32;
    printf("%d\n", ch >> 2);
    printf("%d\n", ch);              // still value of ch is not changed
    printf("%d\n", ch << 1);
    return 0;
}
```

Output: 8 32 64

In the expression a>>b, if b is negative the results are totally unpredictable.

If a is negative, the sign bit continues to extend if shifted by right. Now -5 = 1111 1101,

-5 right shift 1 gives 1111 1101, right shift 2 gives 1111 1110 and right shift 3 gives 1111 1111.

Utility of left shift and right shift operator:

In Windows, the date and time of creation of any file is stored in 2 byte memory each. If you think, date 11/12/2018 like date is stored in 9 bytes (8 + 1 for null).

Let's say we want to store date 09/03/1990. Then,

Required integer = (Current year - 1980)*512

+ Current month*32

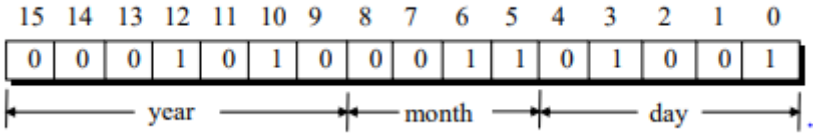
+ Current date

Now, 1980 was year in which DOS/Windows was created and hence taken as base year.

So date = 512*(1990-1980) + 3*32 + 9 = 5225. The binary equivalent of 5225 is

0001 0100 0110 1001.

Now if we divide this binary in 7, 4 and 5 bits, we see leftmost seven contains year, next four contains month and last five contains the date.



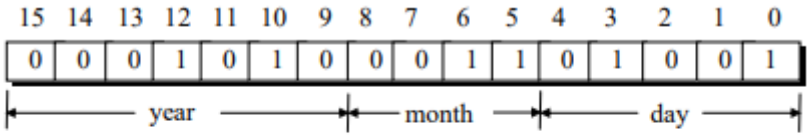
Year = 1*2^1 + 1*2^3 + 1980 = 2 + 8 + 1980 = 1990

Month = 1 + 2 = 3

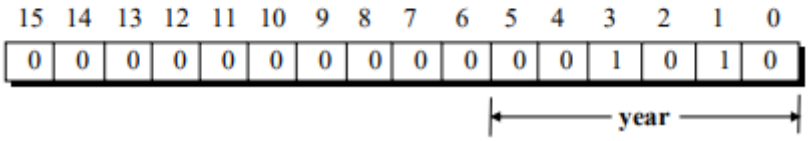
Day = 1 + 8 = 9

Following is the procedure to extract the date from the integer (later a program on same is also given):

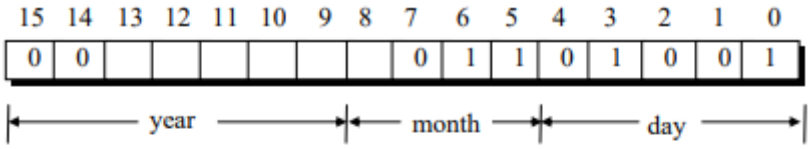
1. Right shift by 9 to get the year



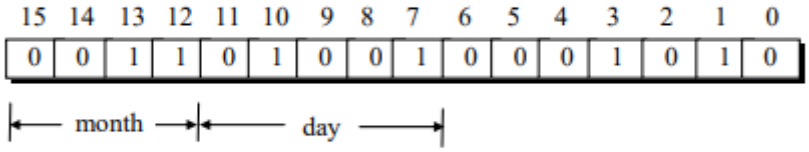
Right shifting by 9 gives



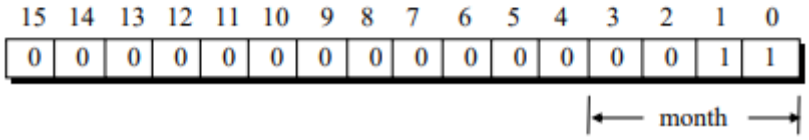
2. Left shift by 7 followed by right shift by 12 to get the month



Left shifting by 7 gives,



Right shifting by 12 gives,



3. Finally left shift by 11 and then right shift by 11, we get the day.

Following is the code:

```
#include <stdio.h>
int main()
{
    int date = 5225;
    int d, m, y;
    y = (date >> 9) + 1980;
    m = (date << 7) >> 12;
    d = (date << 11) >> 11;
    printf("%d %d %d\n", d, m, y);
    return 0;
```

}

Output: 9 3 1990

Likewise, 16 bits of time integer is also distributed as – 5 for the hour, 6 for minutes and 5 for the seconds.

c) AND, OR and XOR

Following is the truth tables of these three bitwise operators:

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Following is the illustration how they are operated:

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x & y	1	0	0	0

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x | y	1	1	1	0

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x ^ y	0	1	1	0

Utility of AND and OR:

AND is used in two situations:

i) To check whether a particular bit is on or off
ii) To turn off a particular bit

Let's say in 8 bits (0 to 7) we want to check whether the 5^{th} bit is on or off. We AND the given binary number with 0010 0000. If the result after ANDing is 32, the bit was ON in original operand and if 0, the bit was OFF.

Thus, depending upon the bit number to be checked in the first operand, we decide the second operand and on ANDing these two operands the result decides whether the bit was ON or OFF. Similarly to put the 3^{rd} bit OFF in a particular bitset, we AND it with 1111 0111.

Now with the logic developed yet we know to ON a particular bit, we OR with 0001 0000 assuming 4^{th} bit was to be made ON.

Such a bitset with one bit value different and all others being same used for the purpose to either access the individual bit or to modify the individual bit at a time is known as 'bitmask'.

Utility of XOR:

Before discussing the utility of XOR, let's look through its definition and properties.

Definition:

i) p ^ q is true only if exactly one of p and q is true.
ii) p1 ^ p2 ^ p3 … ^pn is true if number of variables with true value is odd.

If you wish to write XOR without using ^, you can do so as follows:

x ^ y == (~ x & y) | (x & ~ y)

Properties:

i) $x \wedge 0 = x$.
XORing with 0 gives you back the sum number.

ii) $x \wedge 1 = \sim x$.
XORing with 1 gives the negation of that bit.

iii) $x \wedge x = 0$.
XORing with itself gives zero.

iv) XOR is associative as well as commutative.

Now we can discuss the common application of XOR: Swapping two variables without using temp

To swap the values of x and y, we use the following sequence of steps:

x = x ^ y;

y = x ^ y;

x = x ^ y;

We now see what's happening to values of x and y after each step. Let A and B be the values of x and y initially.

After 1st step: x == A ^ B and y == B

After 2nd step: y == (A ^ B) ^ B == A ^ (B ^ B) ... [By associativity]

y == A ^ 0 ... [By 3rd property]

y == A ...[By 1st property]

So at the end of 2nd step, x == (A ^ B) and y == A.

After 3rd step: x == (A ^ B) ^ A == (A ^ A) ^ B ...[By associativity and commutativity]

x == 0 ^ B == B ... [On similar lines as above]

So at the end of 3rd step: x == B and y == A

There are other applications of XOR too but that will be discussed later on.

d) The showbits() function

We know we can interconvert the hexadecimal, octal and decimal with the use of correct format specifier in printf() function. However there is

no format specifier which prints the binary number when the integer is actually passed to it. We ourselves have to write the function – named showbits ().

Here is the following function (example of char is taken, but with certain modifications, the suitable function of int can also be written :

```
void showbits(unsigned char n)
{
        int i;
        unsigned char j, k, andmask;

        for (i = 7; i >= 0; i--)
        {
                j = i;
                andmask = 1 << j;
                k = n & andmask;
                if (k == 0)             printf("0");
                else                    printf("1");
        }
}
```

First time through the loop, the variable andmask will contain the value 1000 0000 which is obtained by left shifting 1 by seven places. If the variable's leftmost bit is 0, then k will contain 0, else it will contain a non-zero value. In the second go, the value of i is decremented to 6. Hence andmask changes to 0100 0000. The same process of checking takes place till the whole byte is complete.

Utility of showbits () in programs:

The function can be used to actually see what is happening at the bit level. We can –

- Print binary equivalent of 1 byte character
- Print 1's complement of 1 byte character
- Demonstrate the use of leftshift and rightshift operators

The next program uses showbits () to do all of the above.