Windows Programming

Lecture 06

Data Types Classification

- Data types are classified in two categories that is,
  - those data types which stores decimal numbers
  - those do not stores decimal numbers
    - Known as Integral data types
    - For example, integer , char, long integer etc.

Bitwise Operators

- An operator that manipulates individual bits. The operators that most people are familiar with, such as the addition operator (+), work with bytes or groups of bytes. Occasionally, however, programmers need to manipulate the bits within a byte. The C programming language supports the following bitwise operators:

- **>>** Shifts bits right
- **<<** Shifts bits left
- **&** Does an AND compare on two groups of bits
- **|** Does an OR compare on two groups of bits
- **^** Does an XOR compare on two groups of bits
- ~ Complements a group of bits

1 - set / on

0 - reset / off

Bitwise Shift Operators

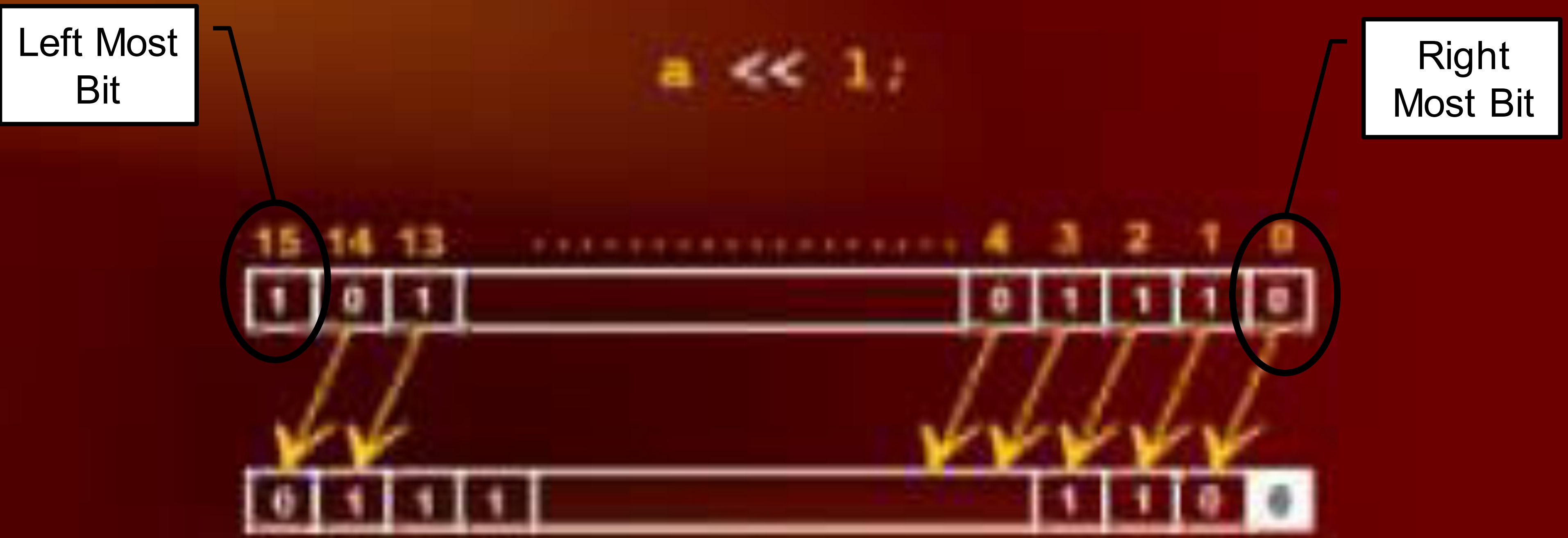
There are two bitwise shift operators, namely shift left and shift right. In C, they are represented by the **<<** and **>>** operators, respectively. These operations are very simple, and do exactly what they say: shift bits to the left or to the right.

Bitwise Left Shift (<<) Operator

Bitwise Left-Shift is useful when to want to MULTIPLY an integer (not floating point numbers) by a power of 2. The operator, like many others, takes 2 operands like this:

```
a << b
```

This expression returns the value of a multiplied by 2 to the power of b.


Left Most Bit
a << 1;
Right Most Bit
15 14 13
4 3 2 1 0
1 0 1
0 1 1 1 0
0 1 1 1
1 1 0 0

Bitwise Left Shift (<<) Operator

Why is it called a left shift?

Take the binary representation of a, and add b number of zeros to the right, consequently "shifting" all the bits b places to the left.

Example: **`4 << 2.`**

4 is 100 in binary. Adding 2 zeros to the end gives 10000, which is 16,

i.e. **`4*2*2 = 4*4 = 16.`**

What is 4 << 3 ? Simply add 3 zeros to get 100000, which is **`4*23 = 4*8 = 32.`**

Bitwise Right Shift (>>) Operator

Bitwise Right-Shift does the opposite, and takes away bits on the right. Suppose we had:

```
a >> b
```

This expression returns the value of a divided by 2 to the power of b.

Example: `8 >> 2.`

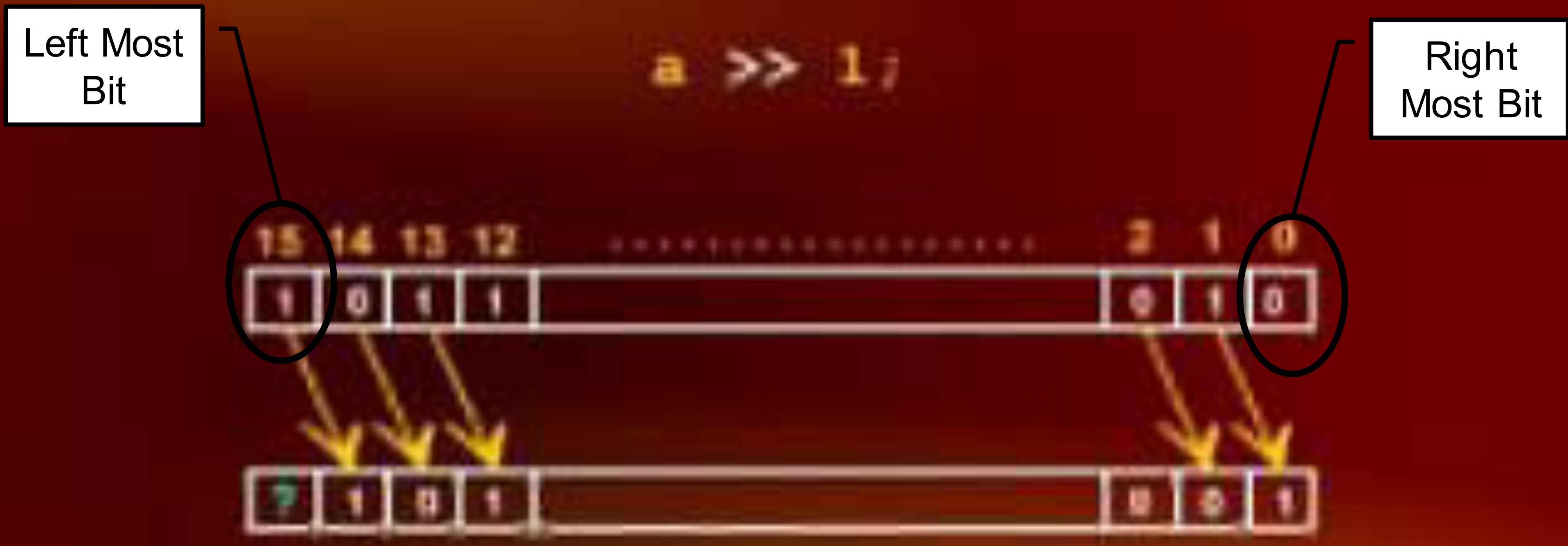
8 is 1000 in binary. Performing a right shift of 2 involves knocking the last 2 bits off: 1000, which leaves us with 10, i.e. 2.

`8 >> 2` is the as doing `8/2*2 = 8/4 = 2.`

But what happens if we had a left operand that's not a power of 2? Let's try `9 >> 2.`

9 is 1001. Now take off the last 2 bits, leaving us with 10, which is 2. But this does make sense, since `9/4 = 2.25`, which rounds down to 2.

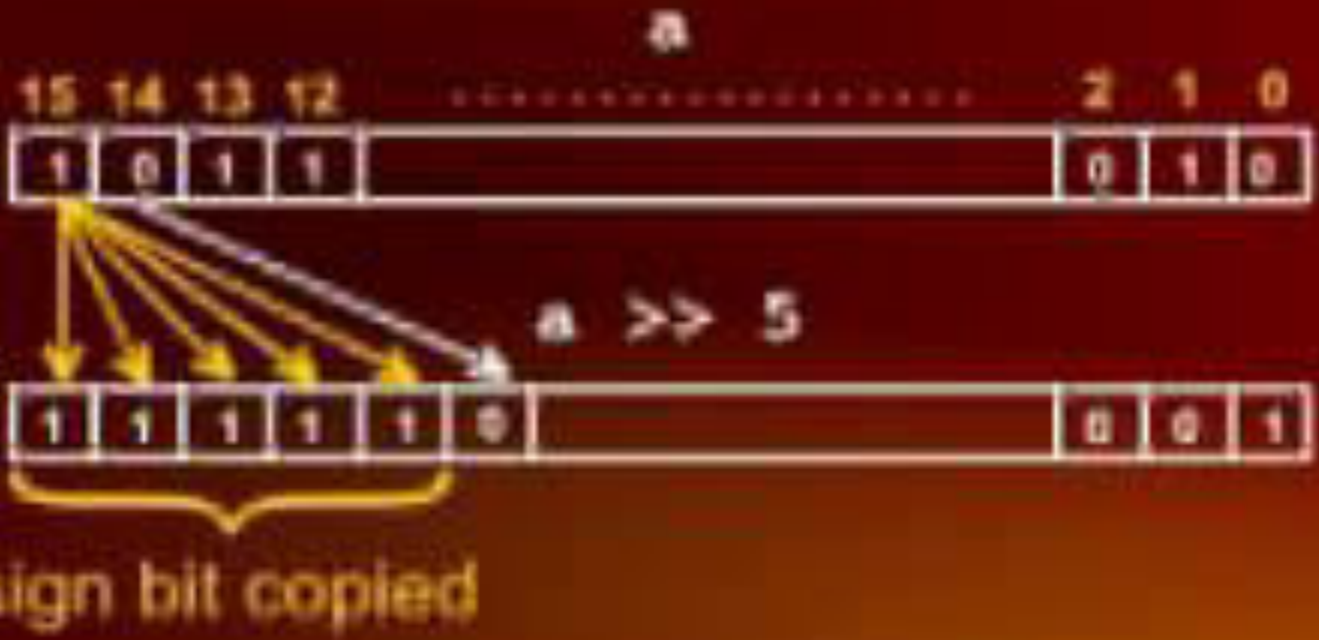
```
signed short int a;

a >> 1;
```



Examples

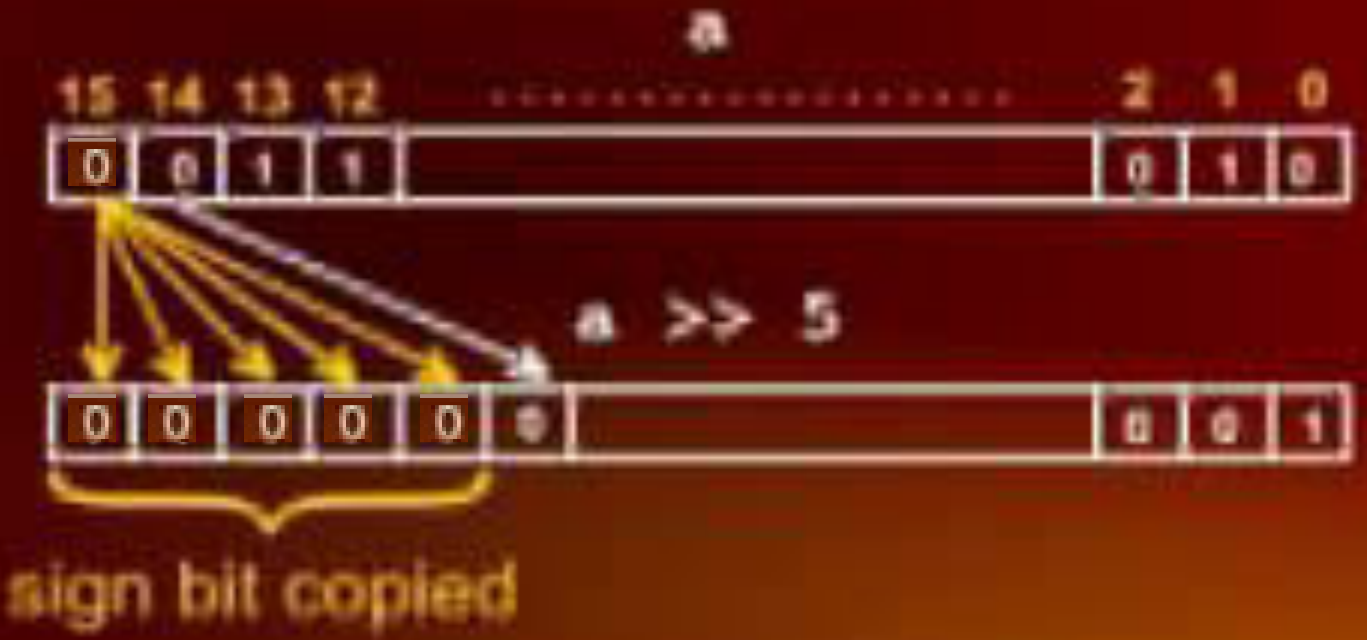
Right shifting of signed numbers

```
signed short int a;
```



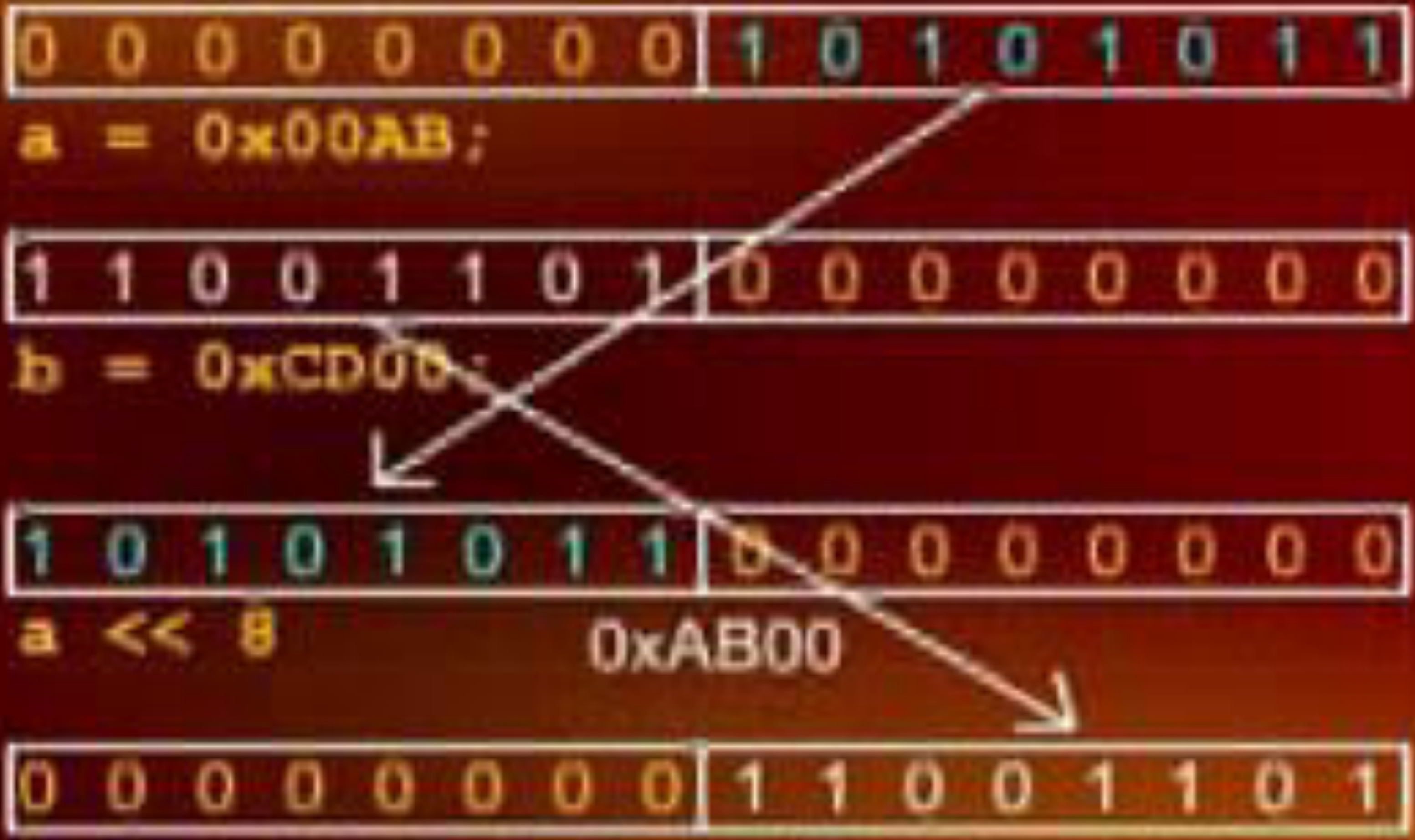
Right shifting of signed numbers

```
signed short int a;
```

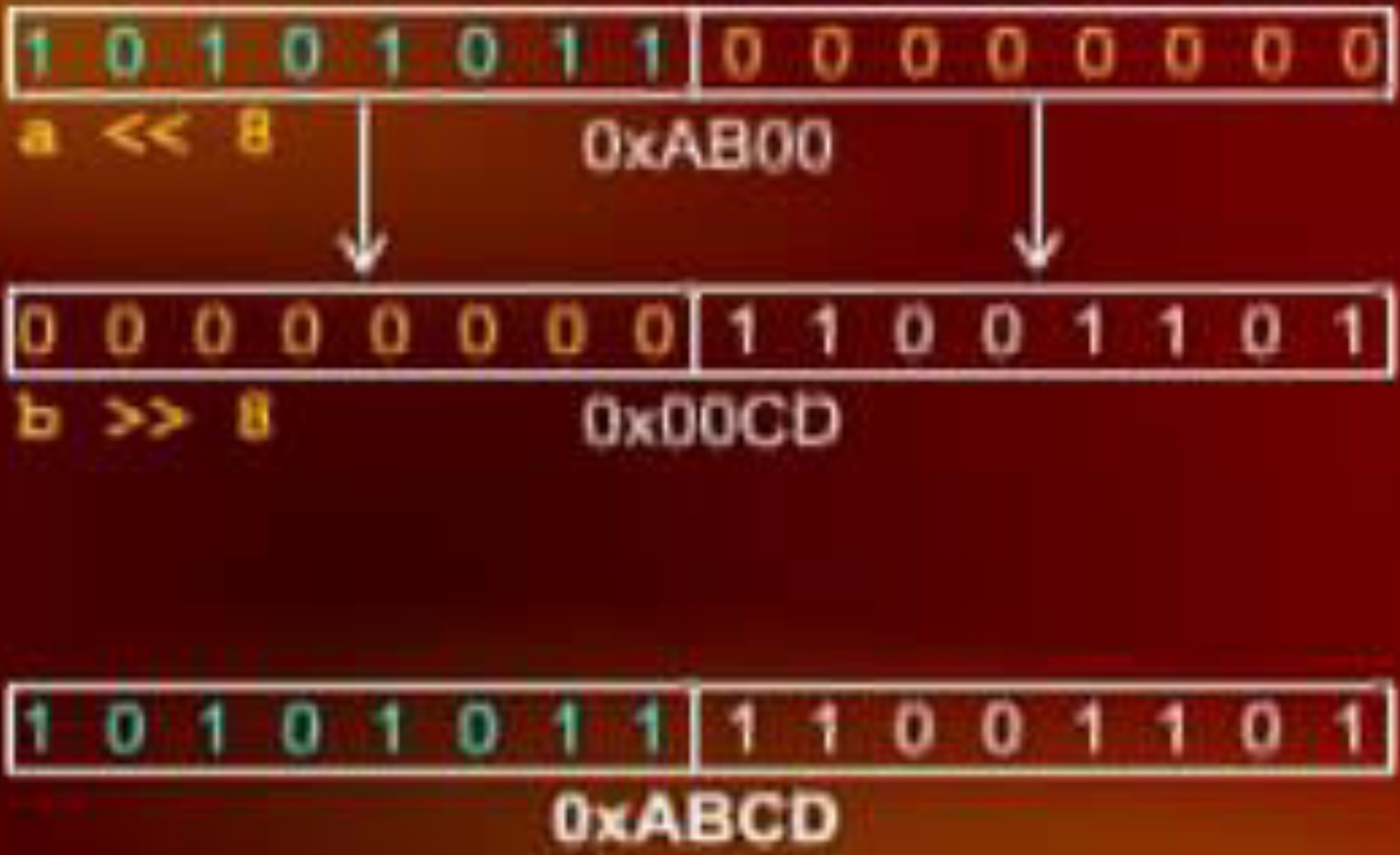


Bitwise shift operations can only applied to Integral data types

```
unsigned int a, b, c;
```



```
b >> 8
```
0x00CD


1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0
a << 8
0xAB00
0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1
b >> 8
0x00CD
1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1
0xABCD

```
c = ( a << 8 ) + ( b >> 8 );
```

Bitwise AND, Bitwise Inclusive OR and Bitwise Exclusive OR can only use Integral operands

AND, OR, NOT

There are three major bitwise operators that can be used to combine two numbers: AND, OR, and XOR. When I say that an operator is *bitwise*, it means that the operation is actually applied separately to each bit of the two values being combined. For example, if x = y AND z, that means that bit 0 of x is actually bit 0 of y ANDed with bit 0 of z. Make sense?

Bitwise AND Operator

There are two kinds of AND operators in the C language: the logical AND, and the bitwise AND. The former is represented by the `&&` operator, and the latter uses the `&` operator.

The bitwise AND works very much the same way, except that it works with two bits instead of two expressions. The bitwise AND operation returns 1 if and only if *both* of its operands are equal to 1. In other words, we have the following truth table for the bitwise AND.

Bitwise AND Operator

```
0 AND 0 = 0
0 AND 1 = 0        x AND 0 = 0
1 AND 0 = 0        x AND 1 = x
1 AND 1 = 1
```

Bitwise OR Operator

Just as with the AND operation, there are two different types of OR in the C language. The logical OR uses the || operator, and the bitwise OR uses the | operator.

The bitwise OR is very similar, in that it returns 0 if and only if *both* of its operands are 0.

Bitwise OR Operator

```
0 OR 0 = 0
0 OR 1 = 1        x OR 0 = x
1 OR 0 = 1        x OR 1 = 1
1 OR 1 = 1
```

XOR Operator

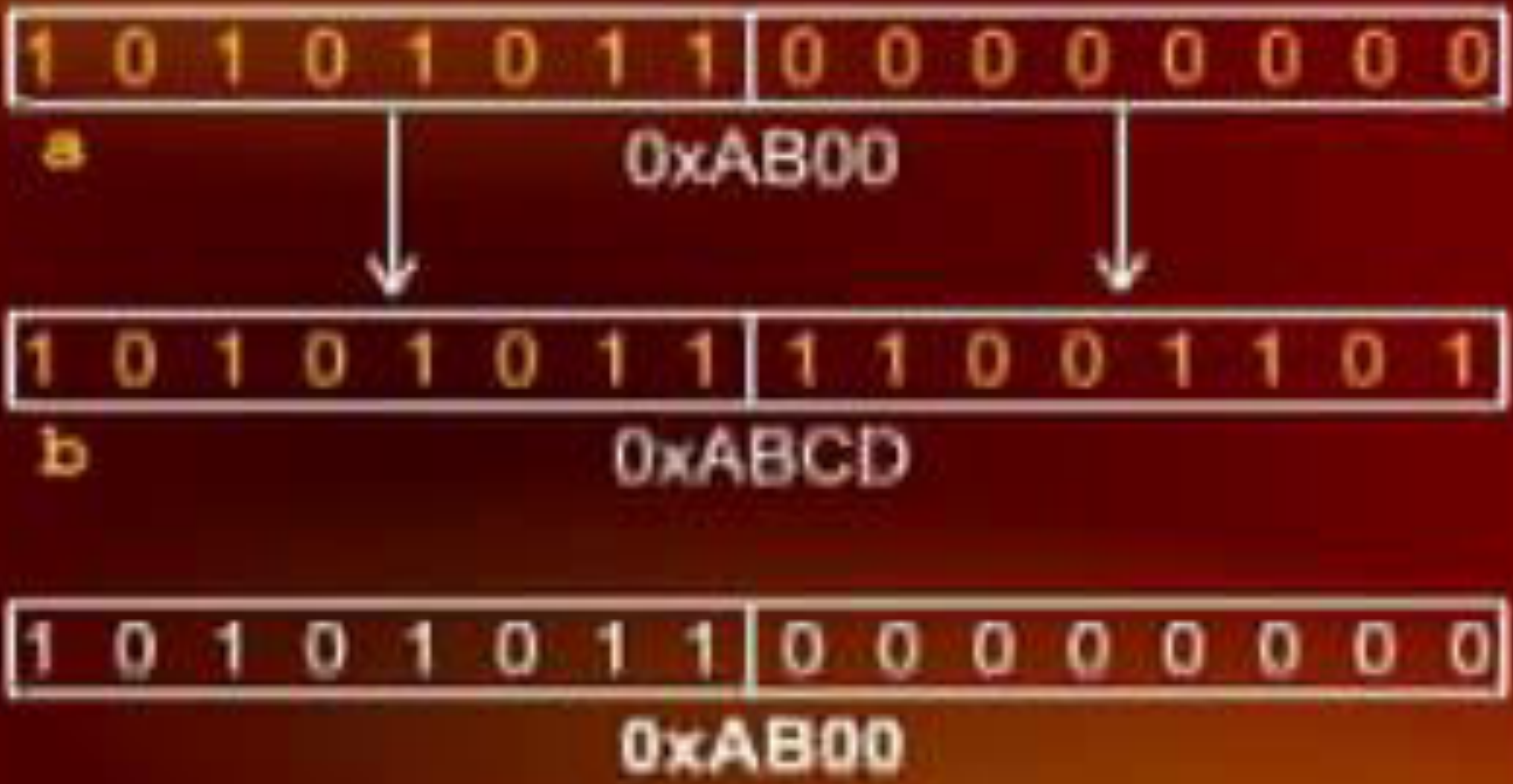
The XOR is a little strange because there is no logical equivalent for it in C, even though many languages include one. The XOR operation is symbolized by the ^ character in C. The term XOR stands for "exclusive OR“, and means "one or the other, but not both." In other words, XOR returns 1 if and only if *exactly one* of its operands is 1. If both operands are 0, or both are 1, then XOR returns 0.

XOR Operator

```
0 XOR 0 = 0
0 XOR 1 = 1        x XOR 0 = x
1 XOR 0 = 1        x XOR 1 = ~x
1 XOR 1 = 0
```

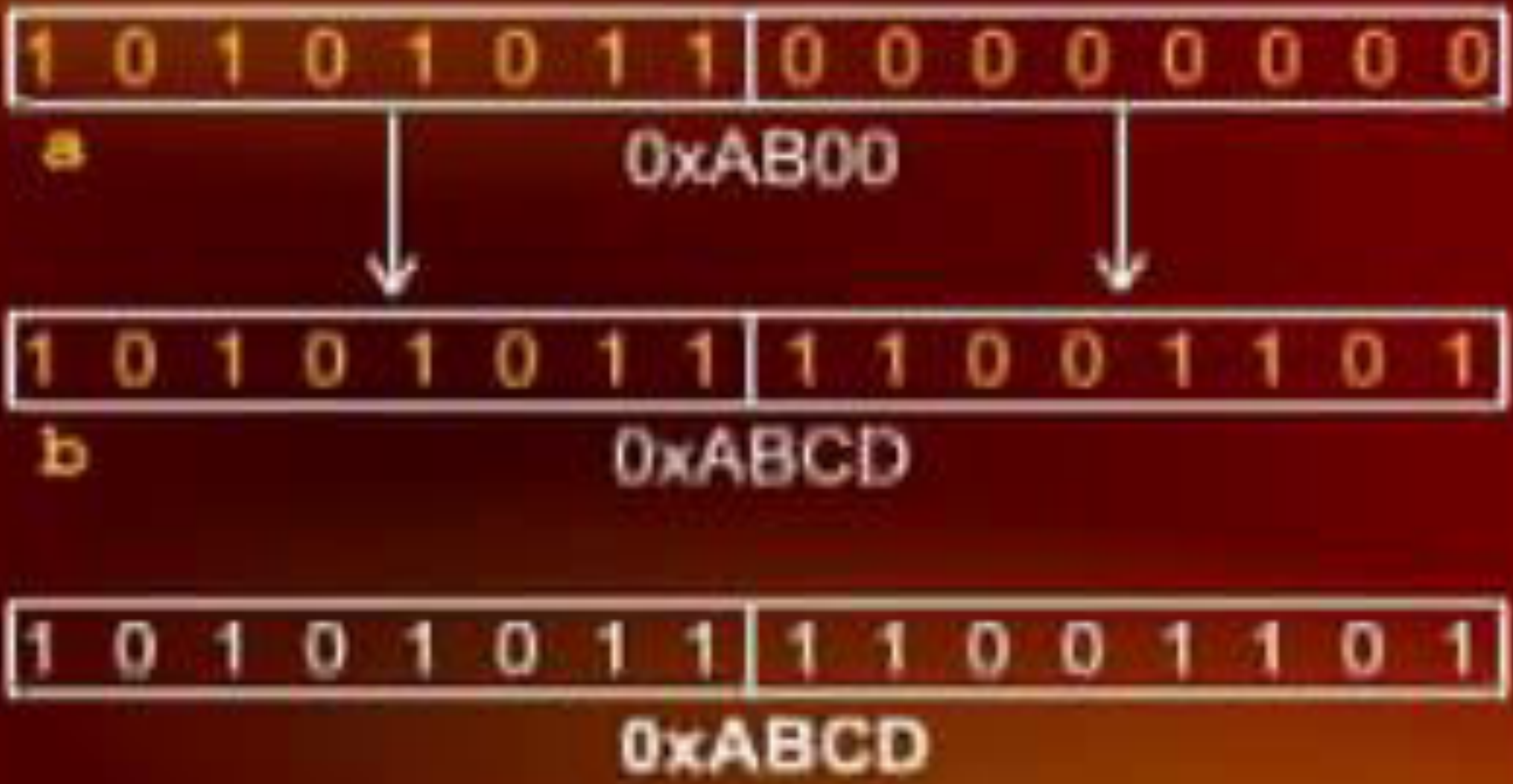
Examples

```
unsigned short a=0xAB00, b=0xABCD, c;
```



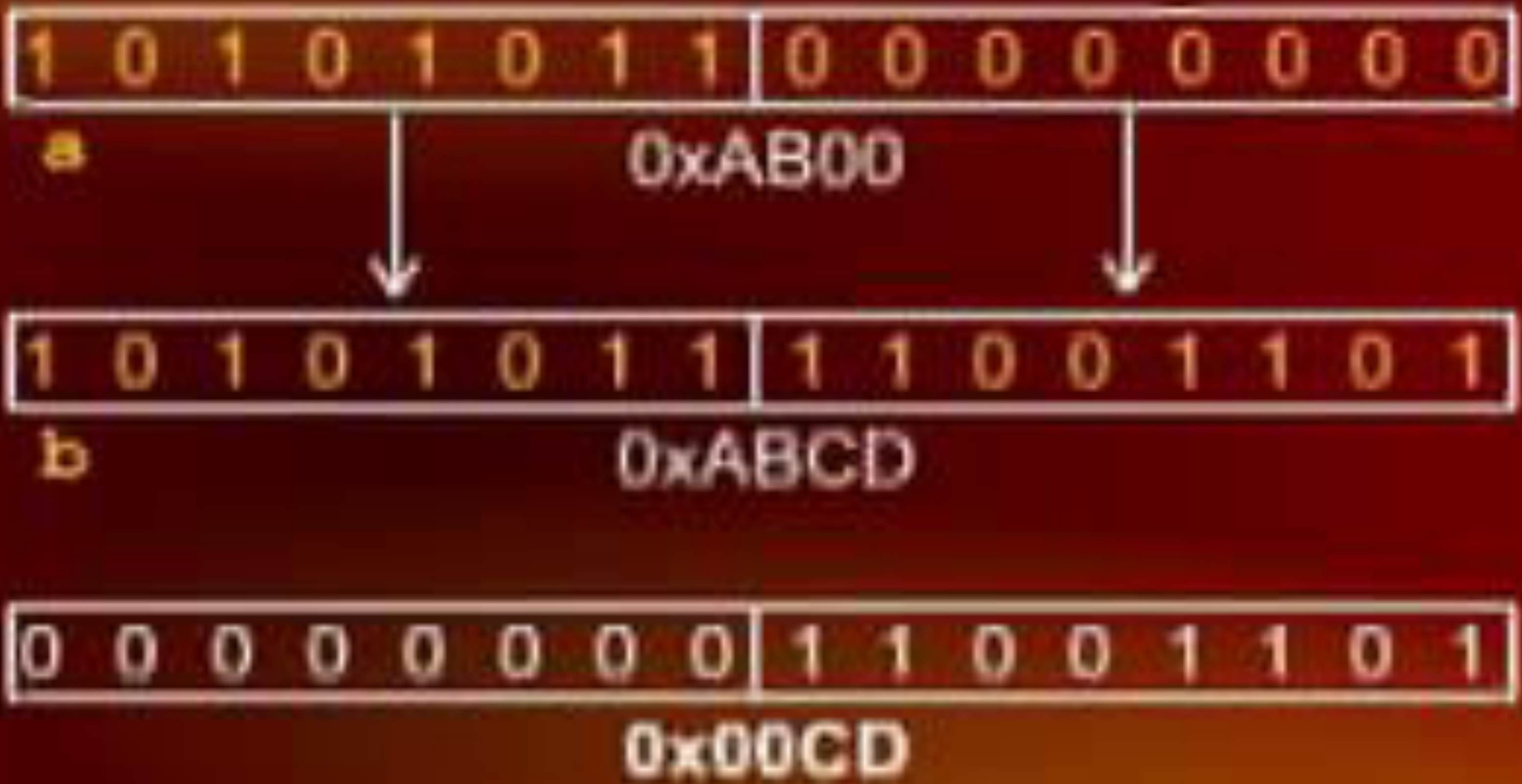
```
c = a & b;
```

```
unsigned short a=0xAB00, b=0xABCD, c;
```



```
c = a | b;
```

```
unsigned short a=0xAB00, b=0xABCD, c;
```



```
c = a ^ b;
```

Application of Bitwise Operators

Bitwise operators have two main applications. The first is using them to combine several values into a single variable. Suppose you have a series of flag variables which will always have only one of two values: 0 or 1 (this could also be true or false). The smallest unit of memory you can allocate to a variable is a byte, which is eight bits. But why assign each of your flags eight bits, when each one only needs one bit? Using bitwise operators allows you to combine data in this way.

typedef

```
typedef unsigned long DWORD
typedef int BOOL
typedef unsigned char BYTE
typedef unsigned short WORD
typedef char CHAR
typedef CHAR *LPSTR
typedef const CHAR *LPCSTR
```

typedef unsigned long DWORD;

$$\begin{tikzcd} {} \arrow[d] \\ \text{4 bytes} \arrow[d] \\ \text{32 bits} \arrow[d] \\ \text{DWORD} \end{tikzcd}$$

- Single Word =16 bits
- Double Word = 32 bits

typedef

```
typedef unsigned long ULONG;
typedef ULONG *PULONG;
typedef unsigned short USHORT;
USHORT  *PUSHORT;
```

Typecasting

```
char  ch;
int i;
…  …  …
some code here
…  …  …
i = ch;         // no problem
ch = i;         // compiler warning
```

Typecasting

Typecasting is making a variable of one type, act like another type for one single application.

There are two types of typecasting.

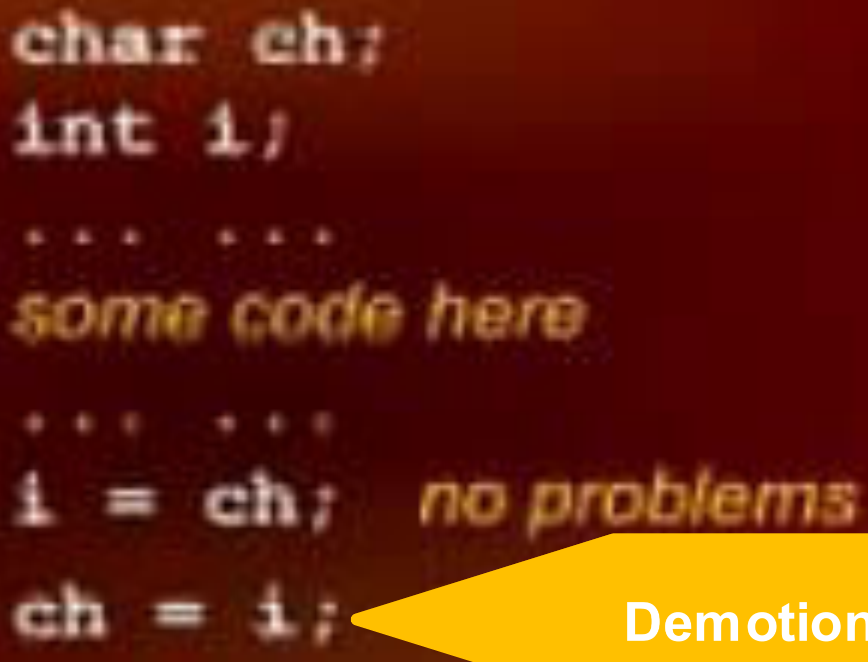
- Implicit typecasting (coercion)
- Explicit typecasting

Implicit type casting(coercion) is further divided in two types.

- Promotion
- Demotion

```
char ch;
int i;
...  ...
some code here
...  ...
i = ch;
```

Promotion Typecasting



```
char ch;
int i;
... ...
some code here
... ...
i = ch;   no problems
ch = i;
```

Demotion Typecasting

Macros

```
#define LOWORD(l)   ( (WORD)(l) )
#define HIWORD(l)   ( (WORD)( ( (DWORD)(l) >> 16)&0xFFFF) )

int  b=2623;             // 0xA3F
HIWORD(2623)             // call to macro

#define LOBYTE(w)    ( (BYTE)(w) )
#define HIBYTE(w)    ( (BYTE)( ( (WORD)(w) >> 8) & 0xFF) )
```

assertions

In C, assertions are implemented with the standard *assert* macro. The argument to *assert* must be **true** when the macro is executed, otherwise the program aborts and prints an error message. For example, the assertion

```
assert( size <= LIMIT );
```

will abort the program and print an error message like this:

```
Assertion violation: file tripe.c, line 34: size <= LIMIT
```

if size is greater than LIMIT.

Turning assertions off

By default, ANSI C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the NDEBUG flag to your compiler.

Turning assertions off

```
#ifdef  NDEBUG
   #define assert(exp)      ((void)0)
#else
   // original definition of assert ...
#endif
```

The `switch` and `case` keywords

- The **switch-case** statement is a multi-way decision statement. Unlike the multiple decision statement that can be created using if-else, the **switch** statement evaluates the conditional expression and tests it against numerous constant values. The branch corresponding to the value that the expression matches is taken during execution.

- The value of the expressions in a switch-case statement must be an (integer, char, short, long), etc. Float and double are not allowed.

The **`switch`** and **`case`** keywords

The syntax is :

```
switch( expression )
{
        case constant-expression1:
             statements1;
        case constant-expression2:
             statements2;
        case constant-expression3:
             statements3;
        default :
             statements4;
}
```

The `switch` and `case` keywords

- The **case** statements and the **default** statement can occur in any order in the **switch** statement. The **default** clause is an optional clause that is matched if none of the constants in the **case** statements can be matched.

- Multiple case labels together are possible. In the 'C' **switch** statement, execution continues on into the next case clause if it is not explicitly specified that the execution should exit the **switch** statement.