CP MODULE 5(Part 3of3)
BITWISE OPERATIONS.

Some applications require the operation of individual bits within a word of memory.
Such operations are called bitwise operations.
Operation done on bits are called **bitwise operations**.

Bitwise operatorsin C can be divided into three general categories:

- the one's complement operator ~
- the logical bitwise operators
  - **bitwise and expression**(&)
  - **bitwise or expression(|**)
  - **bitwise exclusive or expression(^**)
- the shift operators
  - right shift operator >>
  - left shift operator <<

1. **The One's Complement Operator**

The one's complement operator (**~**) is a unary operator that inverts the bits of its operand,
1s become 0s and 0s become 1s.
˜1=0
˜0=1
This operator(~) always precedes its operand.
The one's complement operator is sometimes referred to as the **complementation operator.**
It has same precedence as other unary operators.
Associativity of complementation operator is **RIGHT TO LEFT.**

The operand must be an **integer-type** quantity (including integer, long, short, unsigned, char, etc.). Generally, the operand will be an ***unsigned octal or an unsigned hexadecimal*** quantity.

Complement of octal number example

```
#include<stdio.h>
main()
{
 int a=0273;
int b;
b=~a;
 printf("\nb=%o",b);

}
//OUTPUT
```

//177504

<u>Octal number are represented in 3 bits in binary representation</u>. Since this is int It requires 2 bytes(16 bits). If 16 bits are not there remaining spaces at the beginning are filled with zeo.

If number starts with 0(zero) it is octal number. It is printed using %o .
E.g.octal number 015423

<table>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td></td><td colspan="3">1</td><td colspan="3">5</td><td colspan="3">4</td><td colspan="3">2</td><td colspan="3">3</td></tr>
</table>

E.g.octal number is 0273

2	7	3
010	111	011

Here it has only 9 bits, so to make 16 bits, fill the first 7(16-9) bits with 0
So 0273 is correctly represented as

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td></td><td colspan="3"></td><td colspan="3"></td><td colspan="3">2</td><td colspan="3">7</td><td colspan="3">3</td></tr>
</table>

	0	0	0	2	7	3
a	0	000	000	010	111	011

Now there are 16 bits.

Suppose b=~a.
So a is complemented bit by bit

	0	0	0	2	7	3
a	0	000	000	010	111	011
~a	1	111	111	101	000	100
	1	7	7	5	0	4

So complement of a(0273) is 177504.

<u>Complement of HEXADECIMAL number example</u>

```
#include<stdio.h>

main()
{
 int a=0x73f;
int b;
```

```
b=~a;
 printf("\nb=%x",b);
}
//OUTPUT
// f8c0
```

Hexadecimal number is represented using 4 bits in binary representation

0x(Zero X) at the beginning represents that the number is hexadecimal. It is printed using %x or %X

Hexal numbers are

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F, 20,21,23,24,25,26,27,28,29,2A,2B,2C.2D,2E.2F……………………

E.g 0x6db7

0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1
6				d				b				7			

E.g hexadecimal number is 0x73f. It has to be represented in 16 bits(int need 2 bytes=16 boits). If not sufficient zeroes are filled at the beginning.

a	7	3	f
	0111	0011	1111

Only 12 bits are there so 0x73f is correctly represented as

a	0	7	3	f
	0000	0111	0011	1111

Complement of a that is ~a is

a	0	7	3	f
	0000	0111	0011	1111
~a	1111	1000	1100	0000
	f	8	c	0

Complement of 0x73f is f8c0

Examples

~0XC5	0xff3a (hexadecimal constants)
~ox1111	0xeeee (hexadecimal constants)
~0xffff	0 (hexadecimal constants)
~052	0177725 (octal constants)
~0177777	0 (octal constants

hexadecimal values: i = 5b3c

decimal equivalents: i = 23356

i = 0101 1011 0011 1100

~i = 1010 0100 1100 0011

The decimal equivalent of the first bit pattern can be determined by writing 0101 1011 0011 1100

i= $0x2^{15} + 1x2^{14} + 0x2^{13} + 1x2^{12} + 1x2^{11} + 0x2^{10} + 1x2^{9} + 1x2^{8} +$
$0x2^{7} + 0x2^{6} + 1x2^{5} + 1x2^{4} + 1x2^{3} + 1x2^{2} + 0x2^{1} + 0x2^{0} =$
16384 + 4096 + 2048 + 512 + 256 + 32 + 16 + 8 + 4 = 23356

The Logical Bitwise Operators

There are three logical bitwise operators:

- bitwise and (&)
- bitwise exclusive or (^)
- bitwise or (|).

The operations are carried out independently on each pair of corresponding bits within the operands. Thus, the least significant bits (i.e., the rightmost bits) within the two operands will be compared, then the next least significant bits, and so on, until all of the bits have been compared. The results of these comparisons are:

- A **bitwise and expression**(&)will*return 1* if **both** bits have a value of **1** (i.e., if both bits are true).Otherwise, it will return a value of 0.
- A **bitwise exclusive or** (^) expression will return 1 if **one of the bits** has a value of **1** and the other has a value of 0 (one bit is true, the other false). Otherwise, it will return a value of 0.
- A **bitwise or expression** (|) will return a 1 if **one or more of the bits** have a value of **1** (one or both bits are true). Otherwise, it will return a value of 0.

Logical Bitwise Operations

b1	b2	b1 & b2	b1 ^ b2	b1 | b2
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Suppose a and b are unsigned integer variables whose values are Ox6db7 and Oxa726, respectively.

The results of several bitwise operations on these variables are shown below.

a = 0x9248

b = 0X58d9

a & b = 0x1048

a^ b = 0xca91

a | b = dad9

a	9	2	4	8
	1001	0010	0100	1000
b	5	8	d	9
	0101	1000	1101	1001

…………………………………….

a&b	0001	0000	0100	1000
	1	0	4	8

……………………………………..

a^b	1100	1010	1001	0001
	C	a	9	1

……………………………………..

a|b	1101	1010	1101	1001
	D	a	d	9

The associativity for each bitwise operator is **left to right.**
Precedence of operators
======================

High	= = !=	Equality operators
	&	**Bitwise and**
	^	**Bitwise exclusive or**
	|	**Bitwise or**
low	&&	Logical and

Masking

- Masking is a process in which a given bit pattern is **transformed** into another bit pattern using logical bitwise operation.
    - One operand in bitwise operation is the original bit pattern..
    - The second operand is called the **mask**- Mask is a specially selected bit pattern that helps to transform original bit pattern to another bit pattern.

There are several different kinds of masking operations.

1. A ***portion*** of a given bit pattern can be copied to a new word, while the *remainder of the new word is filled with* ***0****s*.
    a. Thus, part of the original bit pattern will be "masked off' from the final result.
    b. The **bitwise and** operator (&) is used for this type of masking operation

2. A ***portion*** of a given bit pattern to be <u>copied to a new word</u>, while t*<u>he remainder of the new word is filled with **1**s</u>*.
    a. The **bitwise or** (|) is used for this type of masking operation
3. A ***portion*** of a given bit pattern can be <u>copied to a new word</u>, while the*<u>remainder of the original bit pattern is **inverted** within the new word.</u>*
    a. The **bitwise exclusive or** (^) is used for this type of masking operation

……………………..……………………..……………………..………………………..

<u>Masking 1: A **portion** of a given bit pattern can be copied to a new word, while the remainder of the new word is filled with **0**s</u>

E.g. Suppose a is an unsigned integer variable whose value is Ox6db7.**<u> Extract the leftmost 6 bits</u>** of this value and assign them to the unsigned integer variable b.

Answer: Fill the rightmost 10 positions in mask with 0s. Fill left most 6 bits with 1s. (Total bits =16 bits)

Perform bitwise & operation between a and mask

0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1
6				d				b				7			

	6	d	b	7	
a	**0110**	**11**01	1011	0111	&
mask	1111	1100	0000	0000	(MASKING BIT)

……………………………………..

b	**0110**	**11**00	0000	0000	(**Here** leftmost 6 bits of a are only copied into b.Other bits are masked)
	6	c	0	0	

Here a&mask=b

0x6db7 & 0xfc00=0x6c00

<u>If any one bit in bitwise & is 0, result of bitwise & is 0</u>

All rightmost 10 bits in the mask b are 0s. So rightmost 10 bits in the result of a&b will be 0. Remaining 6 bits(leftmost most) are 1s. So remaining leftmost 6 bits in the result of a&b will be same as ;leftmost 6 bits in a. (because a,&1=a)

Here when each of the leftmost 6 bits in a is ***bitwise and*** with the corresponding 1 in the mask,, the result will be the same as the original bit in a. (leftmost 6 bits are therefore copied)

Because,

0 & 1 =0

1 & 1 = 1

When each of the rightmost 10 bits in a is ***bitwise and*** with the corresponding 0in the mask, the result is always 0.((remaining rightmost 0 bits are filled with 0s)

Because,

0 & 0=0
1 & 0 =0
Mask used here is 1111 1100 0000 0000(0xfc00). Since the 1s appear in the leftmost bit positions and ***<u>0s at rightmost in this mask, this is dependent on the 16-bit word size</u>***. To avoid this problem take one's complement of this mask, so that instead of 0s in right-most position 1s will come.
~1111 1100 0000 0000 = 000 0011 1111 1111
1111 1100 0000 0000 = ~000 0011 1111 1111

This means that 1111 1100 0000 0000 can be written as ~ 0000 0011 1111 1111 (~ 0x3ff)
b = a & -0x3ff;
Instead of writing 0x6db7 & 0xfc00=0x6c00,it is better to write:
0x6db7 & ~0x3ff=0x6c00

Now this mask is independent of 16 bits since they contain 1s in the rightmost position and 0s in leftmost.

………………………..………………………..………………………..………………………..

E.g. Suppose a is an unsigned integer variable whose value is Ox6db7. **<u>Extract the rightmost 6 bits</u>** of this value and assign them to the unsigned integer variable b.
Answer: Fill the leftmost 10 positions in mask with 0s. Fill rightmost 6 bits inmask with 1s. (Total bits =16 bits)
Perform bitwise & operation between a and mask

	6	d	b	7	
a	0110	1101	10**11**	**0111**	&
mask	0000	0000	0011	1111	(MASKING BIT)
………………………………………..					
b	0000	0000	00**11**	**0111**	(**Here** rightmost 6 bits of a are only copied into b.Other bits are masked)
	6	c	0	0	

Here a&mask=b
0x6db7 & 0x3f=0x37
Here mask used is 0000 0000 001 1111(0x3f). This mak contains 1s in rightmost position.So this mask is independent of the word length

----

<u>Masking 2: A **portion** of a given bit pattern can be copied to a new word, while the remainder of the new word is filled with **1**s</u>

Eg. Suppose that a is an unsigned integer variable whose value is Ox6db7. Transform the corresponding bit pattern into another bit pattern in which the <u>rightmost 8 bits are all 1s</u>, and the <u>leftmost 8 bits retain their original value</u>.

If any one bit in bitwise or is 1, result of bitwise or is 1.Otherwise it is copied.

a = **0110 1101** 1011 0111 (0x 6db7) **|**
mask = 0000 0000 1111 1111(0x 00ff)
……………………………….
b = **0110 1101** 1111 1111
0x 6 d f f
Here a|mask=b
0x6db7 | 0x00ff = 0x 6dff

Here When each of the leftmost 8 bits in a is **bitwise or** with the corresponding 0 in the mask,, the result will be the same as the original bit in a. (leftmost 8 bits are therefore copied)
Because,
0 | 0 =0
1 |0 = 1
When each of the rightmost 8 bits in a is **bitwise or** with the corresponding 1 in the mask, the result is always 1.(remaining rightmost 8 bits are filled with 1s)
Because,

0|1=1
1|1 =1
==

<u>Masking 3: A **portion** of a given bit pattern can be **copied** to a new word, while the **remainder** of the original bit pattern is **inverted** within the new word.</u>

E.g. Suppose that a is an unsigned integer variable whose value is Ox6db7. Now let us reverse the rightmost 8 bits, and preserve the leftmost 8 bits. This new bit pattern will be assigned to the unsigned integer variable b.
Answer:
Use exclusive or operation
When each of the rightmost 8 bits in a is **bitwise exclusive or** with the corresponding 1 in the mask, the resulting bit will be the opposite of the bit originally in a(INVERTED).
0 ^ 1=**1**
1 ^ 1=**0**

On the other hand, when each of the leftmost 8 bits in a is **bitwise exclusive or** with the corresponding 0 in the mask, the resulting bit will be the same as the bit originally in a.(COPIED)

0 ^ 0=**0**
1 ^ 0 =**1**

a = **0110 1101** *1011 0111* (0x 6db7) ^
mask = 0000 0000 1111 1111 (0xff)
………………………………...
b = **0110 1101***0100 1000* (0x6d48)

E.g Suppose that a is an unsigned integer variable whose value is Ox6db7
The expression
a^ 0x4
will invert the value of bit number 2 (the third bit from the right) in a. If this operation is carried out repeatedly, the value of **a** will alternate between Ox6db7 and Ox6db3. Thus, the repeated use of this operation will toggle the third bit from the right on and off.
The corresponding bit patterns are shown below.

Ox6db7 = 0110 1101 1011 0**1**11 ^
mask = 0000 0000 0000 0100 (**0x4**)
…………………………………….
Ox6db3 = 0110 1101 1011 0**0**11 ^
mask = 0000 0000 0000 0100 (**0x4**)
………………………………….
Ox6db7 = 0110 1101 1011 0**1**11

The Shift Operators

The two bitwise shift operators are
shift left (<<)
shift right (>>).
Each operator requires two operands.

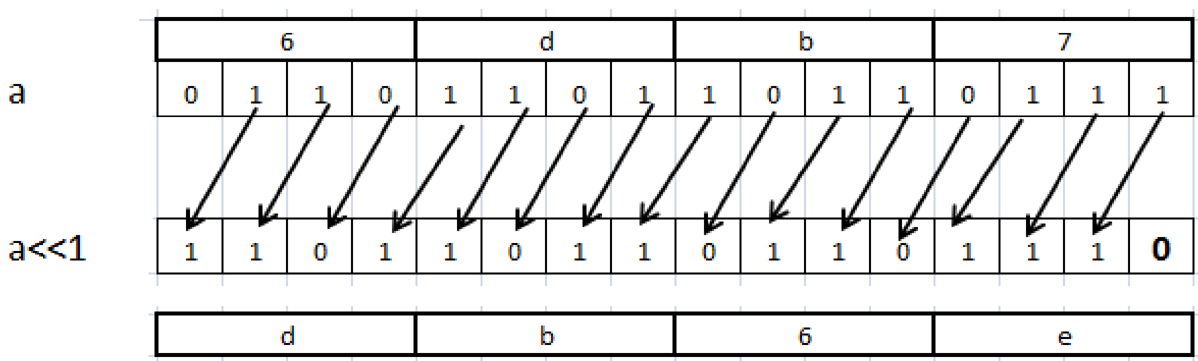
- The fust is an integer-type operand that represents the bit pattern to be shifted.
- The second is an unsigned integer that indicates the number of displacements (i.e., whether the bits in the first operand will be shifted by 1 bit position, 2 bit positions, 3 bit positions, etc.).
    - This value cannot exceed the number of bits in the first operand.

THE LEFT SHIFT OPERATOR <<

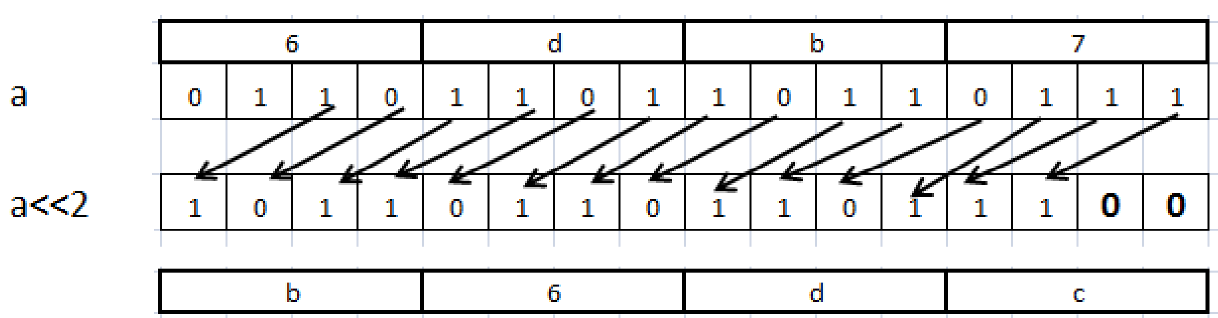
- causes all of the bits in the first operand to be shifted to the left by the number of positions indicated by the second operand.
- The leftmost bits (i.e., the overflow bits) in the original bit pattern will be lost.
- The rightmost bit positions after shifting that become vacant will be filled with Os.

Suppose a is an unsigned integer variable whose value is Ox6db7. The expression
b = a << 1;

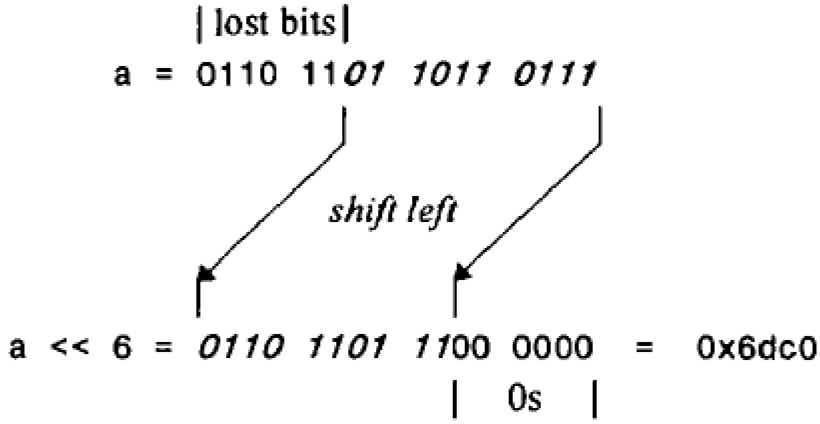
a = **0110 1101 1011 0111** (0x 6db7)
When a<<1 is done, bits in **a** are shifted one position towards left. Then leftmost bit in a is lost. Rightmost one position will be vacant after shifting and that will be filled with 0.
a = **0110 1101 1011 0111** (0x 6db7)
a<<1 **1101 1011 0110 1110**



When a<<2 is done, bits in **a** are shifted two position towards left. Then leftmost two bits in a are lost. Rightmost two position will be vacant after shifting and that will be filled with 0.



E.g.a=0x6db7

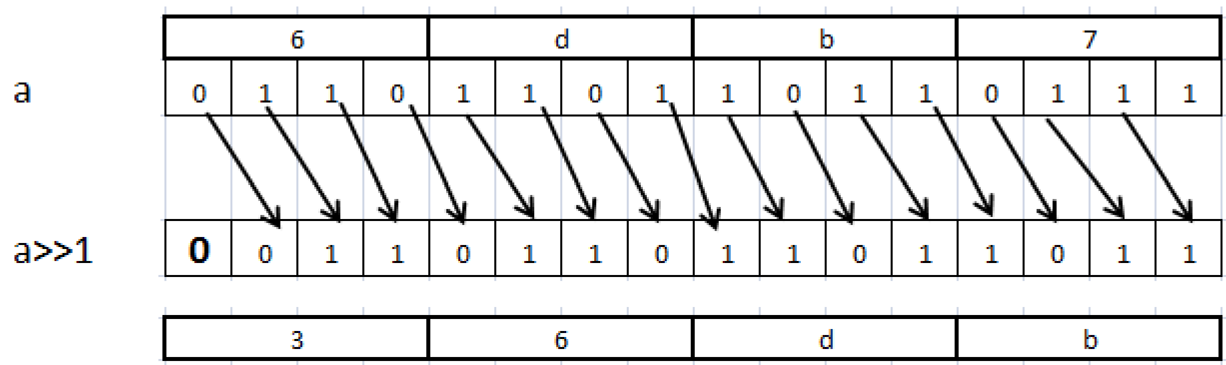


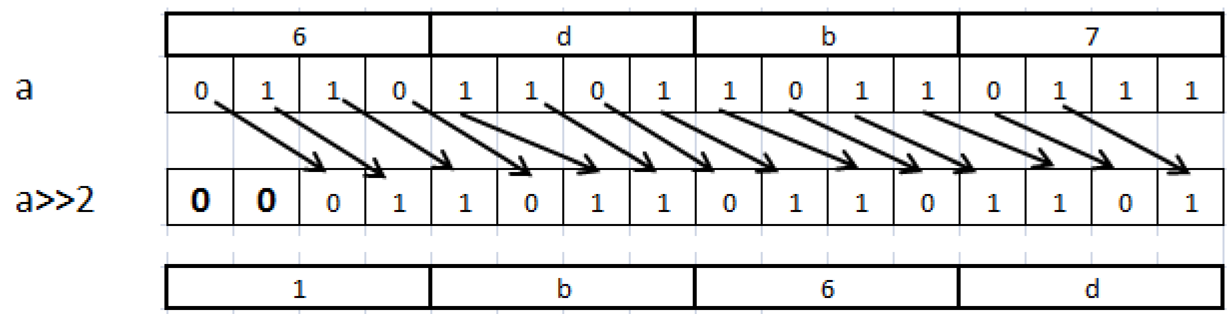
Here after a<<6 all bits are shifted 6 positions to left. Here six leftmost bits in a are lost and rightmost 6 vacant positions are filled with 0s

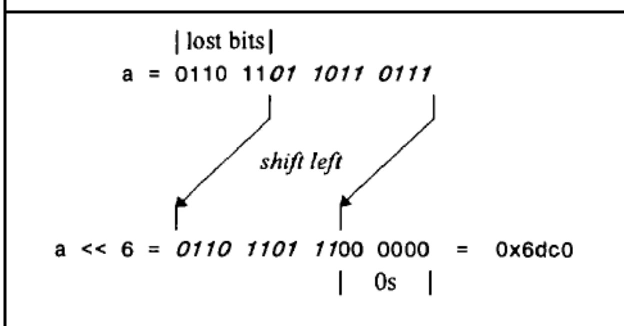
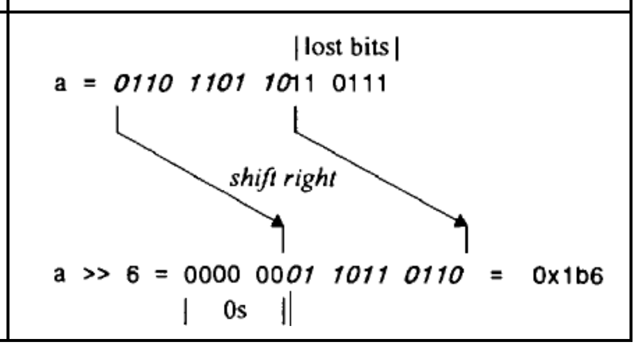
<u>RIGHT SHIFT OPERATOR >></u>

- The right shift operator causes all of the bits in the first operand to be <u>shifted to the right by the number of positions indicated by the second operand</u>.
- The rightmost bits (i.e., the underflow bits) in the original bit pattern will be lost.
- The leftmost bit positions of **<u>unsigned integer</u>** after shifting that become vacant will be filled with Os.
- If the bit pattern representing a **<u>signed integer</u>** is shifted to the right, the outcome of the shift operation may depend on the value of the leftmost bit (the sign bit).
    - Negative integers have a 1 in leftmost position,
        - When the signed negative integer is shifted to the right,the leftmost vacated bit positions are filled with 1s
    - positive integers have a 0 in leftmost position
        - When the signed positiveinteger is shifted to the right,the leftmost vacated bit positions are filled with 0s

Hence, the behavior of the right shift operator is similar to that of the left shift operator when the first operand is an **unsigned** integer





Shift left a=6db7 a<<6	**Shift right a=6db7 a>>6**
	

<u>Eg</u>

```
#include <stdio.h>
main( )
{
unsigned a = 0xf05a;
```

```
int b = a;
printf ( "%u %d\n", a, b) ;
printf ("%x\n", a >> 6 ) ;
printf ( “ % x \ n " , b >> 6 ) ;
}
```

Here a=0xf05a

Binary representation

a=0xf05a	1	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0
	f				0				5				a			

a=0xf05a	1	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	position

Unsigned decimal value=

$1x2^{15}+1x2^{14}+1x2^{13}+1x2^{12}+0x2^{11}+0x2^{10}+0x2^{9}+0x2^{8}+0x2^{7}+1x2^{6}+0x2^{5}+1x2^{4}+1x2^{3}+0x2^{2}+1x2^{1}+0x2^{0}$ =61350

Signed value =2s complement

a=0xf05a	1	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0
1s comp	0	0	0	0	1	1	1	1	1	0	1	0	0	1	0	1
2s comp=1s comp+1	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	0

Since leftmost bit in a is 1, sign is negative

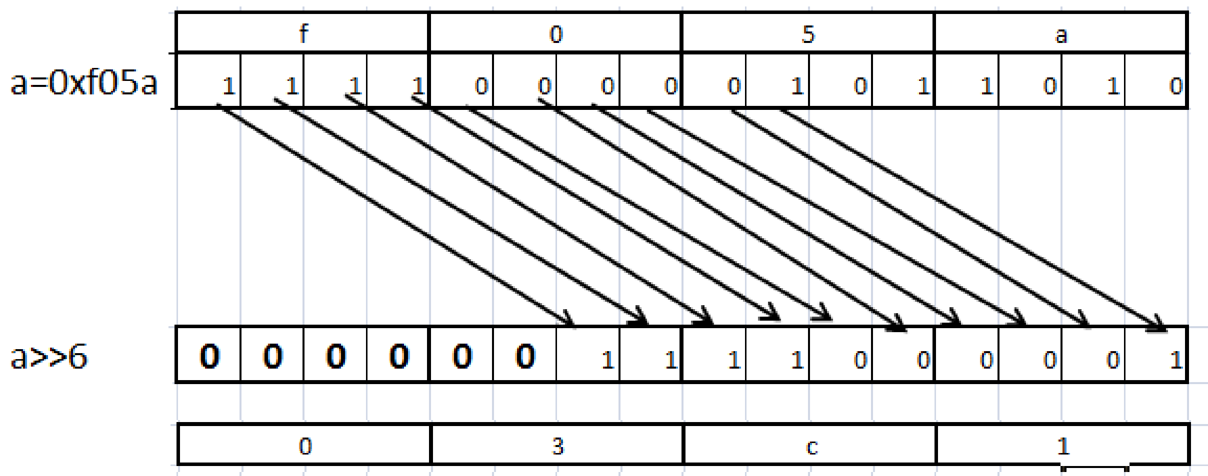
$0x2^{15}+0x2^{14}+0x2^{13}+0x2^{12}+1x2^{11}+1x2^{10}+1x2^{9}+1x2^{8}+1x2^{7}+0x2^{6}+1x2^{5}+0x2^{4}+0x2^{3}+1x2^{2}+1x2^{1}+0x2^{0}$ = -4006

Here a=0xf05a

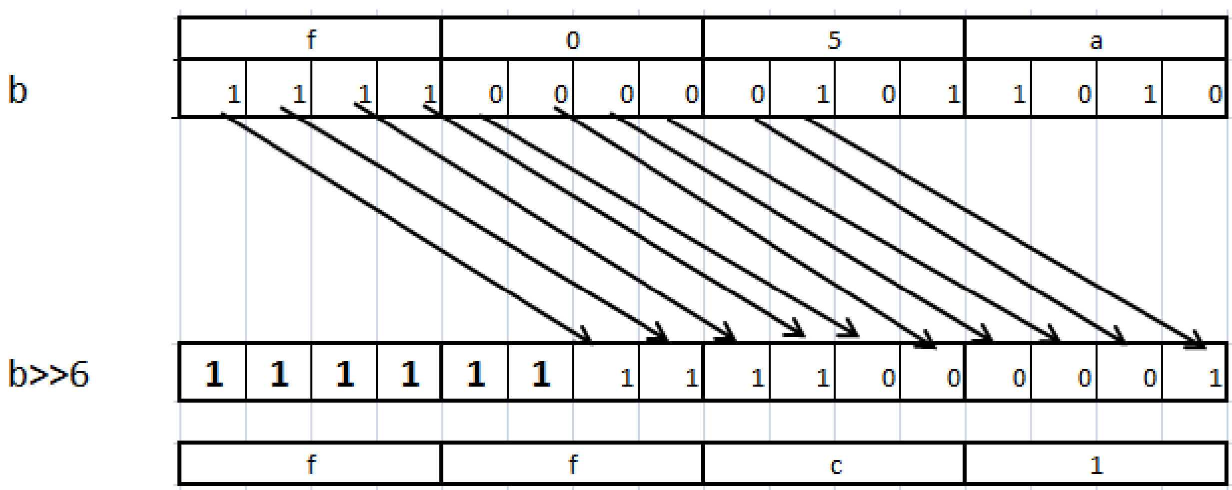
Here a is unsigned its decimal(printed using %u) value is 61530.

Since b=a, b stores the value of a

Here b is signed(printed using %d) it is -4006



After a>>6 ll bits in a are shifted 6 positions towards right. Since a is **<u>unsigned</u>** integer the vacant 6 leftmost positions are filled with **0**s.



After b>>6 all bits in b are shifted 6 positions towards right. Since b is **<u>signed</u>** integer the vacant 6 leftmost positions are filled with **1**s.

THE BITWISE ASSIGNMENT OPERATORS

C also contains the following bitwise assignment operators.

&= ^ = |= <<= >>=

The left operand must be an assignable integer-type identifier (e.g., an integer variable), and the right operand must be a bitwise expression.

Associativity is **RIGHT to LEFT**

a &= 0x7f is equivalent to a = a & 0x7f.

The bitwise assignment operators are members of the same precedence group as the other assignment operators in C.

Precedence of operators
======================

<table>
<tr><th>Category</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>Postfix</td><td>() [] -> . ++ - -</td><td>Left to right</td></tr>
<tr><td>Unary</td><td>+ - ! ~ ++ - - (type) * & sizeof</td><td>Right to left</td></tr>
<tr><td>Multiplicative</td><td>* / %</td><td>Left to right</td></tr>
<tr><td>Additive</td><td>+ -</td><td>Left to right</td></tr>
<tr><td>Shift</td><td><< >></td><td>Left to right</td></tr>
<tr><td>Relational</td><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td>Equality</td><td>== !=</td><td>Left to right</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>Left to right</td></tr>
<tr><td>Bitwise XOR</td><td>^</td><td>Left to right</td></tr>
<tr><td>Bitwise OR</td><td>|</td><td>Left to right</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>Left to right</td></tr>
<tr><td>Logical OR</td><td>||</td><td>Left to right</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Right to left</td></tr>
<tr><td>Assignment</td><td>= += -= *= /= %= >>= <<= &= ^= |=</td><td>Right to left</td></tr>
<tr><td>Comma</td><td>,</td><td>Left to right</td></tr>
</table>

<table>
<tr><th colspan="2">Right to Left Associativity</th></tr>
<tr><td>~</td><td>Bitwise 1s Complementation</td></tr>
<tr><td>&= |= ^= >>= <<=</td><td>Bitwise assignment operators</td></tr>
</table>