Programación de dispositivos electrónicos

Unidad 5

Operaciones con bits-Campos de bits-Directivas al compilador

Héctor Albano Diego Codevilla

Operaciones con bits

Podemos hacer desde C operaciones entre bits:

Operación	Resultado	En C	Ejemplo
Complemento (not)	Cambiar 0 por 1 y viceversa	~	~1100 = 0011
Producto lógico (and)	1 sólo si los 2 bits son 1	&	1101 & 1011 = 1001
Suma lógica (or)	1 sólo si uno de los bits es 1	|	1101 | 1011 = 1001
Suma exclusiva (xor)	1 sólo si los 2 bits son distintos	^	1101 ^ 1011 = 0110
Desplazamiento a la izquierda	Desplaza y rellena con ceros	<<	1101 << 2 = 110100
Desplazamiento a la derecha	Desplaza y rellena con ceros	>>	1101 >> 2 = 0011

```
#include <stdio.h>
void binario(char nbits,int x);
int main()
{   int a = 67;
    unsigned int ab = 67;
    int abc = -67;
    int b = 33;
    unsigned int var = 64;
    printf("\nCaracter: %c\n", var);
    printf("Decimal: %u\n", var);
    printf("Octal: %o\n", var);
    printf("Hexadecimal: %x\n", var);
    printf("Binario: ");
    binario(8,var);
    printf("\n\nLa variable a vale %d\n", a);
    printf("La variable ab vale %d\n", ab);
    printf("La variable abc vale %d\n", abc);
    printf("La variable b vale %d\n", b);
    printf("\nEl complemento de a es: %d\n", ~a);
    printf("El producto logico de a y b es: %d\n", a&b);
    printf("Su suma logica es: %d\n", a|b);
    printf("Su suma logica exclusiva es: %d\n", a^b);
    printf("Desplacemos a a la izquierda: %d\n", a << 1);
    printf("Desplacemos a a la derecha: %d\n", a >> 1);
    printf("Desplacemos a abc la derecha: %d\n", abc >> 1);
    printf("Desplacemos a ab la derecha: %d\n", ab >> 1);
    return 0;}
```

```
void binario(char nbits, int x)
{
 unsigned int contador,inicio;
 if(nbits==8)
    inicio=128;
if(nbits==16)
    inicio=32768;
 for(contador = inicio; contador > 0; contador >>= 1)
                if(contador & x)
                printf("1");
                else
                printf("0");
}
```

NOT

A	NOT A
0	1
1	0

El NOT bit a bit o complemento, es una operación unaria que realiza la negación lógica en cada bit, invirtiendo los bits del número, de tal manera que los ceros se convierten en 1 y viceversa. Por ejemplo:

- El NOT forma el complemento a uno de un valor binario dado.
- En un número entero con signo en complemento a dos, el NOT da como resultado el inverso aditivo del número menos 1, es decir NOT x = -x - 1. Para obtener el complemento a dos de un número, se debe sumar 1 al resultado, dando el negativo del número. Esto equivale a un cambio de signo del número: +5 se convierte en -5, y -5 se convierte en +5.
- Para los enteros sin signo, el complemento bit a bit es la “reflexión de espejo” del número a través del punto medio del rango del entero. Por ejemplo, para los enteros sin signo de 8 bits, NOT x = 255 - x, para los enteros sin signo de 16 bits, NOT x = 65535 - x, y en general, para los enteros sin signo de n bits, NOT x = (2n - 1) - x.

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

El AND bit a bit toma dos números enteros y realiza la operación AND lógica en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de los dos operandos es 1, y 0 de lo contrario, por ejemplo:
El AND puede ser usado para filtrar determinados bits, permitiendo que unos bits pasen y los otros no. También puede usarse en sistemas de mayor fiabilidad.

Determinando el estado de bits

El AND puede ser usado para determinar si un bit particular está encendido (1) o apagado (0). Por ejemplo, dado un patrón de bits 0011, para determinar si el segundo bit está encendido se usa una operación AND con una máscara que contiene encendido solo el segundo bit, que es el que se quiere determinar, si el resultado es diferente de cero, se sabe que el segundo bit está encendido. Esto es a menudo llamado enmascaramiento del bit.

Extrayendo bits

El AND se puede usar para extraer determinados bits de un valor. Si en un byte, por ejemplo, tenemos representados dos dígitos hexadecimales empaquetados, (uno en los 4 bits superiores y el otro en los 4 bits inferiores), podemos extraer cada dígito hexadecimal usando el AND con las máscaras adecuadas:

```
    0011 0101                    0011 0101
AND 1111 0000  (máscara)     AND 0000 1111  (máscara)
  = 0011 0000                  = 0000 0101
    Hex. superior                Hex. inferior
```

Apagando bits

El AND también se puede usar para apagar determinados bits. Solo hay que poner una máscara con bits en cero en las posiciones de los bits que se quieren apagar y 1 en los demás bits. Todos los demás bits con la máscara 1 pasarán inalterados, y los que tienen la máscara 0 se apagarán.

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Una operación OR de bit a bit toma dos números enteros y realiza la operación OR en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de cualquiera de los dos operandos es 1, y 0 si ambos bits son 0.

Encendiendo bits

El OR bit a bit puede ser usado para encender un bit individual o un conjunto de bits. Para ello se usa una máscara OR con los bits que se quieren encender en 1 y el resto de los bits en cero. El resultado será que todos los bits originales quedarán como estaban excepto los bits en donde la máscara tenga 1, que resultarán encendidos.

Copiando bits

El OR, y el desplazamiento lógico puede ser usado para copiar un grupo de bits a una posición determinada. Supongamos que tenemos el signo, el exponente, y la parte significativa de un número, en diferentes registros de 32 bits, y queremos empaquetarlos para formar un número en representación de punto flotante de simple precisión de 32 bits:

```
            Signo: 00000000000000000000000000000001
       Exponente: 00000000000000000000000010000011
Parte significativa: 00000000111000001110000000001110
```

Todos ellos tienen los valores correctos y tenemos que mover cada uno de ellos a su posición para poder armar el punto flotante. Se debe mover el signo 31 posiciones hacia la izquierda, el exponente 23 posiciones hacia la izquierda, y la parte significativa no es necesaria moverla porque ya está en la posición correcta. Estos desplazamientos se hacen con la operación de desplazamiento hacia la izquierda.

Ahora que tenemos cada parte del número en su lugar, las combinamos para empaquetarlas y formar el número en su representación de punto flotante de 32 bits. Para ello usamos el OR: (Resultado final) = (Signo) OR (Exponente) OR (Parte significativa):

```
            Signo: 10000000000000000000000000000000
       Exponente: 01000001100000000000000000000000
Parte significativa: 00000000111000001110000000001110
    Resultado final: 11000001111100001110000000001110
```

Ya tenemos el número en su representación de punto flotante definitiva

Procedimiento genérico para copiar un grupo de bits

Para copiar una serie de bits en un lugar determinado usando OR, se necesita que ese lugar donde se van a copiar tenga sus bits en cero (para hacer un espacio libre para poder copiar los bits). También se necesita que el registro donde se encuentran los bits que se quieren copiar tenga los demás bits (los que no se quieren copiar) apagados. Ambas operaciones, aclarar los bits en el lugar del destino, y aclarar los bits que no se quieren copiar se hacen con AND. Tenemos dos registros de 16 bits:

```
Registro A: 1011 1100 0110 1100
Registro B: 1001 0001 1111 1010
```

Queremos copiar los cuatro bits menos significativos del registro A en el registro B. Para ello, primero aclaramos los 4 bits menos significativos de B con una operación AND, y así tener un espacio libre:

```
    1001 0001 1111 1010   <-- Valor original del registro B
AND 1111 1111 1111 0000   <-- Máscara para aclarar los bits de B donde se van a copiar los que vienen de A
    1001 0001 1111 0000   <-- Registro B preparado para recibir los 4 bits menos significativos de A
```

Luego, aclaramos los bits de A que no queremos copiar, dejando solo los bits que queremos copiar:

```
    1011 1100 0110 1100   <-- Valor original del registro A
AND 0000 0000 0000 1111   <-- Máscara para dejar solo los bits de A que se quieren copiar
    0000 0000 0000 1100   <-- Registro A con solo los bits que se desean copiar
```

Ahora estamos listos para hacer el OR de A sobre B y combinar los 4 bits menos significativos de A sobre B:

```
   0000 0000 0000 1100   <-- Registro A con los 4 bits que se desean copiar
OR 1001 0001 1111 0000   <-- Registro B con un espacio para los 4 bits que desean copiar
   1001 0001 1111 1100   <-- Registro B con los 4 bits menos significativos de A copiados sobre él
```

Ahora, el registro B tiene copiado los 4 bits menos significativos de A. El resto de los bits de B quedaron intactos.

XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

El XOR bit a bit toma dos números y realiza la operación OR exclusivo en cada par correspondiente de bits. El resultado en cada posición es 1 si el par de bits son diferentes y cero si el par de bits son iguales.

Invirtiendo bits selectivamente

A diferencia del NOT, que invierte todos los bits de un operando, el XOR bit a bit puede ser usado para invertir selectivamente uno o más bits en un registro. Dado el patrón de bits 0011, el segundo y el cuarto bit pueden ser invertidos por XOR con una máscara con un patrón de bits conteniendo 1 en las posiciones que se quieren invertir, la segunda y cuarta, y 0 en las demás. Los bits de las posiciones con cero de la máscara resultarán inalterados.

Igualdad y desigualdad de bits

El XOR puede usarse para saber si los bits correspondientes de dos operandos son iguales o diferentes. Por ejemplo, si tenemos dos operandos, 1000 y 0010 y queremos saber si los bits más significativos de ambos son iguales, hacemos la XOR entre ambos luego cada bit del resultado estará en 0 si el bit correspondiente de los dos operandos son iguales, y en 1 si son diferentes. En este caso el bit mas significativo será 1 indicando que son diferentes, pero tenemos que aislarlo de los demás con un AND para poder usarlo o tomar una decisión.

Asignar cero a un registro

Los programadores avanzados de lenguaje ensamblador usan XOR como una manera eficiente y rápida de asignar cero a un registro. Realizar XOR de un valor contra sí mismo siempre resulta en cero (`A XOR A` siempre es cero), y en muchas arquitecturas esta operación requiere menos ciclos de reloj y/o memoria que cargar un valor cero a un registro (`A = 0`).

Uniones de bits

Conocemos lo que es un struct: un dato formado por varios "trozos" de información de distinto tipo. Pero C también tiene dos tipos especiales de "struct", de manejo más avanzado. Son las uniones y los campos de bits.

Una **unión** recuerda a un "struct" normal, con la diferencia de que sus "campos" comparten el mismo espacio de memoria:

```
union
{ char letra; /* 1 byte */
  int numero; /* 4 bytes */ } ejemplo;
```

En este caso, la variable "ejemplo" ocupa 4 bytes en memoria (suponiendo que estemos trabajando en un compilador de 32 bits, como lo son la mayoría de los de Windows y Linux) primer byte está compartido por "letra" y por "numero", y los tres últimos bytes sólo pertenecen a "numero".

Si hacemos

```
ejemplo.numero = 25;
ejemplo.letra = 50;
printf("%d", ejemplo.numero);
```

Veremos que "ejemplo.numero" ya no vale 25, puesto que al modificar "ejemplo.letra" estamos cambiando su primer byte. Ahora "ejemplo.numero" valdría 50 o un número mucho más grande, según si el ordenador que estamos utilizando almacena en primer lugar el byte más significativo o el menos significativo.

Campos de bits

Un **campo de bits** es un elemento de un registro (struct), que se define basándose en su tamaño en bits. Se define de forma muy parecida (pero no igual) a un "struct" normal, indicando el número de bits que se debe reservar a cada elemento:

```
struct campo_de_bits
{
  int bit_1             : 1;
  int bits_2_a_5        : 4;
  int bit_6             : 1;
  int bits_7_a_16       : 10; } variableDeBits;
```

Esta variable ocuparía 1+4+1+10 = 16 bits (2 bytes).

Los campos de bits pueden ser interesantes cuando queramos optimizar al máximo el espacio ocupado por nuestro datos.

```
#include <stdio.h>
#include <stdlib.h>
struct
{
  unsigned int motor1 : 1;
  unsigned int motor2 : 1;
  unsigned int luces1 : 1;
  unsigned int luces2 : 1;
  unsigned int luces3 : 1;
  unsigned int entradas : 3;
} status;

int main( )
{
   printf( "Espacio de memoria ocupado por status : %d\n", sizeof(status));
   status.entradas=1;
   if(status.entradas==0)
    status.motor1=1;
   else
    {
    status.luces1=0;
    status.luces2=1;
    }
    printf("status vale %x",status);
return 0;
}
```

Enumeraciones

Cuando tenemos varias constantes, cuyos valores son números enteros, y especialmente si son números enteros consecutivos, tenemos una forma abreviada de definirlos. Se trata de **enumerarlos**:

enum diasSemana **{** LUNES**,** MARTES**,** MIERCOLES**,** JUEVES**,** VIERNES**,** SABADO**,** DOMINGO **};**

La primera constante valdrá 0, y las demás irán aumentando de una en una, de modo que en nuestro caso valen: LUNES = 0, MARTES = 1, MIERCOLES = 2, JUEVES = 3, VIERNES = 4, SABADO = 5, DOMINGO = 6

Si queremos que los valores no sean exactamente estos, podemos dar valor a cualquiera de las contantes, y las siguientes irán aumentando de uno en uno.

Por ejemplo:

enum diasSemana **{** LUNES**=**1**,** MARTES**,** MIERCOLES**,** JUEVES**=**6**,** VIERNES**,** SABADO**=**10**,** DOMINGO **};**

Ahora sus valores son: LUNES = 1, MARTES = 2, MIERCOLES = 3, JUEVES = 6, VIERNES = 7, SABADO = 10, DOMINGO = 11

Definición de tipos

El tipo de una variable nos indica el rango de valores que puede tomar. Tenemos creados para nosotros los tipos básicos, pero puede que nos interese crear nuestros propios tipos de variables, para lo que usamos "**typedef**". El formato es:

typedef tipo nombre;

Lo que hacemos es darle un nuevo nombre a un tipo de datos. Puede ser cómodo para hacer mas legible nuestros programas.

Por ejemplo, alguien que conozca Pascal o Java, puede echar en falta un tipo de datos "boolean", que permita hacer comparaciones un poco más legibles, siguiendo esta estructura:

if (encontrado **==** FALSE**) ...**

o bien como

if (datosCorrectos **==** TRUE**) ...**

```
#define TRUE 1
#define FALSE 0
typedef int boolean;
typedef int integer;
integer clave;
boolean acertado;
```