Operadores a nivel de bit

Informática II - R2004

Recordando de info 1

Los datos dentro de una variable se encuentran almacenados en “lenguaje máquina” solo con 1 y 0. Para ello existen distintos tipos de Sistemas de Numeración, específicamente en nuestro microprocesador tendremos variables enteras signadas y no signadas (más adelante veremos que pasa con las variables de tipo punto flotante) codificadas en Complemento a 2.

Complemento a 2: Los números positivos se representan con su binario y los negativos con el Ca2 de los mismos.

Recordando Info 1

Un número puede codificarse en distintas bases, las más utilizadas para la programación de microcontroladores son:

- Decimal (en base 10, la que más conocemos)
- Binario (en base 2, sólo 1 y 0)
- Hexadecimal (Se transforma rápidamente a binario y en general es soportada por los compiladores a diferencia de la binaria que no todos los compiladores la soportan)

Por ejemplo:

$52_{(10)} = 0x34 = 00110100_{(2)}$

El 0x le indica al compilador que el número se encuentra en hexadecimal si no pongo nada lo interpreta en decimal.

$-52_{(10)} = -0x34 = 11001100_{(2)}$

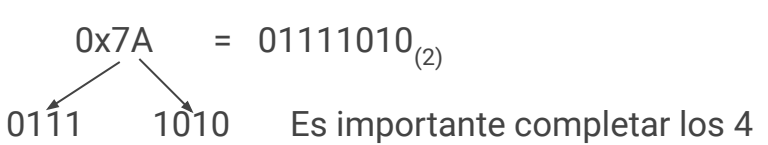
Este número es el Ca2 del 00110100 para que el procesador pueda almacenarlo.

Recordando Info 1

La conversión de bases se realiza con diferentes procedimientos, las que más vamos a utilizar son:

decimal a hexadecimal: Recordamos que el hexadecimal se encuentra compuesto por los números del 0 al 9 más las letras ABCDEF. Para la conversión tengo que dividir el número decimal por 16 sucesivamente y quedarme con los restos. Sin embargo para esta etapa ya pueden hacerlo con la calculadora.

hexadecimal a binario: Esta conversión es muy sencilla y es por eso que se utiliza tanto.



Es importante completar los 4

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

binario o hexa a decimal: Tengo que multiplicar cada número por el peso de su ubicación en la base correspondiente.

$01001110 = 0*2^7+1*2^6+0*2^5+0*2^4+1*2^3+1*2^2+1*2^1+0*2^0 = 78$

$0x7A4 = 7*16^2+10*16^1+4*16^0 = 1956$

EN INFO 2 TRABAJAREMOS CON VARIABLES (Y REGISTROS) DE 32 bits Y SIN SIGNAR

Operadores en C

Podemos hacer operaciones varias utilizando los operadores de C, como +, -, *, /, ==, <, >, etc. En particular, nos focalizaremos en un conjunto de operadores que modifican el contenido de sus operandos A NIVEL DE BIT (esto es, haciendo una operación con cada uno de los bits de la variable, y no con su valor entero, interpretado como un valor decimal). Por ejemplo:

	0	0	0	0	1	0	1	1
					+			
	0	0	0	0	0	0	0	1
					=			
	0	0	0	0	1	1	0	0

11

1

12

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

0x2D

0x70

0x7D

No tiene sentido entender el contenido de ciertas variables por su valor decimal, cuando lo que nos importa es el contenido de cada uno de los bits por separado. Estas variables las expresamos en hexadecimal y utilizamos operadores a nivel de bit para modificar su contenido.

Ejemplo de operadores a nivel de bit: OR (|)

El operador OR compara dos bits, y si ALGUNO DE ELLOS es 1, el resultado de la operación es 1. Entonces:

TABLA DE VERDAD
Operacion OR

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

Si reinterpretamos el resultado de la operación, el uso más común que haremos de este operador será el de **FIJAR UNOS en el resultado**.

Pensémoslo asi: Independientemente del valor del bit A, si el bit B vale 1, el resultado será 1. En cambio, si el bit B vale 0, el resultado será el valor que tenía el bit A previamente. Por lo tanto, siempre que pongo un 1 en el operador B tendré un 1 en el resultado, y siempre que pongo un 0, dejo el valor tal cual está.

Si yo quisiera, en una variable de 8 bits, poner en 1 el bit 5, haría:

x	x	x	x	x	x	x	x	**variable**
0	0	1	0	0	0	0	0	**0x20**
x	x	1	x	x	x	x	x	**variable |= 0x20**

Ejemplo de operadores a nivel de bit: AND (**&**)

El operador AND compara dos bits, y si ALGUNO DE ELLOS es 0, el resultado de la operación es 0. Entonces:

TABLA DE VERDAD
Operacion AND

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

Si reinterpretamos el resultado de la operación, el uso más común que haremos de este operador será el de **FIJAR CEROS en el resultado**.

Pensémoslo asi: Independientemente del valor del bit A, si el bit B vale 0, el resultado será 0. En cambio, si el bit B vale 1, el resultado será el valor que tenía el bit A previamente. Por lo tanto, siempre que pongo un 0 en el operador B tendré un 0 en el resultado, y siempre que pongo un 1, dejo el valor tal cual está.

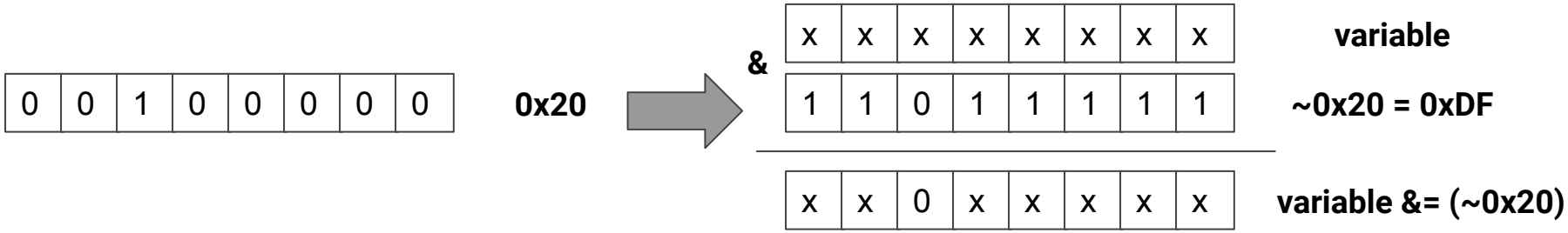
Si yo quisiera, en una variable de 8 bits, poner en 0 el bit 5, haría:

	x	x	x	x	x	x	x	x	**variable**
&	1	1	0	1	1	1	1	1	**0xDF**
	x	x	0	x	x	x	x	x	**variable &= 0xDF**

Ejemplo de operadores a nivel de bit: NOT (~)

El operador NOT es un operador UNARIO (tiene un solo operando), y su resultado es cambiar el valor del bit (de 0 a 1, o de 1 a 0)

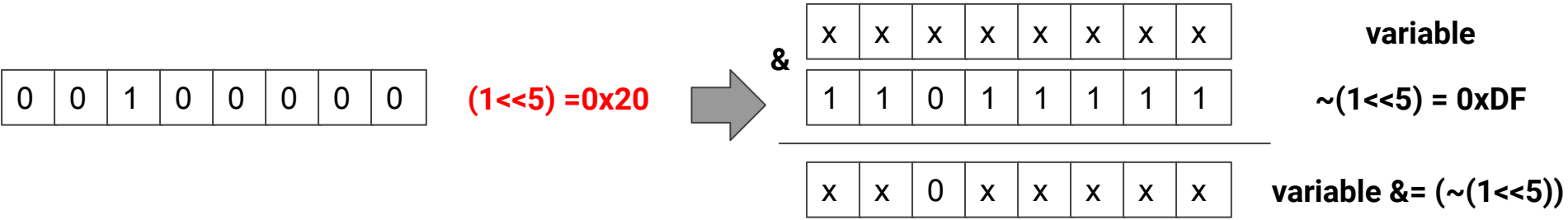
Este operador es útil, por ejemplo, para utilizar el operador &, pero poniendo un 1 en lugar de un 0 en la variable (porque es más sencillo calcular su valor hexadecimal). En el ejemplo anterior:



Ejemplo de operadores a nivel de bit: SHIFT (<< y >>)

Los operadores Shift right y shift left DESPLAZAN los bits de una variable a la izquierda o a la derecha, tantas veces como lo diga el segundo operador de la operación. El bit “nuevo” que aparece como resultado del desplazamiento se rellena con 0

Este operador es útil, para “poner un uno” en una posicion determinada, sin tener que hacer el cálculo del número en hexadecimal. En el ejemplo anterior:



Es interesante notar que un desplazamiento de todos los bits a la izquierda implica multiplicar la variable x 2. Un desplazamiento a la derecha es igual a dividir la variable por 2

Ejemplo de operadores a nivel de bit: XOR (^)

El operador XOR compara dos bits, y si SOLO UNO DE ELLOS es 1, su resultado es 1. Entonces:

TABLA DE VERDAD
Operacion XOR

Bit A	Bit B	Bit A^B
0	0	0
0	1	1
1	0	1
1	1	0

Si reinterpretamos el resultado de la operación, el uso más común que haremos de este operador será el de **CAMBIAR EL VALOR DE UN BIT en el resultado**.

Pensémoslo asi: Independientemente del valor del bit A, si el bit B vale 0, el resultado será el mismo valor que tenía el bit A. En cambio, si el bit B vale 1, el resultado será el valor opuesto al que tenía el bit A previamente. Por lo tanto, siempre que pongo un 1 en el operador B tendré modificaré el valor del bit A, y si pongo un 0 lo dejo como está:

Si yo quisiera, en una variable de 8 bits, cambiar el valor del bit 6 haría::

	x	x	x	x	x	x	x	x	**variable**
^	0	1	0	0	0	0	0	0	**(1<<6)**
	x	~x	x	x	x	x	x	x	**variable ^= 1<<6**

Ejemplo:

Realizar las funciones:

SetearBit `( uint32_t * var , uint8_t nbit );` //Pone un 1 en el bit ***nbit*** de la variable ***var***

BajarBit `( uint32_t * var , uint8_t nbit );` //Pone un 0 en el bit ***nbit*** de la variable ***var***

ToggleBit `( uint32_t * var , uint8_t nbit );` //Cambia de estado el bit ***nbit*** de la variable ***var***

SetBit `( uint32_t * var , uint8_t nbit , uint8_t state);` //Pone el bit ***nbit*** de la variable ***var*** en el estado ***state***

```
SetearBit ( uint32_t * var , uint8_t nbit )
{
     (*var) |= ( 1 << nbit );
}


BajarBit ( uint32_t * var , uint8_t nbit )
{
     (*var) &= ~( 1 << nbit );
}

ToggleBit ( uint32_t * var , uint8_t nbit )
{
     (*var) ^= ( 1 << nbit );
}

SetBit ( uint32_t * var , uint8_t nbit , uint8_t state)
{
     if(state == 0)
          BajarBit ( var , nbit );
     else
          SetearBit ( var , nbit );
}
```