Operadores de bits

Sebastián Santisi

2020-05-31

Independientemente de la aritmética que utilicemos a alto nivel, los procesadores internamente operan y almacenan resultados en la memoria según patrones de bits y por lo tanto en formato binario. El poder acceder a los patrones de bits de una variable puede ser una operación útil para múltiples aplicaciones.

Sistemas de numeración

Antes de entrar en las operatorias de bits repasemos algunos conceptos de sistemas de numeración.

Habitualmente utilizamos sistemas de numeración *pesados*, es decir, cada uno de los dígitos de un número tiene un *peso* según su posición. Ese peso está dado por la base que utilicemos para representar.

Por ejemplo el número 9511 en sistema decimal (base 10) se puede descomponer como $1\,10^0 + 1\,10^1 + 5\,10^2 + 9\,10^3 + 0\,10^4 + 0\,10^5 + \ldots$. De derecha a izquierda cada dígito tiene un peso relativo, dado por potencias de la base, y podemos ver que los ceros a izquierda no importan, es decir el número 9511 es el mismo que el 09511 o el 00009511.

Para cualquier base b dada representamos un número N según sus dígitos como:

$$N = \sum_{i=0}^{\infty} d_i b^i,$$

y esta representación será única si exigimos que $d_i < b, \forall i$.

Por ejemplo, retomando el ejemplo anterior los dígitos del número 9511 se representan como $d = \{1, 1, 5, 9, 0, \ldots\}$ en base 10. Ahora bien, el mismo número se puede representar como $d = \{7, 4, 4, 2, 2, 0, \ldots\}$ en sistema octal (base 8) dado que $7\,8^0 + 4\,8^1 + 4\,8^2 + 2\,8^3 + 2\,8^4 = 9511$.

Pero... ¿en qué base está representado N para que la ecuación anterior tenga sentido? Bueno, esto es lo curioso, N es siempre N sin importar en qué base lo representemos y sin importar en qué base hagamos las cuentas. Cuando no aclaremos nada vamos a asumir (como lo vinimos haciendo en los 20 años previos de formación matemática) que los números son en base decimal, en cambio cuando los expresemos en otro sistema lo vamos a aclarar con un subíndice. Entonces, lo que estamos diciendo es que $9511 = 9511_{10} = 22447_8$. El N de nuestra cuenta estará expresado en la base en la que hayamos hecho la cuenta y operar con números tiene que dar el mismo resultado sin importar en qué base hagamos las cuentas.

Entonces, cuando operamos $5 + 6$ en una computadora sabemos que la computadora opera en binario, por lo que la cuenta que se realiza es $101_2 + 110_2$ y eso va a ser igual a 1011_2 que no es otra cosa que 11.

¿Cómo hacemos operaciones en otras bases?, con el mismo algoritmo que en base 10, pero no es de importancia para este curso.

¿Se pueden hacer conversiones entre bases?, sí, claro, pero no es de importancia para este curso con excepción de la conversión entre 3 bases específicas.

Conversión entre base 2, 8 y 16

De entre todas las bases posibles nos interesan 3 de ellas y son la base binaria (2), octal (8) y hexadecimal (16).

La binaria es una base de interés porque ya mencionamos que las computadoras operan en binario... ¿Y las otras dos?

Las bases octal y hexadecimal nos interesan porque son cómodas para representar números binarios utilizando muchos menos dígitos por el hecho de que $2^3 = 8$ y que $2^4 = 16$. Entonces un número binario puede escribirse en octal agrupando sus dígitos de 3 en 3 y convirtiendo estos tripletes individualmente y análogamente para los números hexadecimales pero de a 4 dígitos.

Lamentablemente no existen n y m tales que $2^n = 10^m$ por lo que no existe una conversión fácil entre binario y decimal. Y esto es un *problema* conocido que hace que las computadoras consuman mucho tiempo en conversiones (y que a veces se implementen computadoras que funcionan en BCD que es un estándar para la representación de dígitos decimales).

> Para evitar perder tiempo en conversiones los números que pensemos en decimal los expresaremos en decimal y nunca nos preocuparemos por cómo se escriben en binario, mientras que los números que pensemos en binario los escribiremos en binario, octal o hexadecimal.

La siguiente tabla muestra los primeros 17 dígitos en distintas bases:

Decimal	Binario	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10

Si por ejemplo quisiéramos convertir el número 10010100100111_2 a octal lo primero que haríamos sería agrupar de a 3 dígitos desde el dígito menos pesado: $10\ 010\ 100\ 100\ 111_2$. Luego convertimos cada una de estas tríadas de dígitos binarios en su equivalente octal, completando con ceros a izquierda de ser necesario: 22447_8. En cambio si quisiéramos convertirlo a hexadecimal agruparíamos de a 4 dígitos: $10\ 0101\ 0010\ 0111_2$ por lo que el número sería 2527_{16}.

Si quisiéramos convertir el número $1D4E_{16}$ en binario reemplazaríamos cada uno de sus dígitos por los 4 bits que representan: $0001\ 1101\ 0100\ 1110_2$ por lo que el número es 1110101001110_2.

Dado que $16^2 = 2^8$, un byte se representa con 2 dígitos hexadecimales y por lo tanto el hexadecimal será el formato preferido para representar memoria.

Sistemas de numeración y C

El lenguaje de programación C permite representar literales en base decimal, octal y hexadecimal. No hay forma de escribir literales binarios en el lenguaje estándar.

Los literales decimales sencillamente se escriben según sus dígitos, como por ejemplo `9511`. Los literales octales tienen un número impar de ceros adelante, como por ejemplo `022447`. Los literales hexadecimales llevan un cero seguido de una equis adelante, como por ejemplo `0x2527`. En el caso de los tres ejemplos internamente se almacenará el valor 10010100100111_2 con tantos ceros a izquierda como haga falta para completar los bytes de un `int`.

Para imprimir números utilizando `printf()` puede utilizarse el formato `%i` o `%d` para decimales, `%o` para octales y `%x` y `%X` para hexadecimales (según la x sea mayúscula lo serán los dígitos alfabéticos).

Para convertir cadenas de caracteres numéricas en distintas bases se puede utilizar la función `strtol()` que permite ingresar cualquier base entre 2 y 36.

Operaciones de bits

El lenguaje C tiene operadores aritméticos, lógicos y de bits. Estos últimos operan sobre la memoria bit a bit.

Estos operadores son `|` el OR de bit, `&` el AND de bit, `~` el NOT de bit (también llamado complemento a uno, o complemento a la base menos uno) y `^` el XOR de bit. Además están los operadores `<<` y `>>` que son el desplazamiento a la *izquierda* y a la *derecha* respectivamente.

Para un solo bit, repasemos las tablas de verdad de estos operadores:

`A`	`B`	`A | B`	`A & B`	`~A`	`A ^ B`
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	0	1
1	1	1	1	0	0

El resultado de una operación de bits en C es el resultado de operar estos operadores bit a bit.

Ejemplos:

$$1100\ 1010_2 \ \&\ 1010\ 0101_2 = 1000\ 0000_2,$$
$$1100\ 1010_2 \mid 1010\ 0101_2 = 1110\ 1111_2,$$
$$1100\ 1010_2 \hat{}\ 1010\ 0101_2 = 0110\ 1111_2,$$
$$\sim 1100\ 1010_2 = 0011\ 0101_2.$$

Los corrimientos desplazan todos los bits del número una cantidad dada de posiciones.

El corrimiento a *izquierda*, `<< n` desplaza los bits `n` posiciones en el sentido del bit más pesado (o most significant bit, o MSB). Al realizar esta operación los valores que se *caigan* en el sentido del MSB se pierden y siempre entran ceros en el sentido del bit menos pesado (o bit 0, o least significant bit, o LSB).

El corrimiento a *derecha*, `>> n` desplaza los bits en el sentido del LSB. Al realizar esta operación los valores que se *caigan* por el LSB del número se pierden, y los valores que entran por el MSB serán ceros sólo si el número está declarado como `unsigned`. En otro caso el resultado será indefinido.

Por ejemplo, asumiendo que los números son `uint8_t`:

$$1100\ 1010_2 \ \ll\ 1 = 1001\ 0100_2,$$

$$1100\ 1010_2 \gg 2 = 0011\ 0010_2.$$

Cabe destacar que desplazar la representación binaria de un número n veces a la izquierda es equivalente a multiplicar por 2^n mientras que hacerlo a la derecha es equivalente a dividirlo por 2^n. (Este es un resultado que ya tenemos bien sabido para la representación decimal y las potencias de 10.)

> Las operaciones de bits tienen sentido en valores que pensamos en función de sus bits. Por lo general no utilizamos operadores de bits para números que pensamos en su sentido numérico.

Aplicaciones

Las operaciones de bits son últiles en muchas aplicaciones de bajo nivel dado que suele ser usual empaquetar diversas variables adentro de pocos bytes, dándole a cada uno de los bits de dicha variable un sentido determinado.

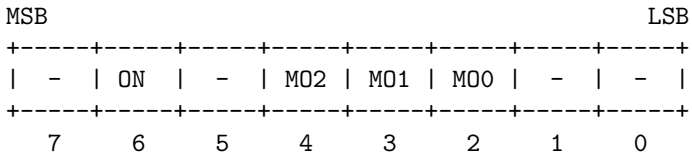
Nos va a interesar, entonces, poder obtener o almacenar valores en determinados bits de una variable más grande sin interferir con los demás bits.

Las operaciones de interés se pueden separar en dos: adquisición (get) o almacenamiento (set) y a su vez ambas pueden ser o para un solo bit por vez o para un conjunto de bits por vez.

Máscaras

Para cualquier aplicación determinada donde nos interese acceder a bits particulares estos bits tendrán una función específica dada por el problema.

Por ejemplo, supongamos que una aplicación codifica si está activa o no y el modo de operación en un registro de control dentro de un byte determinado según el siguiente esquema:



donde si el bit 6 está en 1 eso significa que la aplicación está activa mientras que cero implica que está inactiva y donde los bits 4, 3 y 2 tienen que interpretarse como 3 dígitos binarios de un modo que puede estar entre 000_2 y 111_2.

Los bits restantes del registro no conocemos qué función tienen, pero puede ser que estén codificando otra cosa, por lo que no podemos ni modificarlos ni asumir que están en algún valor determinado.

Dado que estos bits tienen una función específica el primer paso será crear *máscaras* para esos dígitos. Una máscara no es más que un número que tiene unos en las posiciones que *enmascara* y ceros en las que no son de interés.

Para el ejemplo:

```
#define MASK_ON 0x40    // 0100 0000
#define MASK_MO 0x1C    // 0001 1100
```

Se incluyen los valores binarios como un comentario para hacer más claro el ejemplo. En una aplicación real se utiliza exclusivamente hexadecimal.

Para los valores de más de un bit a veces nos resulta de interés saber a cuántos bits están estos bits del LSB y definimos estos valores como desplazamientos:

```
#define SHIFT_MO 2
```

Notar que esta cantidad es de tipo decimal, se trata de una cantidad de valores a desplazar. Como ya sabemos: Los valores numéricos los expresamos en decimal.

Obtener el valor de un bit

Cuando queremos obtener el valor de un único bit el mismo puede tener uno de dos valores: o 1 o 0, es decir: es un resultado booleano.

Para obtener un bit aprovecharemos que tenemos una máscara que lo define y que sabemos que la operación de AND elimina todos los valores donde se opere contra algo igual a cero.

Por ejemplo si tenemos un registro y una máscara que enmascara el bit 6:

$$1\underline{1}00\ 1010_2\ \&\ 0\underline{1}00\ 0000_2 = 0\underline{1}00\ 0000_2,$$
$$1\underline{0}01\ 0101_2\ \&\ 0\underline{1}00\ 0000_2 = 0\underline{0}00\ 0000_2.$$

Como la máscara tiene un sólo bit en alto y se realiza una operación AND hay dos resultados posibles: O el resultado vale cero o el resultado vale la misma máscara. Si el resultado es cero es porque el bit enmascarado estaba en cero, si vale la máscara es porque estaba en uno.

Entonces, si tuviéramos que programar la operación:

```
bool get_on(uint8_t reg) {
    return reg & MASK_ON;
}
```

Definir el valor de un bit

La operación de definir el valor de un bit tiene dos enfoques diferentes dependiendo de si se quiera poner en 1 (set) o en 0 (clear) el bit en cuestión.

Para poner en 1 el bit puede aprovecharse que una operación de OR con una máscara independientemente del valor del registro forzará un uno:

$$1\underline{1}00\ 1010_2\ |\ 0\underline{1}00\ 0000_2 = 1\underline{1}00\ 1010_2,$$
$$1\underline{0}01\ 0101_2\ |\ 0\underline{1}00\ 0000_2 = 1\underline{1}01\ 0101_2.$$

En código esta operación sería:

```
uint8_t set_on(uint8_t reg) {
    return reg | MASK_ON;
}
```

En el caso de que se quisiera poner a cero eso se logra haciendo una operación AND donde todos los valores son 1 con excepción del bit de interés.

Para construir esa máscara **debe** utilizarse la máscara **ya** definida, esto puede hacerse realizando una operación de NOT sobre la misma. Cabe destacar que **bajo ningún punto de vista** definimos una máscara nueva para esta operación. Esto es porque definir una máscara *invertida*, como por ejemplo `0xBF` no es equivalente a `~0x40` dado que la primera tiene infinitos ceros a izquierda mientras que la segunda tiene infinitos unos a izquierda (por haber negado un número con infinitos ceros a izquierda).

Entonces:

$$1\underline{1}00\ 1010_2\ \&\ \sim 0\underline{1}00\ 0000_2 = 1\underline{1}00\ 1010_2\ \&\ 1\underline{0}11\ 1111_2 = 1\underline{0}00\ 1010_2,$$
$$1\underline{0}01\ 0101_2\ \&\ \sim 0\underline{1}00\ 0000_2 = 1\underline{0}01\ 0101_2\ \&\ 1\underline{0}11\ 1111_2 = 1\underline{0}01\ 0101_2.$$

En código:

```
uint8_t clear_on(uint8_t reg) {
    return reg & ~MASK_ON;
}
```

Estas dos operaciones se pueden juntar en una única función:

```
uint8_t set_on(uint8_t reg, bool value) {
    if(value)
        return reg | MASK_ON;
    return reg & ~MASK_ON;
}
```

Por otro lado cabe destacar que generalmente se suele trabajar actualizando el mismo registro, es decir modificando el registro recibido por la interfaz:

```
void set_on(uint8_t *reg, bool value) {
    if(value)
        *reg |= MASK_ON;
    else
        *reg &= ~MASK_ON;
}
```

Obtener el valor de un conjunto de bits

Las operaciones para manipular un conjunto de bits son análogas, lo que va a cambiar es el resultado de dichas operaciones.

Mientras que en las operaciones sobre un solo bit el resultado era booleano, en las operaciones con n bits se obtendrán 2^n valores posibles, y para manipular esos valores de forma numérica se busca que el mismo esté en el intervalo $[0, 2^n)$. Para obtener esto, luego de enmascarar el valor el mismo debe ser desplazado hacia el LSB.

Entonces:

```
int get_mo(uint8_t *reg) {
    return (*reg & MASK_MO) >> SHIFT_MO;
}
```

y se devolverá un número entre 0 y 7 (111_2).

(Se deja expandir las operaciones al alumno.)

> En el caso de operaciones de más de un bit, bajo ningún caso se trabaja con bits individuales. La asignación y obtención de los valores se hace siempre en bloque.

Definir el valor de un conjunto de bits

A diferencia del caso de un solo bit donde tiene sentido plantear una estrategia para la limpieza y otra para asignar en el caso de conjuntos de bits limpiamos primero para luego asignar.

El problema es que como no sabemos si los bits de interés están en 0 o en 1 antes de la operación en principio no sabemos cómo van a reaccionar OR o AND porque ambos dependen del valor previo. Entonces lo que se hace es asegurar primero que los bits estén en 0, para luego poder asignar el valor de interés utilizando OR.

Entonces:

```
// Precondición: 0 <= mo < 8
void set_mo(uint8_t *reg, int mo) {
    *reg &= ~MASK_MO;
    *reg |= mo << SHIFT_MO;
}
```