Manejo de Bits en Lenguaje C

Licenciatura en Ciencias de la Computación-FCEIA-UNR
Arquitectura del Computador
Dr. Diego Feroldi

Agosto de 2019

1. Introducción

Como hemos visto en los apuntes anteriores, la computadora internamente trabaja en formato binario. Por lo tanto, la manipulación de bits es algo muy usual. En efecto, las operaciones de bits (u operaciones bit a bit) son muy comunes en Assembler dado que es un programa de bajo nivel. El lenguaje C no provee un tipo de datos específico para trabajar con bits. Sin embargo, podemos operar a nivel bits trabajando con datos de tipo entero considerando su representación interna.

Por ejemplo, si queremos expresar el valor $(01000001)_2$ podemos declarar una variable `A` como `char` y luego escribir en C:

- `A=65` (decimal)
- `A=0x41` (hexadecimal)
- `A=0101` (octal)
- `A=0b01000001` (binario)
- `A='A'` (carácter)

Estas expresiones son equivalentes dado que la representación interna es la misma, es decir es la misma secuencia de bits.

Como ya hemos visto, la forma más conveniente es utilizar hexadecimal dado que es una representación compacta y además la conversión binario-hexadecimal es fácil. Recordar que para realizar dicha conversión basta con tomar grupos de cuatro bits y realizar la conversión individual de cada uno de esos grupos.

2. Operadores de bits en C

Veamos primero en la siguiente tabla cuáles son los principales operadores de bits (*bitwise operators*) en lenguaje C:

Operador	**Acción**
&	Operación AND
|	Operación OR
$\wedge$	Operación XOR
$\sim$	Complemento a uno
$\gg$	Desplazamiento a la derecha
$\ll$	Desplazamiento a la izquierda

Importante: Los operadores de bits pueden ser aplicados a variables de tipo entero ya sea con signo o sin signo (`char`, `short`, `int`, `long`, etc.) y NO pueden ser aplicados a variables tipo flotante (`float`, `double`, etc.).

2.1. Operador AND (&)

La operación AND esta definida por la siguiente tabla de verdad:

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Supongamos que tenemos dos variables, `a=56` y `b=72`, entonces la operación AND entre estas dos variables resulta:

$$\begin{array}{cccccccccc} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \& & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Es decir, `a&b=8`.

Notar que el resultado hubiera sido diferente si se hubiera aplicado el operador lógico `&&`. En este caso habría resultado `a&&b=1`. El operador AND lógico (`&&`) devuelve el valor booleano *true* si ambos operandos son *true*. En caso contrario, devuelve *false*. Los operandos se convierten implícitamente al tipo *bool* antes de su evaluación y el resultado de la operación es de tipo *bool*.

Una aplicación útil para el operador `&` consiste en determinar si un determinado bit de cierto número es 1 o 0.

Ejemplo 1. Si se tiene el número `a=72` y se quiere averiguar si el cuarto bit de dicho número es 1 o 0, podemos aplicar el operador `&` realizando la operación `a&b`, donde `b` es un número cuya representación binaria es 00001000, es decir un número cuyo cuarto bit es 1 y todos los demás son ceros. A este último número se lo denomina "máscara"[1]. Entonces:

$$\begin{array}{cccccccccc} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \& & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Por lo tanto, al realizar `a&b` vemos que efectivamente el cuarto bit del número es 1. Ya veremos que este ejemplo se puede mejorar utilizando el operador desplazamiento.

2.2. Operador OR (|)

La operación OR esta definida por la siguiente tabla de verdad:

[1] Las "máscaras" son secuencias de bits que combinadas en una operación binaria con cualquier otra secuencia de bits permite modificar esta última u obtener alguna información sobre ella.

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

Supongamos que tenemos dos variables, `a=56` y `b=72`, entonces la operación OR entre estas dos variables resulta:

$$\begin{array}{cccccccc} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ | & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Es decir, `a|b=120`. Notar que el resultado es distinto al obtenido si se utiliza el operador OR lógico (||).

El operador | se puede utilizar para "encender" determinados bits de un número.

Ejemplo 2. Sea el número `a=28`, si queremos poner en uno el séptimo bit podemos realizar `a=a|b`, donde `b=01000000`:

$$\begin{array}{cccccccc} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ | & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{array}$$

Podemos observar que el séptimo bit de `a` ahora es uno.

2.3. Operador XOR ($\wedge$)

La operación XOR está definida por la siguiente tabla de verdad:

A	B	A $\wedge$ B
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos que tenemos dos variables, `a=56` y `b=72`, entonces la operación XOR entre estas dos variables resulta:

$$\begin{array}{cccccccc} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \wedge & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Es decir, `a`$\wedge$`b=112`.

El operador $\wedge$ se puede utilizar para invertir determinados bits de un número.

Ejemplo 3. Sea el número `a=28`, si queremos invertir el cuarto bit podemos realizar `a=a`$\wedge$`b`, donde `b=00001000`:

$$\begin{array}{cccccccc} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \wedge & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Vemos que el cuarto bit de `a` pasó de ser uno a ser cero.

2.3.1. Propiedades del operador XOR

A partir de la tabla de la verdad del operador XOR se pueden determinar las siguientes propiedades:

- Conmutativa: $A \wedge B = B \wedge A$
- Asociativa: $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
- $A \wedge A = 0$
- $A \wedge 0 = A$

A partir de las propiedades anteriores resulta:

- $(B \wedge A) \wedge A = B \wedge 0 = B$

Esto resulta útil para hacer cifrado de datos. Una cadena de texto puede ser cifrada aplicando el operador de bit XOR sobre cada uno de los caracteres utilizando una clave. Para descifrar la salida, solo hay que volver a aplicar el operador XOR con la misma clave.

Ejemplo 4. La cadena de caracteres "Hola" se puede representar utilizando código ASCII como 01001000 01101111 01101100 01100001. Esta cadena puede ser cifrada con la clave 11110011 de la siguiente manera:

$$\begin{array}{cl} & 01001000\ 01101111\ 01101100\ 01100001 \\ \wedge & 11110011\ 11110011\ 11110011\ 11110011 \\ \hline & 10111011\ 10011100\ 10011111\ 10010010 \end{array}$$

Si a este resultado se le vuelve a aplicar el operador XOR con la misma clave volvemos a tener la cadena original:

$$\begin{array}{cl} & 10111011\ 10011100\ 10011111\ 10010010 \\ \wedge & 11110011\ 11110011\ 11110011\ 11110011 \\ \hline & 01001000\ 01101111\ 01101100\ 01100001 \end{array}$$

2.4. Operador complemento a uno ($\sim$)

El operador binario $\sim$ se conoce como operador complemento a uno o también como operador NOT. Es un operador unario, es decir solo necesita un operando. La tabla de la verdad de este operador es la siguiente:

A	$\sim$A
0	1
1	0

Por lo tanto, si tenemos la variable `a=56`, resulta $\sim$`a=-57` dado que la secuencia de bits que representa a la variable `a` es 00111000 y entonces la secuencia que resulta de aplicarle el operador $\sim$ resulta 11000111.

Ejemplo 5. La expresión `x=x&~0xff` pone los últimos 8 bits de `x` en cero. Notar que esto es independiente de la longitud del tipo de dato a diferencia de la expresión `x=x&0xffffff00` que asume que `x` tiene 32 bits.

2.5. Operadores desplazamiento de bits ($\ll$, $\gg$)

Se pueden realizar corrimientos de N bits, con $N \in \mathbb{N}$, es decir $N = 1, 2, \ldots$. Los desplazamientos de bits pueden ser en dos direcciones: desplazamiento hacia la derecha o desplazamiento hacia la izquierda. Cada vez que se hace un desplazamiento se completa con ceros en el otro lado (no se trata de una rotación).

Supongamos que tenemos el número `a=56` ($56_{10} = 00111000_2$) y le realizamos un corrimiento de un bit hacia la derecha: `a`$\gg$`1`. El resultado es $00011100_2 = 28_{10}$. Es decir, se ha dividido al número `a` por dos. Por lo tanto, desplazando el número n bits hacia la derecha se realiza la división entera del número:

$$a \gg n = a/2^n$$

Análogamente, el desplazamiento a la izquierda es equivalente a multiplicar por potencias de dos:

$$a \ll n = a \times 2^n \tag{1}$$

La ventaja de los desplazamientos es que son menos costosos que hacer las operaciones de multiplicación y división.

> **Importante:** Hay que tener cuidado al realizar desplazamientos hacia la izquierda dado que se obtener resultados erróneos (con respecto a la ecuación 1) debido al *overflow*.
> También hay que tener precauciones la desplazar hacia la derecha:
>
> - En los desplazamientos hacia la derecha de valores sin signo siempre se completan los bits vacíos con ceros.
> - En los desplazamientos hacia la derecha de valores con signo se completa de acuerdo al bit de signo ("*arithmetic shift*") en algunas máquinas y con ceros ("*logical shift*") en otras.

Ejemplo 6. En el Ejemplo 1 vimos cómo determinar si un bit determinado de un número es 0 o 1. Dicho ejemplo se puede mejorar realizando un desplazamiento hacia la derecha luego de aplicar el operador &:

```
(a&b)>>3
```

El resultado es 1, indicando que el cuarto bit es efectivamente 1.

3. Operador ternario (? :)

El operador ternario[2] se puede interpretar desde el punto de vista de un `if/else`. El operador ternario funciona con tres expresiones: $E1$, $E2$ y $E3$. Por lo tanto, el código `E1?E2:E3` resulta equivalente al siguiente código:

```
if (E1)
   E2
else
   E3
```

[2]Si bien el operador ternario no es un operador bit a bit, lo introducimos en este apunte dado que es útil al trabajar con operadores de bits como se puede ver en el Ejemplo 1.

Ejemplo 1. Determinar si un número es par o impar.

```
#include <stdio.h>

int main()
{
   char a;
   printf ("Ingrese un número \n" );
   scanf ("%d",&a);
   (a&1)?printf("El número es impar \n" ):printf("El número es par \n");
   return 0;
}
```

4. Campos de bits

Cuando el espacio de almacenamiento es escaso, puede ser necesario empaquetar varios objetos en una sola palabra de máquina. El método que se utiliza en C para operar con campos de bits, está basado en las estructuras (*structs*). La forma general de definición de un campo de bits es la siguiente:

```
typedef struct bits
{
   <tipo de dato> nombre1 : <cantidad de bits>;
   <tipo de dato> nombre2 : <cantidad de bits>;
   . . .
   <tipo de dato> nombre3 : <cantidad de bits >;
}campo_bits ;
```

Cada campo de bits puede declararse como `char`, `unsigned char`, `int`, `unsigned int`, etc. y su longitud puede variar entre 1 y el máximo número de bits disponible de acuerdo a la arquitectura del microprocesador en que se esté trabajando.

Ejemplo 1. Uso de campos de bits.

```
#include <stdio.h>

typedef struct s_bits
{
   unsigned int mostrar: 1;
   unsigned int rojo: 8;
   unsigned int azul: 8;
   unsigned int verde: 8;
   unsigned int transparencia: 1;
}campo_bits;

int main()
{
   campo_bits unColor;
   /* Se crea un color verde */
   unColor.rojo = 51;
```

```
   unColor.azul = 55;
   unColor.verde = 255;
   unColor.transparencia = 1;

   /* Se verifica si el color es transparente */
   if (unColor.transparencia == 1)
   {
      printf("El color es transparente\n");
   }
   return 0;
}
```

Referencias

[KR17] Brian Kernighan and Dennis M Ritchie. *The C programming language*. Prentice Hall, 2017.

[War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.