International Journal of Scientific & Engineering Research, Volume 4, Issue 11, November-2013
ISSN 2229-5518

Application of Bitwise Operators in C

Reshant Chandra, Saurabh Rawat, Tushar Jain

Abstract— Generally a programmer is not concern about the functioning at the bit level, he deals with data type i.e. int char as a whole. He is not concern about how data is actually stored in the memory in the form of array of bits holding value '1' or '0'. C language was particularly created to make development of operating system easier. C language was developed as a replacement for UNIX in development of operating systems. Manipulating data at individual bit level or as group of bits is required in development of operating systems. Functioning at bit level is kept abstracted in normal C program. This research paper deals with the usage of bit wise operator in normal C programs. The research paper tells about the different bitwise operator and illustrate how shift operator work for signed and unsigned integer. Here we have also shown the different implementation of bitwise operator and how it can be used to calculate modulus, manage Boolean flags in C programming.

Index Terms— binary equivalent, bits, bit mask, bitwise operator, Boolean flag, decimal equivalent, modulus operator, shift operator, signed bit

IJSER

1 INTRODUCTION

In C language a 32bit integer is stored as its 32bit binary equivalent. Logical operators work on the whole decimal equivalent of these binary sequence but bit wise operator works on each individual bit. For example a signed integer 245 will be saved as shown in fig 1.1:
.

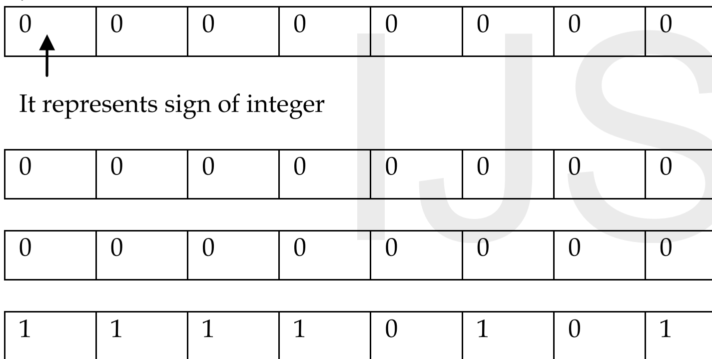

Fig 1.1 Memory allocation of signed interger 245

The left most integer represent the sign of the integer, '1' for negative and '0' for positive. Negative numbers are saved as the 2 compliment of its binary equivalent. Hence -245 will be saved as shown in fig 1.2

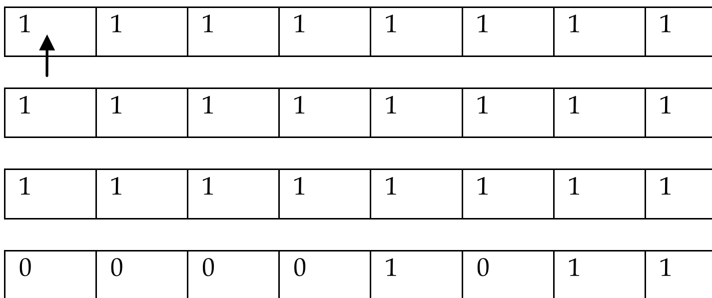

Fig 1.2 Memory allocation of signed interger -245

In fig 1.2 left most bit indicate the sign of the interger which is negative hence 1. Bitwise operator works on each of the above bit of the integer.

2 TYPE OF BITWISE OPERATOR

2.1 Bitwise AND (a & b)

Bitwise AND is similar to logical AND in functioning, the bit-wise AND is applied to each bit of the both the operand, it returns one if all the bits are '1' else it always returns '0'. Truth table of bitwise AND:

A	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Example: a=12, b=21
a = 0 1 1 0 0 (12)
b = 1 0 1 0 1 (21)
a&b = 0 0 1 0 0 (4)

2.2 Bitwise OR (a | b)

Bitwise OR is similar to logical OR in functioning, the bitwise OR is applied to each bit of the both the operand, it returns '1' if any the bits are '1', it returns '0' only if all the bits are '0'. Truth table of bitwise OR:

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Example: a=12, b=21
a = 0 1 1 0 0 (12)
b = 1 0 1 0 1 (21)
a|b = 1 1 1 0 1 (29)

- *Author Reshant Chandra is a faculty of computer science department in Graphic Era University, India. E-Mail: reshant.chandra@gmail.com*
- *Co-Author Saurabh Rawat, faculty, Graphic Era University, India.*
- *Co-Author Tushar Jain*

2.3 Bitwise XOR (a ^ b)

Bitwise XOR is similar to logical XOR in functioning, the bitwise XOR is applied to each bit of the both the operand, it simple adds the bits and discard the remainder i.e. it return '1' if both the bits are different and '0' if both the bits are same. Truth table of bitwise XOR:

A	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Example: a=12, b=21
a = 0 1 1 0 0 (12)
b = 1 0 1 0 1 (21)
a^b = 1 1 0 0 1 (25)

2.4 Bitwise NOT (~ a)

It is applied to single operand. It inverts the each bit of the operand. '0' is changed to '1' and vice versa. Truth table of NOT is as followed.

A	~a
0	1
1	0

Example: a=12
a = 0 1 1 0 0 (12)
~a = 1 0 0 1 1 (19)

2.5 Left shift (a << b)

In left shift operator every bit in the operand is simply moved a given number of bit positions towards left. '0' are inserted from right in the vacant places. Left shifting also work for signed negative integers, since the negative numbers is saved in the form of 2's complement the left shift is applied to 2's compliment of the positive counterpart of the number, but in case of signed integer it may cause a problem cause of the sign bit as a positive number might turn negative and negative number might turn positive. Due to this problem shift are normally implemented on unsigned int. Left shifting for unsigned integer is equivalent to multiplying the number by 2 for each bit shifted till overflow occurs i.e. number becoming lager than a 32 bit integer.

Example of left shifting:
1) Example for left shift for positive integer
Let a = 324 = 0000 0000 0000 0000 0000 0001 0100 0100
x = a << 2 = 0000 0000 0000 0000 0000 0101 0001 0000 = 1296 = 324*2*2 (since 2 bits is shifted hence the result is a*2*2)
2) Example for left shift for negative signed integer
Let a = -10 = 1111 1111 1111 1111 1111 1111 1111 0110
x = a << 1 = 1111 1111 1111 1111 1111 1111 1110 1100 = which is 2's compliment of 20 hence denote -20.

I. 3) Example for inconsistence in left shift for signed integer ,change in signed bit ('0' to '1' and '1' to '0').
Let a = - 34218 = 1111 1111 1111 1111 1000 0001 0101 1110
Now let x= a << 16 will result in
x = 1000 0001 0101 1110 0000 0000 0000 0000 = - 2124546048 which is equal to -34218 * (2^16) hence till now it has worked fine. Now what will happen if we'll left shift x one more time. On x << 1 the resultant will be
0000 0010 1011 1100 0000 0000 0000 0000 = 45875200 which is a positive number as the sign bit got changed. Hence inconsistence may occur if we try to left shift signed integer.

II. 4) Example for overflow due to left shifting in unsigned integer.
Let a = 45875200 = 0000 0010 1011 1100 0000 0000 0000 0000.
Now let x = a << 6 will result in
x = 1010 1111 0000 0000 0000 0000 0000 0000 = 2936012800 = a * (2^6)
till now it works fine (here it's an unsigned integer hence all 32 bits represent value of the integer, there is no signed bit). Now if we try to left shift one more bit the answer should be x * 2 = 5872025600 which is a very big number and can't be represented using 32 bits. Hence implementing one more shift will make the number go out of bound.
x << 1 = 0101 1110 0000 0000 0000 0000 0000 0000 = 1577058304 which not the answer as expected since overflow has occurred.

2.6 Right shift (a>>b)

In right shift operator every bit in the operand is simply moved a given number of bit positions towards right. The right most bits are drop and new bits are inserted in from left. On the bases of which bit to be inserted from the left, right shift is divided into two type, logical right shift and arithmetic right shift.

2.6.1 Logical right shift

Inserts '0' and does not preserve the sign bit. In logical right shift every single bit is shifted right and '0' is inserted at the vacant bit position on the left. It is used to perform division of unsigned integer by 2 for each bit shifted. When 'n' bits are shifted the number is divided by (2^n).

Example of logical right shift:
a = 1988 = 0000 0000 0000 0000 0000 0111 1100 0100 now let us perform x = a >> 2 then
x = 0000 0000 0000 0000 0000 0001 1111 0001 = 497
x = a / (2^2)

2.6.2 Arithmetic right shift

Fill the vacant bits position with the value stored in the signed bit or the left most bit. It preserved the sign bit hence it is also known as signed bit. In case left most bit is '1' then '1' is inserted and if the left most but is '0' then '0' is inserted. It is mainly implemented for signed integer. Shifting right by n bits on the 2's compliment of the signed integer gives the effect of dividing by (2^n).

IJSER

Example of Arithmetic right shift:
a = - 320 = 1111 1111 1111 1111 1111 1110 1100 0000
now let x = a >> 1 then
x = 1111 1111 1111 1111 1111 1111 0110 0000
which is 2's compliment of 160 hence -160
x= a/2

It depends on compiler which right shift to use. Logical right shift is used many when the bits does not represent a number but simply a sequence of bits, while arithmetic right shift are used in case of sign integer when sign bit is need to be preserved. Most of the C compiler use logical right shift for unsigned integer while arithmetic right shift for the signed integer. Dues to this inconsistence shift operations are usually performed on unsigned integer.

3 IMPLEMWNTATION OF BITWISE OPERATORS

Till now we have different type of bitwise operators now we'll show how these bitwise operators can be used in a C programme. Multiple tasks can be done using bitwise operators; shift operators can be used to multiply and divided a number by 2 has been already illustrated above. Similarly there are many other applications of bitwise operator in C language. Different applications of bitwise operators are:

3.1 As a modulus operator:

The modulus operator computes the remainder after dividing its first operand by its second. Bitwise AND operator can be used as modulus operator.
Bitwise AND operator can be used to calculate the modulus of numbers denoted by (2^n), where n is natural number. To calculate modulus of number (2^n) we need to perform bitwise AND with (2^n)-1.To understand how it works we need to first find the binary equivalent (2^n)-1 numbers where n is natural number.

1 = 0000 0000 0000 0000 0000 0000 0000 0001
3 = 0000 0000 0000 0000 0000 0000 0000 0011
7 = 0000 0000 0000 0000 0000 0000 0000 0111
15 = 0000 0000 0000 0000 0000 0000 0000 1111
31 = 0000 0000 0000 0000 0000 0000 0001 1111
63 = 0000 0000 0000 0000 0000 0000 0011 1111
127= 0000 0000 0000 0000 0000 0000 0111 1111

Here we can notice that all these numbers are unique because they have continuous sequence of '1' from left hand side followed by continuous sequence of '0' till the right end. To explain how it works let us take a number say 7 which will calculate modulus for 8.The number 7 has its 3 left most bits as 1 and rest all as zero. If we perform bitwise AND of any number 'x' with 7 the result will be; 3 left most bit will remain unchanged and others will be set to '0'. Hence only the 3 left most binary bits will be left from number 'x' and these 3 bits will represent modulus 8 of x as shown
3 binary bits represent 0-7 values as shown:

a	B	c	Decimal equivalent
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Hence the result will be 3 left most bits which represent 0-7 in decimal which are actually the possible answers for modulus 8. Hence performing bitwise AND with 7 will actually calculate modulus 8.

X = 68 = 0000 0000 0000 0000 0000 0000 0100 0100
Y = 07 = 0000 0000 0000 0000 0000 0000 0000 0111
X & Y = 0000 0000 0000 0000 0000 0000 0000 0100
68 % 8 = 4 (hence it is working correctly)

```
main(){
int num = 15;
for (i=0; i< =100;i++){
	prinf("%d", i & num);
}
}
```

This code prints the modulus 16 of first 100 numbers using bitwise AND with 15.

3.2 As Boolean flags and bit mask:

Let suppose a teacher need to send attendance of a student of 20 days as parameter to a function. One possible solution is that he makes 30 different variables, or an array of 30 integers. The bitwise logical operators are often used to create, manipulate, and read sequences of flags, which are like binary variables. Instead of variables or array these sequence can be used which take much less memory as 1 integer can handle 32 flag.

We need to clear flag for each day of the month:

Flag_one: attendance of day one
(0000 0000 0000 0000 0000 0000 0000 0001)
Flag_Two: attendance of day two
(0000 0000 0000 0000 0000 0000 0000 0010)
Flag_Three: attendance of day three
(0000 0000 0000 0000 0000 0000 0000 0100)
Flag_four: attendance of day four
(0000 0000 0000 0000 0000 0000 0000 1000)
Flag_five: attendance of day five
(0000 0000 0000 0000 0000 0000 0001 0000)
Flag_six: attendance of day six

(0000 0000 0000 0000 0000 0000 0010 0000)
.
.
.
Flag_twenty: attendance of day twenty
(0000 0000 0000 1000 0000 0000 0000 0000)

Flag one mean student was present on day one; flag two mean the student was present on day two and so on.

A single variable 'attendance' will contain the record of the student the values will be stored as followed:

If the student was present on all days the data will be stored as
(0000 0000 0000 1111 1111 1111 1111 1111)

If the student was present on all days except on 5,8 ,13,and 20
(0000 0000 0000 0111 1110 1111 0110 1111)

Attendance & flag_one > 0 means the student was present on day one.
Attendance 0000 0000 0000 0111 1110 1111 0110 1111
Flag_four 0000 0000 0000 0000 0000 0000 0000 1000
Result of & 0000 0000 0000 0000 0000 0000 0000 1000 >0
hence was present on day four which is true.

And if Attendance & flag_one = 0 means the student was absent on day one.
Attendance 0000 0000 0000 0111 1110 1111 0110 1111
Flag_five 0000 0000 0000 0000 0000 0000 0001 0000
Result of & 0000 0000 0000 0000 0000 0000 0000 0000 = 0
hence was absent on day five which is true

To implement this we should be able to set and reset each bit individually of an integer. Handling bit individually are done by using bit mask.

Let us consider a number X, temp =1.

a) If I want to set nth bit i.e. set it to '1' OR operator will be used
X = X | (temp <<n), where X is the number and nth bit need to set to '1'
In the above example if we want to mark the presence of day two we perform
Attendance = attendance | (Flag_Two)

b) If I want to clear nth bit i.e. set it '0' AND operator will be used as followed.
X = X & ~(temp <<n), where X is the number and nth bit need to set to '0'
In the above example if we want to mark the absence of day two we perform
Attendance = attendance & ~(Flag_Two)

c) If I want to toggle nth bit i.e. change its value (if '1' the '0' and if '0' then '1')
X = X ^ (temp <<n), where X is the number and nth need to be toggled

4 Conclusion

We have concluded that an integer is treated as its 32 bit binary equivalent by bitwise operators and there are applied on each bit of the integer. We have seen that we can set clear or toggle a single bit of an integer. Shifting with signed integer may not give consistence result hence shifting is mainly done with unsigned integers but they too face overflow problem. We can use bitwise operator to multiply divide and even find the modulus of a number. Bitwise operators allow us to treat binary sequence of an integer as Boolean flags hence using comparatively very less space. .

References

[1] *Kernighan; Dennis Ritchie (March 1988). The C Programming Language (2nd ed.).*
[2] *Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming,. Addison Wesley, Reading, MA, 2nd edition, 1973.*
[3] *The Ultimate C: Concepts, Programs and Interview Questions (Paperback) by R. Nageswara Rao*
[4] *"Operator (C# Reference)". Microsoft. Retrieved 14 July 2013.*
[5] *C : The Complete Reference 4 Edition (Paperback) by Herbert Schildt*
[6] . Head First C (Paperback) by David Griffiths
[7] Programming with C 3 Edition (Paperback) by Byron Gottfried
[8] Links:
http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/bitwise.html
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators