2.- SISTEMAS NUMÉRICOS Y DE REPRESENTACIÓN DE LA INFORMACIÓN

OBJETIVOS DEL TEMA:

- Revisión del sistema de numeración decimal
- Definición de sistema de numeración posicional. Ejemplos
- Estructura de pesos del sistema de numeración decimal
- Contar en el sistema de numeración binario
- Contar en el sistema de numeración hexadecimal
- Estructura de pesos de los sistemas de numeración binario y hexadecimal
- Conversiones entre sistemas de numeración binario, decimal y hexadecimal
- Aplicar las operaciones aritméticas de suma, resta y multiplicación a los números binarios
- Determinar el Complemento a 1 y el Complemento a 2 de un número binario
- Expresar números con signo en los formatos binarios: Signo/Magnitud, C-1, C-2 y coma flotante
- Realizar operaciones aritméticas con números binarios con signo
- Expresar los números decimales en el código BCD natural
- Sumar números en BCD
- Diferencias entre las representaciones en decimal desempaquetado y decimal empaquetado
- Códigos de paridad a partir de otros códigos
- Códigos de Hamming

2.1.- SISTEMAS DE NUMERACIÓN

A lo largo del tiempo, ha ido evolucionando el tipo de información almacenada y procesada en un ordenador. Inicialmente, sólo eran capaces de procesar información numérica. De hecho los primeros ordenadores se construyeron para facilitar complejos cálculos numéricos de tipo técnico.

Poco después se vio su utilidad para realizar cálculos sencillos pero de gran volumen, los típicos de los procesos administrativos. Entonces, se hizo preciso procesar información de tipo alfabético para poder emitir información impresa, como facturas o recibos de nómina.

Posteriormente se ha ido dotando a los ordenadores de la capacidad de procesar otros tipos de información más compleja, desde gráficos sencillos hasta los potentes equipos multimedia actuales.

Comenzaremos pues viendo cómo se representa en un ordenador la información numérica y para ello repasaremos algunos conceptos sencillos sobre Sistemas de Numeración.

En primer lugar debemos incluir algunas definiciones.

- **Sistema de Numeración**:

  "Conjunto de símbolos y reglas empleados para la representación de magnitudes numéricas"

- **Base de un Sistema de Numeración**

  "Número de símbolos numéricos que se emplean en ese Sistema de Numeración"

¿Cuáles son los Sistemas de Numeración más frecuentes? En nuestra vida habitual, no hay duda de que el Sistema de Numeración más comúnmente utilizado es el sistema decimal, aunque perdure de forma residual algún otro como pueda ser el de base 12, cuando se habla de 'docenas'. Sin embargo, en Sistemas Digitales y en Informática, los sistemas de numeración más empleados son el Sistema Binario y el Sistema Hexadecimal cuyas bases son 2 y 16, respectivamente.

2.1.1.- NÚMEROS DECIMALES

Todos estamos familiarizados con el sistema de numeración decimal porque usamos los números decimales cada día. Aunque los números decimales son triviales, a menudo su estructura de pesos no se comprende. En esta sección revisaremos la estructura de los números decimales para así entender más fácilmente la estructura del sistema de numeración binario, que es tan importante en las computadoras y la electrónica digital.

Habitualmente empleamos el Sistema Decimal. El valor numérico de una magnitud en este sistema viene dado por el conjunto de unidades, decenas, centenas, etc. Por ejemplo quince personas, se representan numéricamente, en el sistema decimal por el número 15 y éste representa cinco unidades y una decena, o bien quince unidades. *Quince* es el cardinal de este conjunto y es independiente del sistema de numeración empleado, su representación numérica en decimal es 15, mientras que su representación numérica en binario es 1111.

Cuando analizamos un número expresado en el sistema decimal, vemos (Figura 1) que un mismo símbolo (p.e. el 7) puede representar una magnitud diferente. En este caso, el primer 7 (por la derecha) representa 7 unidades, mientras que el segundo representa 7 centenas, o sea, 700 unidades. Es decir, el mismo símbolo en distinta posición representa un valor diferente.

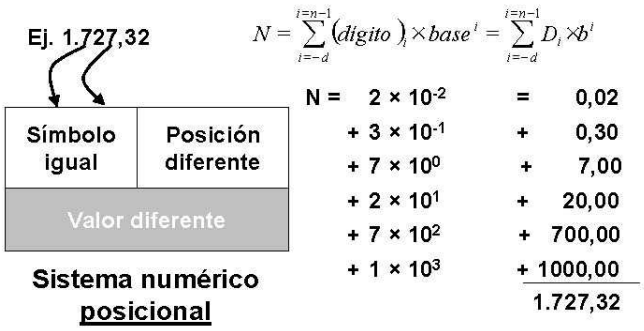

Figura 1

Cualquier sistema de numeración en el cual el valor de un símbolo depende de su posición se conoce como s*istema numérico posicional*. Prácticamente cualquier sistema numérico actualmente empleado es de este tipo.

Volviendo a la Figura 1, podemos obtener el valor de la cantidad representada en el sistema decimal por 1.727,32 con la siguiente suma:

a) dos centésimas, *más*
b) tres décimas, *más*
c) siete unidades, *más*
d) dos decenas, *más*
e) siete centenas, *más*
f) un millar

Un ejemplo de *sistema numérico no posiciona*, empleado hoy día, aunque sin utilidad ninguna en cálculos matemáticos, es el formado por los números romanos. En él la letra V tiene el valor cinco, independientemente de su posición, y la letra M el valor mil, también con independencia de su posición. Obsérvese que en este sistema de numeración no existe una representación para el valor cero.

La posición de cada dígito en un número decimal indica la magnitud de la cantidad presentada, y se le puede asignar un peso. Los pesos para los números enteros son potencias positivas de 10, que aumentan de derecha a izquierda, comenzando por $10^0=1$.

$$\ldots 10^5\ 10^4\ 10^3\ 10^2\ 10^1\ 10^0$$

Para números fraccionarios, los pesos son potencias negativas de diez que aumentan de izquierda a derecha, comenzando por 10^{-1}.

$$10^2\ 10^1\ 10^0, 10^{-1}\ 10^{-2}\ 10^{-3}\ \ldots$$

El valor de un número decimal es la suma de los dígitos después de haber multiplicado cada dígito por su peso.

Resumiendo:

- El sistema decimal con sus 10 dígitos es un sistema en base 10.
- Los 10 dígitos decimales son los comprendidos entre el 0 y el 9.
- La posición de un dígito en un número decimal indica su peso, o valor, dentro del número. Los pesos de un número decimal están basados en las potencias de 10.

Ejemplo 1: Expresar el número decimal 204 como suma de los valores de cada dígito.

Pesos	10^2	10^1	10^0
Dígitos	2	0	4

Como indican sus respectivas posiciones, el dígito 4 tiene un peso de 10^0 (1) y el dígito 2 tiene un peso de 10^2 (100). Por tanto,

$204 = 2 * 10^2 + 4 * 10^0 = 200 + 4$

Ejemplo 2: Expresar el valor de cada dígito en el número 523,51.

Pesos	10^2	10^1	10^0	,	10^{-1}	10^{-2}
Dígitos	5	2	3	,	5	1

Análogamente a como hicimos en el Ejemplo 1:

$523,51 = 5 * 10^2 + 2 * 10^1 + 3 * 10^0 + 5 * 10^{-1} + 1 * 10^{-2} = 500 + 20 + 3 + 0,5 + 0,01$

2.1.2.- NÚMEROS BINARIOS

El sistema de numeración binario es simplemente otra forma de representar magnitudes. El sistema binario es menos complicado que el sistema decimal ya que sólo tiene dos dígitos aunque al principio pueda parecer más complicado por no ser familiar.

- El sistema binario con sus dos dígitos es un sistema en base 2.
- Los 2 dígitos binarios (bits) son 1 y 0.
- La posición de un 1 o un 0 en un número binario indica su peso o valor dentro del número, así como la posición de un dígito decimal determina el valor de ese dígito. Los pesos de un número binario están basados en las potencias de 2.

Antes de conocer la estructura de pesos del sistema de numeración binario aprendamos a contar.

2.1.2.1.- Contar en binario

Para aprender a contar en el sistema binario, en primer lugar, hay que observar cómo contamos en el sistema decimal.

- Empezamos en cero y contamos hasta 9 quedándonos sin dígitos.
- Comenzamos con otra posición de dígito (a la izquierda), y contamos de 10 hasta 99. En este punto se terminan todas las combinaciones con 2 dígitos.
- Es necesaria una tercera posición de dígito para contar desde 100 hasta 999, etc.

Cuando contamos en binario se produce una situación similar, excepto que ahora sólo tenemos 2 dígitos.

- Empezamos en 0 y contamos hasta 1, quedándonos ya sin dígitos.
- Incluimos otra posición de dígito y contamos: 10 y 11. Ahora hemos agotado todas las combinaciones de 2 dígitos.
- Se requiere una tercera posición. Con 3 dígitos podemos continuar contando 100,101,110,111.
- Para continuar necesitamos una cuarta posición de dígito, y así sucesivamente.

En la Tabla 1 se muestra cómo se cuenta de cero hasta quince. Observe la alternancia de 0’s y 1’s en cada columna.

Decimal	Binario			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Tabla 1

Como puede verse en la Tabla 1, se requieren 4 bits para contar desde 0 hasta 15. En general, con n bits, se puede contar hasta un número igual a 2^n-1.

$$\textbf{\textit{Máximo número decimal} = } 2^n\textbf{-1}$$

2.1.2.2.- La estructura de pesos de los números binarios

Como ya hemos dicho, el Sistema de numeración binario es un sistema de numeración posicional. Es decir, un número binario es un número con peso. El bit más a la derecha es el bit menos significativo (*LSB, Least Significant Bit*), y tiene un peso de 2^0=1. Los pesos de los respectivos bits crecen de derecha a izquierda según las potencias de 2 (2 es la base del sistema de numeración binario, así como 10 es la base del decimal). El bit más a la izquierda es el bit más significativo (*MSB, Most Significant Bit*), y su peso depende del tamaño del número binario.

Los números con parte fraccionaria también se pueden representar en binario colocando bits a la derecha de la coma, del mismo modo que los dígitos decimales fraccionarios. En un número binario con parte fraccionaria, el bit más a la izquierda tiene un peso de 2^{-1}=0,5. Los pesos fraccionarios de los respectivos bits decrecen de izquierda a derecha según las potencias negativas de 2.

La estructura de pesos de un número binario es:

$$2^{n-1}\ldots 2^3\ 2^2\ 2^1\ 2^0, 2^{-1}\ 2^{-2}\ldots 2^{-n}$$

Todos los bits a la izquierda de la coma tienen pesos que son potencias positivas de 2 Todos los bits a la derecha de la coma tienen pesos que son potencias negativas de 2.

2.1.3.- NÚMEROS HEXADECIMALES

El sistema de numeración hexadecimal consta de 16 dígitos y caracteres alfabéticos. Es por tanto un sistema en base 16. Cada dígito hexadecimal se representa mediante un número binario de 4 bits, como se puede ver en la Tabla 2.

Decimal	Hexadecimal	Binario
0	0	0 0 0 0
1	1	0 0 0 1
2	2	0 0 1 0
3	3	0 0 1 1
4	4	0 1 0 0
5	5	0 1 0 1
6	6	0 1 1 0
7	7	0 1 1 1
8	8	1 0 0 0
9	9	1 0 0 1
10	A	1 0 1 0
11	B	1 0 1 1
12	C	1 1 0 0
13	D	1 1 0 1
14	E	1 1 1 0
15	F	1 1 1 1

Tabla 2

Diez dígitos numéricos y seis caracteres alfanuméricos forman el sistema de numeración hexadecimal. El uso de las letras A, B, C, D, E y F para representar números puede parecer extraño al principio, pero hay que tener en mente que cualquier sistema de numeración es sólo un conjunto de símbolos secuenciales. Únicamente necesitamos comprender qué cantidades representan estos símbolos. Entonces la forma de los símbolos en sí tiene poca importancia. Utilizaremos el subíndice 16 o la letra "h" después del número, para designar a los números hexadecimales y evitar así cualquier confusión con los números decimales.

Utilidad de los números hexadecimales

Puesto que las computadoras y microprocesadores sólo entienden los 1's y los 0's, es necesario emplear estos dígitos cuando se programa en "lenguaje máquina". Imaginemos tener que escribir una instrucción de 60 bits para un sistema de mircroprocesador utilizando 0's y 1's. Los números binarios largos son difíciles de leer y escribir, ya que es fácil omitir o cambiar un bit. Es por tanto mucho más efectivo utilizar los números hexadecimales.

Resumiendo:

- El sistema hexadecimal con sus 16 dígitos es un sistema en base 16.
- Los 16 dígitos hexadecimales son los comprendidos entre el 0 y el 9 seguidos de las letras comprendidas entre la A y la F.
- Al igual que en los sistemas de numeración decimal y binario, la posición de un dígito en un número hexadecimal indica su peso o valor dentro del número. Los pesos de un número binario están basados en las potencias de 16.

2.1.3.1.- Contar en hexadecimal

¿Cómo se cuenta en hexadecimal una vez que se ha llegado a la F?. Sencillamente se inicia otra columna y se continúa así:

10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F,20,21,22,23…

Con 2 dígitos hexadecimales se puede contar hasta FFh, que corresponde al decimal 255. Para continuar contando, son necesarios 3 dígitos hexadecimales. Por ejemplo 100h es el 256 decimal, 101h es el 257 decimal, y así sucesivamente. El máximo número hexadecimal de 3 dígitos es FFFh, es decir, el decimal 4095.

2.2.- CONVERSIONES ENTRE SISTEMAS DE NUMERACIÓN

2.2.1.- Conversión de binario a decimal

El valor decimal de cualquier número binario se puede determinar sumando los pesos de todos los bits que son 1, y descartando los pesos de todos los bits que son 0.

Ejemplo 1: Convertir el número entero binario 1101101 a decimal:

Se determina el peso de cada bit que está a 1, y luego se obtiene la suma de los pesos para obtener el número decimal.

2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	1

$$1101101_2=2^0+2^2+2^3+2^5+2^6=1+4+8+32+64=109_{10}$$

Ejemplo 2: Convertir el número fraccionario binario 110110,01 a decimal:

2^5	2^4	2^3	2^2	2^1	2^0,	2^{-1}	2^{-2}
1	1	0	1	1	0,	0	1

$$110110,01_2=2^{-2}+2^1+2^2+2^4+2^5=0,25+2+4+16+32=54,25_{10}$$

2.2.2.- Conversión de hexadecimal a decimal

Para estas conversiones se pueden emplear dos procedimientos diferentes:

- Convirtiendo primero a binario para posteriormente convertir de binario a decimal
- Sumando los pesos en base 16

a) Si se emplea el primero de los procedimientos habrá que convertir el número hexadecimal a binario (cada carácter hexadecimal se corresponde a 4 bits en binario) y, posteriormente, se hará la conversión binario-decimal como hemos visto en el punto 2.2.1. Veámoslo con ejemplos:

Ejemplo 1: Convertir a decimal el siguiente número hexadecimal: 1Ch.

1. En primer lugar hay que convertir a binario el número hexadecimal: $0001\ 1100_2$
2. En segundo lugar convertiremos el número binario a decimal como vimos en la sección 2.2.1.: $2^4+2^3+2^2=28_{10}$

Ejemplo 2: Convertir a decimal el siguiente número hexadecimal: A85h.

1. En primer lugar hay que convertir a binario el número hexadecimal: $1010\ 1000\ 0101_2$
2. En segundo lugar convertiremos el número binario a decimal como vimos en la sección 2.2.1.: $2^{11}+2^9+2^7+2^2+2^0=2693_{10}$

b) El segundo procedimiento se basa en la suma de pesos, de forma análoga a como vimos en el apartado 2.2.1. Habrá que tener en cuenta que dichos pesos, así como en el sistema binario eran potencias de 2, en el sistema hexadecimal son potencias de 16.

Para un número hexadecimal fraccionario con 6 dígitos, 2 de ellos fraccionarios, los pesos son:

16^3	16^2	16^1	16^0	16^{-1}	16^{-2}
4096	256	16	1	0,0625	0,00390625

El siguiente ejemplo muestra este método de conversión:

Ejemplo 3: Convertir a decimal el siguiente número hexadecimal: E5,3h

$$E5h=E*16^1+5*16^0+3*16^{-1}=14*16+5*1+3*0,0625=224+5=229,0625_{10}$$

2.2.3.- Conversión de decimal a binario

Para estas conversiones se pueden emplear dos procedimientos diferentes:

- Por divisiones y multiplicaciones sucesivas
- Por restas sucesivas.

En cualquier caso debe separarse la parte entera de la fraccionaria del número expresado en decimal.

a) Si se emplea el primero de los procedimientos, el de las *divisiones y multiplicaciones sucesivas*, las divisiones sucesivas se aplicarán a la parte entera mientras que las multiplicaciones deberán hacerse sobre la parte fraccionaria. Veamos con un ejemplo cómo debe procederse (Figura 2). Sea el número decimal 19,3125

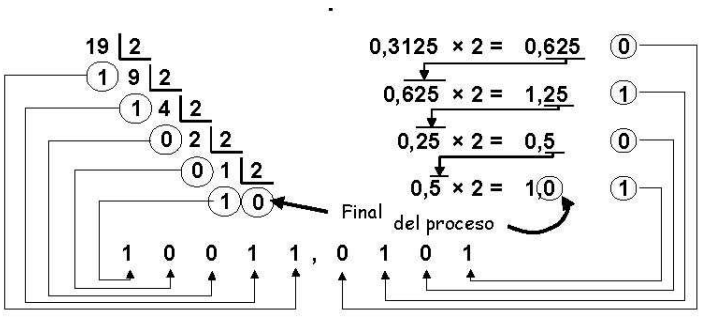

Figura 2

Para convertir un número decimal a binario convertiremos por separado la parte entera y la fraccionaria:

- *Parte entera* (19 en el ejemplo)

Se divide por 2 (base del sistema al que se quiere convertir)

El resto de esta división (1) será la primera cifra a la izquierda de la coma

El cociente (9) se vuelve a dividir por 2

El resto (1) será la segunda cifra a la izquierda de la coma

El cociente (4) se vuelve a dividir por 2

Se continúa el proceso hasta que el cociente sea 0.

- *Parte fraccionaria (0,3125 en el ejemplo)*

Se multiplica por 2 (base del sistema al que se quiere convertir)

La parte entera del producto (0) será la primera cifra a la derecha de la coma

La parte fraccionaria de ese producto (0,625) se vuelve a multiplicar por 2

La parte entera (1) será la cifra siguiente a la derecha de la coma

La parte fraccionaria (0,25) se vuelve a multiplicar por 2

Se continúa el proceso hasta que la fracción sea cero (como en el ejemplo), o hasta que tengamos un número suficiente de cifras fraccionarias.

b) Para hacer la misma conversión por *restas sucesivas* se debe preparar una tabla con las sucesivas potencias de 2, desde 2^0 hasta 2^n, siendo n tal que 2^n sea mayor o igual que la parte entera del número a convertir. En nuestro ejemplo (Figura 3), bastaría llegar hasta 2^5 aunque se ha incluido un elemento más.

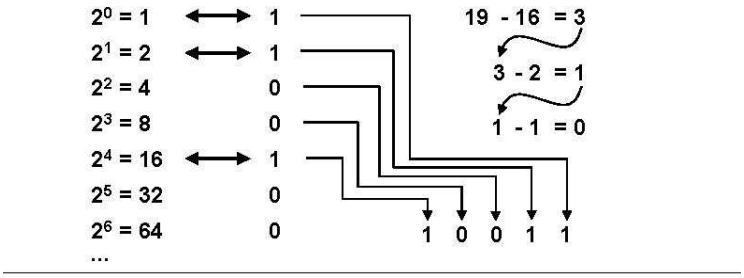

Figura 3

Confeccionada la tabla, el proceso a seguir es el siguiente:

- Parte entera (19):

a) Se toma el mayor número que <u>*no sea superior*</u> a la parte entera, en nuestro caso el 16 (2^4), lo que nos indica que el número binario resultante deberá tener un '1' en la posición 4 a la izquierda de la coma.

b) Después se resta del número original el valor de la potencia seleccionada anteriormente (19-16=3) y con esta diferencia se repite el proceso. Vemos que el mayor número de la tabla de potencias que no es superior a 3 es el 2 (2^1). Esto nos indica que el número binario resultante deberá tener un '1' en la posición 1 a la izquierda de la coma.

c) Se repite el proceso de resta anterior, (3 – 2=1) y se vuelve a comparar con los elementos de la tabla. Ahora es el primer elemento, con el valor 1 (2^0), el que resulta seleccionado lo que significa que habrá un '1' en la posición 0 a la izquierda de la coma.

d) Se vuelve a restar este número (1-1=0), momento en el que finaliza este proceso repetitivo.

El número binario, en su parte entera, queda formado por '1' en las posiciones seleccionadas (en nuestro caso las posiciones 4, 1 y 0 a la izquierda de la coma) y '0' en las restantes (las posiciones 2 y 3).

- Parte fraccionaria (0,3125):

Para la parte fraccionaria, el proceso es similar, aunque hay que tener en cuenta que probablemente no se pueda representar en binario con un número finito de cifras el número decimal deseado. Se deberá preparar una tabla de potencias con exponentes negativos de 2 cuyo número de elementos sea igual al número de cifras fraccionarias que se desea obtener (Figura 4).

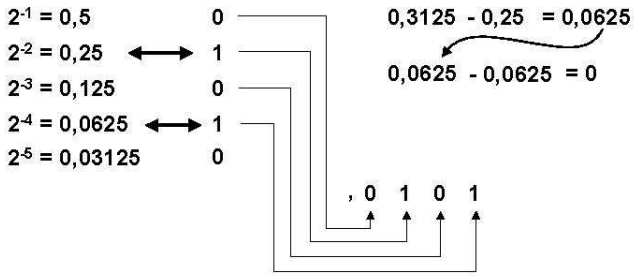

Figura 4

El proceso, por lo demás, es similar al seguido para la parte entera:

a) Tomamos el mayor valor de la tabla que no sea superior a la parte fraccionaria, en nuestro caso 0,25 (o sea, 2^{-2}). Esto supone que en la posición -2 (segunda a la derecha de la coma) debe haber un '1'.

b) Se resta este valor del inicial (0,3125-0,25=0,0625) y se vuelve a comparar con los elementos de la tabla. El mayor, no superior a éste 0,0625 es precisamente ese mismo valor, y corresponde a 2^{-4} lo que significa que habrá un '1' en la posición -4. Como el resultado de esta resta es exactamente 0 (0,0625-0,0625=0), el proceso habrá finalizado. Se colocará un '1' en las posiciones -2 y -4 a la derecha de la coma y un '0' en las restantes.

Obsérvese que el orden de las posiciones a la izquierda de la coma comienza en el 0, mientras que a la derecha lo hace en el -1.

Como hemos dicho, es muy posible que el número que se desee convertir de decimal a binario, no pueda representarse en binario con un número finito de cifras fraccionarias. Puede verse, como ejemplo, el número 19,45. Este número, en binario sería 10011,0111001, suponiendo que se considere suficiente la precisión obtenida con siete cifras fraccionarias.

2.2.4.- Conversión de hexadecimal a binario

Para convertir un número hexadecimal en un número binario se reemplaza cada símbolo hexadecimal por el grupo de cuatro bits adecuado, como vemos en el ejemplo:

Ejemplo: Determinar el número binario correspondiente al número hexadecimal 10A4h.

10A4h → $0001\ 0000\ 1010\ 0100_2$

Como ya hemos comentado, es evidente que es mucho más fácil tratar con un número hexadecimal que con el número binario equivalente. Puesto que la conversión es fácil, el sistema hexadecimal se usa ampliamente para representar los números binarios en programación, salidas de impresora y displays.

2.2.5.- Conversión de decimal a hexadecimal

Veamos como ejemplo el número decimal 1998,3125. Si empleamos el método de divisiones/multiplicaciones sucesivas, el proceso es similar, aunque el número a emplear para las divisiones o multiplicaciones sucesivas será el 16, base del sistema hexadecimal (Figura 5):

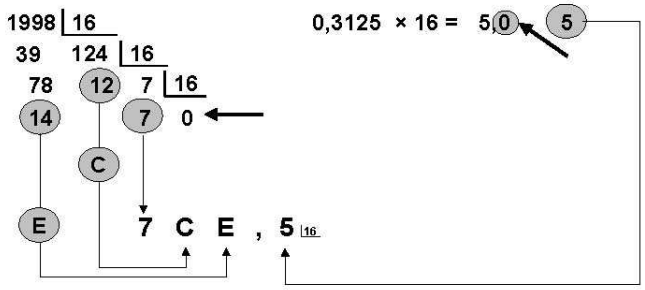

Figura 5

- *Parte entera* (1998 en el ejemplo)

Se divide por 16 (base del sistema al que se quiere convertir)

El resto de esta división (14) será la primera cifra a la izquierda de la coma (posición '0'). (naturalmente este valor 'catorce' se debe expresar en hexadecimal, es decir, por la letra 'E')

El cociente (124) se vuelve a dividir por 16

El resto (12='C' en hexadecimal) será la segunda cifra a la izquierda de la coma

El cociente (7) se vuelve a dividir por 16

El resto (7) será la posición '2'

El cociente es 0, con lo que termina el proceso para la parte entera

- *Parte fraccionaria (0,3125 en el ejemplo)*

Se multiplica por 16 (base del sistema al que se quiere convertir)

La parte entera del producto (5) será la primera cifra a la dcha. de la coma

La parte fraccionaria es 0: ya se ha terminado el proceso de conversión

2.2.6.- Conversión de binario a hexadecimal

El paso de binario a hexadecimal es extraordinariamente simple. Para ello basta dividir el número binario que se desea expresar en hexadecimal, en grupos de cuatro cifras (cuatro bits) comenzando por la coma a izquierda y derecha de la misma. Si algún grupo, como ocurre en el ejemplo, tiene menos de 4 bits, se rellena con ceros, hacia la izquierda para la parte entera o hacia la derecha para la parte fraccionaria, lo que no modifica el valor del número original.

Una vez formados estos grupos de 4 bits, basta reemplazar cada uno de ellos por el valor hexadecimal que lo representa.

Ejemplo: Convertir a hexadecimal el número binario: $11111000101101001,11011_2$
0011-1111-0001-0110-1001,1101-1000 -> 3-F-1-6-9-D-8 = 3F169D8h

El sistema hexadecimal no se utiliza realmente para realizar operaciones aritméticas. Sin embargo, repetimos, es extraordinariamente útil y ampliamente utilizado para representar series de bits ya sea en una pantalla o en una impresora. La representación directa en binario resulta farragosa y su interpretación es proclive a errores humanos. Es mucho más cómodo escribir, por ejemplo, 8AFCB51 que no su equivalente binario 1000101011111100101101010001.

La conversión en sentido contrario es igualmente inmediata: basta reemplazar cada cifra hexadecimal por los cuatro bits que, en binario, representan su valor.

2.3.- ARITMÉTICA BINARIA

La aritmética binaria es esencial en todas las computadoras digitales y en muchos otros tipos de sistemas digitales. Para entender los sistemas digitales, se deben conocer los principios básicos de la suma, resta, multiplicación y división binarias.

Las operaciones aritméticas en el sistema binario se ejecutan de forma similar a las realizadas en el sistema decimal, aunque empleando únicamente los dígitos 0 y 1.

2.3.1.- SUMA BINARIA

Las cuatro reglas básicas para sumar dígitos binarios son:

0 + 0 = 0	Suma 0 con acarreo 0
0 + 1 = 1	Suma 1 con acarreo 0
1 + 0 = 1	Suma 1 con acarreo 0
1 + 1 = 0	Suma 1 con acarreo 1

La única situación 'especial' es la que se produce cuando en la suma el valor resulta superior a 1. Es similar a lo que ocurre cuando en decimal se suman dos dígitos cuya suma es superior a 9; al efectuar la suma decimos "me llevo una" o, de una manera más formal, "se produce acarreo". Es decir, se genera una unidad de orden superior.

En esta situación, al sumar en binario, lo que hacemos es añadir '1' a la cifra de orden superior, lo que representamos con ese pequeño '1' sobre los sumandos originales en la Figura 6.

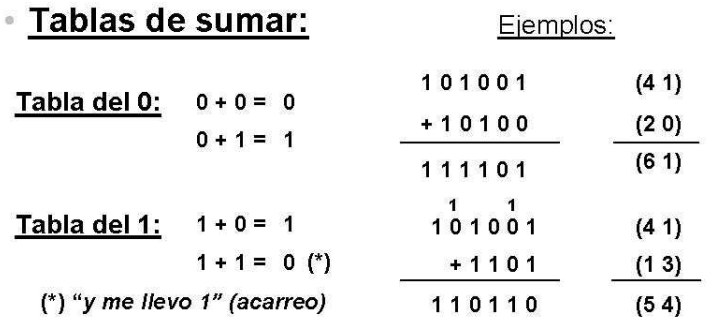

Figura 6

Ejemplos: Sumar los siguientes números binarios

a) 11 + 11

$$\begin{array}{cccc} & & 1 & 1 \\ + & & 1 & 1 \\ \hline & 1 & 1 & 0 \end{array}$$

b) 100 + 10

$$\begin{array}{cccc} & 1 & 0 & 0 \\ + & & 1 & 0 \\ \hline & 1 & 1 & 0 \end{array}$$

c) 111 + 11

$$\begin{array}{cccc} & 1 & 1 & 1 \\ + & & 1 & 1 \\ \hline & 10 & 1 & 0 \end{array}$$

2.3.2.- RESTA BINARIA

Igual que con la suma binaria, veamos qué ocurre en el sistema decimal; el problema se plantea cuando se trata de restar de un dígito otro de mayor valor (por ejemplo, restar 8 de 5) y entonces debemos 'tomar prestada' una unidad de orden superior para poder realizar la resta (restamos 8 de 15).

En binario se presenta cuando se trata de restar 1 de 0; también entonces 'tomamos prestada' una unidad de orden superior y realizamos la resta desde el valor 10 (dos), pero deberemos 'devolver' esa unidad de orden superior que hemos 'tomado prestada', lo que representamos con ese '-1' que aparece en la Figura 7.

- **Tablas de restar:**

Tabla del 0:
0 – 0 = 0
0 – 1 = 1 (*)

(*) *"y me llevo -1"*

Tabla del 1:
1 – 0 = 1
1 – 1 = 0

Ejemplos:

1 0 1 0 0 1	(4 1)
– 1 0 0 0	(8)
1 0 0 0 1	(3 3)

-1 -1 -1	
1 0 1 0 0 1	(4 1)
– 1 1 0 1	(1 3)
0 1 1 1 0 0	(2 8)

Figura 7

Ejemplos: Realizar las siguientes restas binarias

a) 11 - 01

$$\begin{array}{r} 1\ 1 \\ -\ \ 1 \\ \hline 1\ 0 \end{array}$$

b) 11 - 10

$$\begin{array}{r} 1\ 1 \\ -\ 1\ 0 \\ \hline \ \ \ 1 \end{array}$$

c) 101 - 11

$$\begin{array}{r} 1\ 0\ 1 \\ -\ 1\ 1 \\ \hline 1\ 0 \end{array}$$

2.3.3.- MULTIPLICACIÓN BINARIA

La última operación binaria que vamos a considerar es la multiplicación. La multiplicación en binario resulta extraordinariamente simple puesto que se realiza de la misma forma que con números decimales: el multiplicando deberá multiplicarse sucesivamente por cada dígito del multiplicador, desplazando una posición a la izquierda cada resultado obtenido. Este resultado será '0' cuando el dígito correspondiente del multiplicador sea '0' (multiplicar por 'cero' es 'cero' en cualquier sistema de numeración) o será igual al multiplicando cuando la cifra correspondiente en el multiplicador sea '1' (multiplicar por 'uno' no cambia el multiplicador, en cualquier sistema de numeración). Veamos un ejemplo (Figura 8):

Multiplicar 10011 × 101:

Multiplicando:	**10011**	
Multiplicador:	**× 101**	
	10011	Multiplicar por '1' (igual al multiplicando)
	00000	Multiplicar por '0' y desplazar a la izquierda
	10011	Multiplicar por '1' (igual al multiplicando) y desplazar a la izquierda
Resultado:	**1011111**	Sumar en binario

Figura 8

2.4.- REPRESENTACIONES BINARIAS

La representación de números en binario sería perfecta si no hubiese que emplear nada más que 'ceros' y 'unos', ya que el ordenador sólo puede emplear estos símbolos para representar cualquier información. Pero en la realidad se hace imprescindible utilizar el signo 'menos' para representar números negativos y la 'coma' para separar parte entera y fraccionaria. Y esto refiriéndonos de momento sólo a información numérica en binario, sin entrar en cómo representar información numérica decimal, alfabética o de otro tipo. Por ejemplo, ¿cómo almacenar en el ordenador el número binario -010011,1101?

Para solucionar este problema recurrimos a lo que se conoce como *representaciones binarias* que no son sino un conjunto de normas que permiten representar, empleando sólo 'ceros' y 'unos', cualquier magnitud numérica (binaria, decimal, positiva, negativa, entera, fraccionaria...), alfabética o incluso de cualquier otro tipo.

Limitándonos, por el momento, a las representaciones numéricas para valores binarios, tenemos ciertas convenciones:

- En primer lugar, la coma que separa la parte entera y la fraccionaria se omite por completo. Es preciso alinear previamente los diferentes números de acuerdo con la posición de la coma y, a partir de entonces, considerar todos los números como enteros. Esta alineación de los números es responsabilidad del programa aunque, evidentemente, será una función hecha de forma automática por el compilador, de la que el programador no deberá preocuparse. El programa sabrá dónde está la coma, o lo que es lo mismo, cuantas cifras de la representación corresponden a la parte entera y cuántas a la fraccionaria.
- Una vez superado el problema de la coma, todos los números son enteros y sólo nos queda cómo representar números positivos o negativos, sin emplear para ello el signo 'menos'.

  Para esto hay al menos cinco alternativas:

  1. Representación en Módulo y Signo (MS)
  2. Representación en Complemento a 1 (C-1)
  3. Representación en Complemento a 2 (C-2)
  4. Representación en Exceso a 2^{N-1}
  5. Representación en Coma Flotante

Las cinco formas de representación binaria llevan, en la práctica, una limitación. Todas ellas tienen un número prefijado de bits. En un determinado lenguaje habrá representaciones, por ejemplo, con 32 bits o con 16 bits, y en otro las podrá haber con 64, 32 o 16 bits, etc. Así, en el primer lenguaje no se podrán representar, con estas representaciones binarias, números que requieran más de 32 bits, ni en el segundo lenguaje números que requieran más de 64 bits.

Características de una Representación Numérica Binaria

Sea cual sea la representación numérica utilizada, hay dos características principales ligadas a ellas.

a) *Rango de esa representación*: Define el conjunto de números representables. Cualquier representación tiene limitado el número de bits a utilizar que, según los casos, podrá ser mayor o menor, pero siempre tendrá un límite. Este límite establece el mayor número posible a representar. Como cualquier representación numérica debe ser capaz de representar números negativos, el número de bits establece también el número más pequeño que se puede representar (entendiendo por más pequeño el número negativo de mayor valor absoluto).
b) *Forma de representar el valor 'cero'*: Hay representaciones que únicamente permiten una representación para este valor, mientras que otras permiten más de una.

2.4.1.- REPRESENTACIÓN EN MÓDULO Y SIGNO (MS)

La forma de representación numérica binaria más sencilla es la conocida como representación en Módulo y Signo (MS). Con esta representación, dado siempre un número determinado y fijo de bits para representar un valor, se reserva:

- El primer bit para indicar el signo del número
- El resto para representar su valor absoluto (su valor absoluto en binario)

Su rango es simétrico y varía entre los márgenes siguientes:

$$-(2^{N-1}-1) \leq X \leq 2^{N-1}-1$$

siendo N el número de bits empleado en la representación, incluido el bit de signo.

Según el número de bits empleado en la representación, el rango varía entre los valores que se indican a continuación:

a) con 8 bits: -127 y 127
b) con 16 bits: - 32767 y +32767
c) con 24 bits: - 8388607 y +8388607

La representación del valor cero es doble ya que, dado su valor absoluto (tantos ceros como bits tenga la representación menos uno), se le puede anteponer como bit de signo tanto el '0' (positivo) como el '1' (negativo). Así, empleando por ejemplo 16 bits, su representación sería 000000000000000 o bien 100000000000000.

Esta representación podría servir para almacenar en un ordenador números, positivos o negativos, pero resulta muy poco útil para realizar operaciones aritméticas con ellos.

2.4.2.- REPRESENTACIÓN EN COMPLEMENTO A 1 (C-1)

La representación en Complemento a 1 (C-1) es otra forma de representar y de almacenar en un ordenador números enteros. En esta representación también se reserva el primer bit para indicar el signo del valor a representar. Esta forma de representación tiene las siguientes propiedades:

- Los números positivos se representan exactamente igual que en la representación con Módulo y Signo.
- Los números negativos, sin embargo, se representan complementando el número positivo de igual valor absoluto. Para obtener su representación basta con cambiar todos los bits '0' por '1' y viceversa, incluyendo el bit de signo.

Ejemplo 1: Representar los siguientes valores decimales en C-1 empleando 8 bits:

a) +67

   0100 0011 ($2^0+2^1+2^6=1+2+64=67$)

b) -67

   Deberemos cambiar todos los '0' por '1' y viceversa, con lo que su representación será 1011 1100

Para conocer el valor de un número negativo en esta representación, número que comenzará necesariamente por '1', bastará complementarlo, con lo que se tiene la representación de ese mismo número, pero positivo, y con ello su valor absoluto.

Ejemplo 2: Obtener el valor del número 10011001 representado en C-1 con 8 bits.

Por comenzar por '1' sabemos que se trata de un número negativo. Complementaremos esa representación, resultando 01100110 cuyo valor, positivo, es 102.

El valor del número 1001 1001 es, por tanto, -102.

El rango de esta representación es el mismo que en la representación MS y también la representación del 'cero' es doble: empleando 8 bits: tanto '00000000' como '11111111' son representaciones válidas para este valor.

Tampoco esta forma de representación es empleada para la realización de operaciones aritméticas.

2.4.3.- REPRESENTACIÓN EN COMPLEMENTO A 2 (C-2)

La representación en C-2 es la más empleada para realizar operaciones aritméticas con números binarios en un ordenador. En ella los valores positivos se representan igual que en MS (o, lo que es lo mismo, igual que en C-1), mientras que los valores negativos se representan obteniendo el Complemento a dos (C-2) de la representación del número positivo de igual valor absoluto.

Para obtener este complemento a dos, basta con:

- Obtener el complemento a 1 (cambiar '0' por '1' y viceversa) y a continuación,
- Sumar 1 al número resultante despreciando el acarreo en la suma, si lo hubiera.

Ejemplo 1: representar el valor -67 en C-2:

1. Partiremos de la representación de su valor positivo +67, por ejemplo, con 8 bits: 01000011
2. obtenemos su C–1, cambiando '0' por '1' y viceversa: 10111100
3. y a este C-1 le sumamos 1: 10111101

La representación del valor -67 en C–2 resulta ser, pues: **10111101**.

Ejemplo 2: Si el valor a representar fuera -12, también con 8 bits:

1. Partiremos de la representación de su valor positivo +12: 00001100
2. obtenemos su C–1, cambiando '0' por '1' y viceversa: 11110011
3. y a este C-1 le sumamos 1: 11110100

La representación del valor –12 en C–2 resulta ser, pues: **11110100**.

Transformación inversa: para conocer el valor de un número negativo representado en C-2 debe obtenerse su complemento a dos, (cambiar '0' por '1' y viceversa y sumar 1 al resultado anterior). En este caso no se produce nunca acarreo.

Ejemplo 3: suponiendo la representación en C-2 con 8 bits, 10011110, calcule el número decimal representado:

> Su C-1 será 01100001 y sumando 1 a este valor obtendremos 01100010, que es el complemento a dos del número original y que representa el valor 98, por lo que el valor del número de partida representado en C-2 (10011110) no es otro que **-98**.

El rango de esta representación varía de acuerdo con la siguiente expresión:

$$-2^{N-1} \leq X \leq 2^{N-1}-1$$

a la que, según el número de bits empleados, corresponden los valores:

a) con 8 bits: – 128 y +127
b) con 16 bits: – 32768 y +32767
c) con 24 bits: – 8388608 y + 8388607

Obsérvese que el rango de esta representación de C-2 es asimétrico mientras que la representación del cero es única: '00000000' (la asimetría en el rango de una representación no representa ningún problema en la práctica, mientras que la representación única del valor cero es muy conveniente).

Importante: No debemos olvidar que los números positivos se representa exactamente igual en:

- MS
- C-1
- C-2.

2.4.3.1.- Suma en Complemento a 2

La representación en C–2 es la más empleada en la práctica para realizar operaciones aritméticas. Comenzando con la Suma aritmética, el procedimiento es muy sencillo:

Para efectuar una suma, tanto con números positivos como negativos, nos limitaremos a sumar en binario las representaciones en C–2 de ambos números, despreciando el acarreo si lo hubiera.

Ahora bien, puede ocurrir que el resultado obtenido no sea correcto. La representación de ambos sumandos debe tener necesariamente el mismo número de bits y también el resultado. Sin embargo, es posible que el valor de la suma esté fuera del rango correspondiente a ese número de bits.

Ejemplo 1: Se pretende sumar los valores 57 y 102, ambos positivos, empleando una representación de 8 bits.

> El resultado debería ser 159, valor que cae fuera del rango de la representación con los 8 bits supuestos ($-128 \leq X \leq 127$). Esto es algo inevitable; podría emplearse una representación con 16 bits, pero lo único que conseguiríamos sería elevar el listón donde se presentaría el problema.

De hecho este problema se presenta en los ordenadores y lo único que éstos pueden hacer es avisar al programador de que se ha producido un resultado incorrecto para que éste tome las medidas que considere oportunas. Este fenómeno se conoce como *desbordamiento* ('*overflow*', en inglés).

¿Cómo se puede detectar que se produce desbordamiento en una suma en C-2?: Sigamos la siguiente regla:

- Si los dos sumandos tienen igual signo, el signo del resultado debe ser también el mismo, de lo contrario se produce desbordamiento y el resultado obtenido no es el correcto.
- Cuando los sumandos tienen signos diferentes, el resultado siempre es correcto.

Veamos algunos ejemplos, por sencillez, siempre con 8 bits:

Ejemplo 2: Sumar 00110011 y 01000111 (Figura 9, lado izquierdo)

El resultado es 01111010. Como el signo de los dos sumandos es el mismo (positivo) y también el resultado tiene este mismo signo, el resultado es correcto.

Puede observarse que el valor de los sumandos es 51 y 71, mientras que el valor de la suma es 122, evidentemente correcto y dentro del rango de la representación.

Ejemplo 3: Sumar 00110011 y 01100000 (Figura 9, lado derecho)

El resultado es 10010011. El signo del resultado es diferente al de los sumandos, lo que indica que se ha producido desbordamiento y que este resultado no es correcto. ¿Qué ha ocurrido? Muy sencillo, el valor de los sumandos, ambos positivos, es 51 y 96; la suma debería ser 147, pero este valor cae fuera del rango de la representación con 8 bits. Por otra parte, se puede ver cómo el resultado obtenido es un número negativo (por comenzar por '1'), con lo que es evidente que no puede ser correcto.

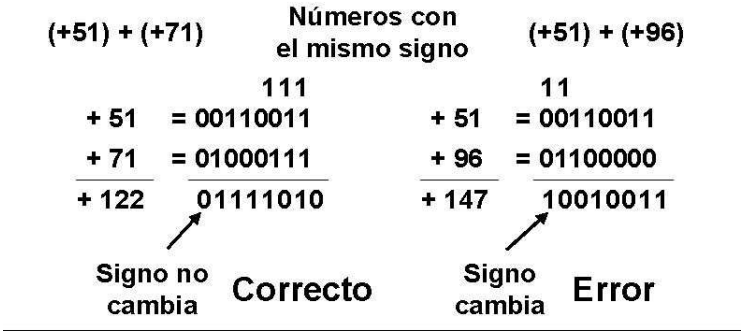

Figura 9

Si se hubiese efectuado esta misma suma con 16 bits, el resultado sí habría sido correcto; 'cabría' en el rango de esa representación.

Ejemplo 4: Sumar 10001111 y 11111101

El resultado, despreciando el acarreo, es 10001100, el signo es el mismo que el de los dos operandos (todos son negativos, al comenzar por 1) y, por tanto el resultado es correcto. De hecho, se trata de los valores -113 y -3, cuya suma (-116) cae dentro del rango.

Veamos ahora algún ejemplo con números de distinto signo.

Ejemplo 5: Sumar 11001101 y 01000111 (Figura 10, lado izquierdo)

El resultado de la suma es, despreciando el acarreo como indican las reglas de la suma, 00010100. Como los dos números a sumar tienen signo diferente, el resultado es correcto sin tener que atender a ninguna otra consideración. El valor de los sumandos expresado en decimal es, respectivamente, -51 y +71, siendo +20 el valor del resultado obtenido, que coincide, naturalmente, con el valor de la suma.

Ejemplo 6: Sumar 00110011 y 10100000 (Figura 10, lado derecho)

Ahora el resultado de la suma es 11010011. En este caso no se ha producido acarreo. La operación, en decimal, sumaría los números +51 y -96 cuyo resultado (-45) coincide con el obtenido.

Resumen: cuando ambos números tienen signo contrario, el resultado es siempre correcto. Puede producirse acarreo (en cuyo caso se desprecia) o no y el resultado puede ser tanto positivo como negativo.

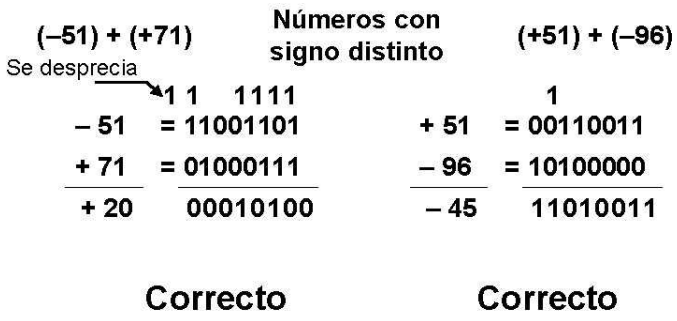

Figura 10

2.4.3.2.- Resta en Complemento a 2

Analicemos ahora la resta en C–2.

Para realizar la resta de dos números cualesquiera expresados en C–2,

- Deberá hallarse el C–2 del sustraendo y, a continuación,
- Se sumará este número con el minuendo sin modificación alguna y *empleando las reglas de la suma antes descritas.*

En realidad, lo que estamos haciendo en el primer paso no es sino cambiar de signo el sustraendo, con lo que sumándolo al minuendo, evidentemente, obtendremos el valor de la resta deseada.

Veamos ahora algún ejemplo.

Ejemplo 1: Efectuar la resta 01000111 – 00110011 (Figura 11)

De acuerdo con las reglas, deberemos:

1. Calcular el C–2 del sustraendo (00110011): tras intercambiar '0' por '1' y sumar 1, este C-2 resulta ser 11001101.
2. Sumar este número con el minuendo (01000111). El resultado de esta suma es 00010100. ***El acarreo producido en esta suma debe despreciarse.***

(+71) – (+51)

+ 51	= 00110011	Se toma el sustraendo
	11001100	Se cambian 0's por 1's
	+ 1	Se suma 1
– 51	11001101	Se tiene el C-2
+ 71	= 01000111	Se toma el minuendo
+ 20	00010100	Se suma normalmente

Signos diferentes, resultado correcto

Figura 11

El resultado obtenido es **correcto**, puesto que los números que se suman son de diferente signo. De hecho, expresando en decimal los valores de esta operación, estamos restando 71-51, cuyo resultado es 20, que coincide con el obtenido.

Ejemplo 2: Efectuar la resta 10100111 - 00110011

De acuerdo con las reglas, deberemos:

1. Calcular el C–2 del sustraendo. El mismo que en el primer caso: 11001101.
2. Sumar este número con el minuendo (10100111). El resultado de esta suma, despreciando el acarreo, es 01110100.

El resultado obtenido es **incorrecto**: Los números que se suman tienen igual signo mientras que el resultado tiene signo diferente. Obsérvese que al aplicar las reglas de la suma debemos atender a los números de esta suma y no a los de la resta inicial. En este caso, se pretende realizar la resta -89-51, cuyo resultado es -140, valor que no coincide con el obtenido. La razón es clara: el valor -140 "no cabe", es decir, cae fuera del rango de la representación en C–2 con 8 bits.

Con las reglas mencionadas para efectuar la suma o la resta de valores expresados en C–2 podemos llegar a una conclusión muy interesante:

Para efectuar cualquier suma o resta en C–2, sólo necesitamos realizar sumas binarias.

La conclusión es muy importante, como veremos en temas posteriores, pues necesitaremos un único tipo de circuito: el sumador, en lugar de precisar sumadores y restadores. Desde luego también se deberá complementar el sustraendo antes de sumarlo al minuendo pero, como veremos, esto resulta extremadamente simple.

2.4.4.- REPRESENTACIÓN EN EXCESO A 2^{N-1}

Con la representación *Exceso a* 2^{n-1}, se representa en binario puro, sin signo, no el valor binario deseado, sino este valor incrementado en 2^{n-1}, donde n es el número de bits empleado en la representación. El propósito de este desplazamiento es poder definir números muy grandes o muy pequeños sin necesidad de emplear un bit de signo.

Así, para representar el valor 12 (con 8 bits) se suma a esta cantidad el valor $2^{n-1} = 2^{8-1} = 128$ y el resultado (140) se representa en binario puro, sin signo obteniéndose 10001100.

Si deseásemos representar el valor -12, tras sumarle 128 el valor a representar en binario sería el 116, que en binario puro es 01110100.

- Obsérvese que en este caso un '0' al comienzo del número no significa que éste sea positivo, ni un '1' significa que sea negativo. De hecho, ocurre exactamente lo contrario (con la excepción del valor cero, que no puede considerarse ni positivo ni negativo y que se representa como 10000000).
- Además, a diferencia de las representaciones analizadas anteriormente, ahora el primer bit interviene en el cálculo del valor del número representado.

Como veremos en el próximo punto, esta representación forma parte de la representación en coma flotante empleada para el tratamiento de cantidades con valores absolutos muy grandes o muy pequeños, utilizadas generalmente en cálculos de naturaleza técnica o científica.

El rango de esta representación es el mismo que el de la representación en C–2 y, por tanto, asimétrico. Variará de acuerdo con la expresión siguiente:

$$-2^{N-1} \leq X \leq 2^{N-1} - 1$$

a la que, según el número de bits empleados, corresponden los valores:

a) con 8 bits: – 128 y +127
b) con 16 bits: – 32.768 y +32.767
c) con 24 bits: – 8.388.608 y + 8.388.607

La representación del cero es única: (0 + 128) = 10000000.

2.4.5.- REPRESENTACIÓN EN COMA FLOTANTE

Las representaciones binarias, o las BCD que veremos próximamente, son adecuadas en general para trabajos de tipo administrativo o de gestión, pero en otro tipo de trabajos su empleo no es satisfactorio, pues la precisión y el rango de estas formas de representación son relativamente pequeños, aun empleando bastantes bits.

Por ejemplo, con representaciones binarias, empleando 32 bits y teniendo en cuenta el bit de signo, puede llegarse sólo hasta 2.147.483.647 euros. Esta cantidad puede parecer suficiente para manejar, por ejemplo, el saldo de una cuenta corriente (no para los Presupuestos del Estado). Es más, si recordamos que en las representaciones numéricas no se incluyen las cifras fraccionarias, vemos que en ese mismo ejemplo el saldo máximo (utilizando dos cifras decimales) no podría ser superior a 214.748.836,47, cantidad que puede resultar insuficiente.

La solución evidente consiste en emplear un mayor número de bits. Con 64 bits, podríamos alcanzar hasta 92.233.720.368.547.758,08, o podríamos manejar cantidades de hasta 19 dígitos decimales (Obsérvese que, como hemos mencionado anteriormente, en este número de cifras debe incluirse tanto las cifras enteras como las fraccionarias).

Pero en los cálculos científicos la situación es muy diferente. Frecuentemente se debe trabajar con números mucho mayores o mucho más pequeños que los permitidos con representaciones binarias, e incluso, con ambos tipos de números a la vez.

Ejemplo 1: la masa de la Tierra, expresada en gramos es 5.980.000.000.000.000.000.000.000.000 gramos y la de un protón es 0,000000000000000000000001672623 gramos. ¿Cómo representamos estos valores?

En este caso el rango de las representaciones binarias es insuficiente. La solución empleada por los científicos (notación científica) consiste en poner estas cantidades en forma de potencia de 10:

$$5.980.000.000.000.000.000.000.000.000 = 5{,}98 \times 1027$$

$$0{,}000000000000000000000001672623 = 1{,}672623 \times 10{-}24$$

Por otra parte, también es relativamente frecuente el caso de operaciones aritméticas en las que se deben restar dos números muy próximos y el resultado multiplicarlo por un valor muy alto. Si el número de cifras con que se trabaja no es suficiente, el resultado final puede ser altamente inexacto.

Ejemplo 2: Supongamos que se deseara efectuar la siguiente operación:

$$(456{,}3454553491 - 456{,}3454553089) \times 500.000.000$$

cuyo resultado exacto es 20,1 pero imaginemos que únicamente se pudiera trabajar con un máximo de 10 dígitos. El ordenador debería truncar ambos números a un total de 10 cifras y la operación quedaría como

$$(456{,}3454553 - 456{,}3454553) \times 500.000.000$$

cuyo resultado es 0. El error, evidentemente, es enorme.

En este segundo ejemplo, el problema de una representación binaria estaría en la precisión con la que se puede representar un valor determinado.

Una forma más adecuada de representar números muy grandes, o muy pequeños, es hacerlo en forma de potencia. En la vida real, en potencias de 10. Esta forma de representación se conoce como representación en *Coma Flotante.*

$$N = mantisa \times base^{característica}$$

Los elementos que forman una representación en Coma Flotante son:

- *Mantisa*: el primer valor que aparece en la expresión.
- *Base*: la del sistema elegido (en la vida real, base 10)
- *Característica*: el exponente al que se debe elevar la base

Para representar el valor de una magnitud cualquiera en Coma Flotante habría que almacenar los valores de la mantisa, de la base y de la característica. El problema que surge a la hora de utilizar la representación en Coma Flotante es el de la multiplicidad de representaciones:

$$5{,}98 \times 10^{27} = 598 \times 10^{25} = 0{,}598 \times 10^{28}$$
$$1{,}67 \times 10^{-24} = 16{,}7 \times 10^{-25} = 0{,}167 \times 10^{-23}$$

Cualquiera de los valores indicados anteriormente son matemáticamente correctos, pero las comas "no se llevan bien" con las representaciones numéricas. ¿Cómo indicamos dónde está la coma? Simplemente, fijando siempre una misma posición, es decir, mediante una normalización:

Cuando un entero se expresa como un número en coma flotante, se normaliza desplazando la coma decimal a la izquierda de todos los dígitos, de modo que la mantisa es un número fraccionario y el exponente es una potencia de 10.

La representación normalizada de los ejemplos anteriores será:

$$0{,}598 \times 10^{28}$$
$$0{,}167 \times 10^{-23}$$

Una vez decidida cuál es la representación normalizada, para almacenarla no son necesarios ni el cero inicial ni la coma, puesto que ya se sabe que el número que describe la mantisa debe comenzar por estos caracteres. Tampoco es necesario almacenar la base, pues, dada una representación, se conoce cuál es su base. (En los ejemplos anteriores la base sería 10, aunque en los ordenadores no se emplee esa base).

Análogamente, para los números en coma flotante binarios necesitamos representar:

1. La mantisa
2. La característica
3. El signo (aunque hasta ahora no lo habíamos mencionado, lógicamente deberemos incluirlo para poder manejar magnitudes positivas y negativas).

Una vez vistos los elementos que deben almacenarse en la representación en Coma Flotante, hay que elegir cómo representar cada uno de ellos. Para homogeneizar la forma de representación por los diferentes fabricantes surgen los Organismos de Normalización, en este caso el IEEE (Institute of Electrical and Electronic Engineering), que establece:

- El *signo:* parece lógico que se represente por un único bit ('0' positivo y '1' negativo) y así lo determina el IEEE.
- La *característica* (exponente): podría representarse con cualquiera de las representaciones binarias que hemos visto. El IEEE la normaliza en forma de *Exceso a* $2^{N-1}-1$, siendo N el número de bits reservados para la característica. Este exceso o desplazamiento simplifica algunas operaciones al hacer el número siempre positivo.
- La *mantisa:* se normaliza en binario como 1,xxxx. Por tanto este 1 se entiende que estará siempre y no se incluye de forma explícita en la representación. Esta forma permite un bit más de precisión.

Dos son las formas de representación en Coma Flotante que están reguladas por Organismos de Normalización y que, por tanto, son empleadas por cualquier fabricante de ordenadores: ***simple precisión y doble precisión***. Ambas están recogidas en la norma IEEE754. El contenido de cada una de estas formas de representación con 32 o con 64 bits se indica en la Figura 12.

El que estas dos formas estén normalizadas facilita que se puedan intercambiar programas entre distintos suministradores, mientras se ajusten a estas normas

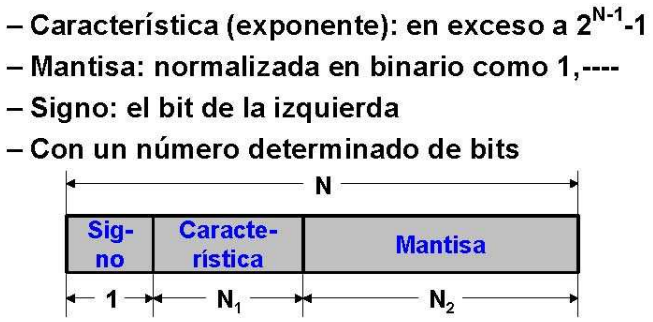

Figura 12

De todas formas, no es raro encontrar formas adicionales. Por ejemplo, en los grandes ordenadores de IBM existe, además de las formas normalizadas, la llamada ***precisión ampliada*** en la que cada

número ocupa un total de 128 bits.

En cualquiera de los casos la base, no incluida en la representación, es el 2.

Ejemplo: Ilustremos con un ejemplo cómo un número binario se expresa en formato de coma flotante. Se trata del número binario 1011010010001.

En primer lugar, podemos expresarlo como 1 más un número binario fraccionario, desplazando la coma binaria 12 posiciones a la izquierda y multiplicándolo después por la apropiada potencia de 2.

$$1011010010001 = 1{,}011010010001 * 2^{12}$$

a) Como se trata de un número positivo, el bit de signo es 0.
b) El exponente, 12, se expresa como un exponente desplazado añadiéndole 127 (12+127=139). El exponente desplazado se expresa como el número binario 10001011.
c) La mantisa es la parte fraccionaria del número binario (011010010001). Dado que siempre existe un 1 a la izquierda de la coma binaria en la expresión de la potencia de 2, no se incluye en la mantisa.

El número en coma flotante completo en simple precisión (32 bits) es:

0 10001011 01101001000100000000000

2.4.5.1.- Representación del Cero

Según las normas de la representación normalizada que hemos comentado, el valor del cero no podría representarse. La expresión numérica del cero sería $0{,}00000 * 2^0$. El número de 'ceros' puede ser tan grande como se desee pero la representación no sería una representación normalizada ya que ésta, por definición, exige que la primera cifra a la izquierda de la coma sea un '1'. Sin embargo, el cero tiene que poder ser representado necesariamente.

La solución adoptada consiste en codificar el valor 'cero' con todos los bits de la representación a '0', es decir, con 32, 64 ó 128 bits a '0'.

2.4.5.2.- Rango en Coma Flotante

La representación en Coma Flotante es útil para trabajar con cantidades muy grandes o muy pequeñas o, incluso, con magnitudes muy grandes y muy pequeñas simultáneamente, algo típico del cálculo técnico-científico. Veamos pues cuál es el rango de representación. Ahora hay que tener en cuenta cuál es el mayor valor absoluto que se puede representar y *además* cuál el menor valor absoluto representable (además del cero, siempre posible).

En la Figura 13, los números negativos menores que A (valor absoluto mayor que A), los comprendidos entre B y C (con excepción del cero) y los mayores que D no pueden representarse. El valor del rango depende de la representación elegida.

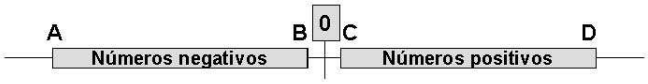

Figura 13

Veamos cuáles son estos rangos con 32 bits, aproximadamente:

- *Menor número negativo* (mayor valor absoluto, con signo negativo):

  Signo: –
  Mantisa = 1,11111....1 (23 cifras) = $1 + (1 - 2^{-23}) \approx 2$
  Exponente = 11111111 = 128 (128 en exceso a $2^{8-1} - 1$ equivale a 11111111)
  Por tanto, $A \approx -2^{129} \approx -6{,}8 \times 10^{38}$

- *Mayor número negativo (menor valor absoluto, con signo negativo):*

  Signo: –
  Mantisa = 1,000000.....0 (23 cifras) = 1
  Exponente = 00000000 = – 127 (127 en exceso a $2^{8-1} - 1$ equivale a 00000000)
  Por tanto, $B = -1 \times 2^{-127} = -2^{-127} \approx -5{,}9 \times 10^{-39}$

- *Menor número positivo (menor valor absoluto, con signo positivo):*

  Signo: +
  Mantisa = 1,000000.....0 (23 cifras) = 1
  Exponente = 00000000 = – 127 (127 en exceso a $2^{8-1} - 1$ equivale a 00000000)
  Por tanto, $C = 1 \times 2^{-127} = 2^{-127} \approx 5{,}9 \times 10^{-39}$

- *Mayor número positivo (mayor valor absoluto, con signo positivo):*

  Signo: +
  Mantisa = 1,11111....1 (23 cifras) = $1 + (1 - 2{-}23) \approx 2$
  Exponente = 11111111 = 128 (128 en exceso a $2^{8-1} - 1$)
  Por tanto, $D \approx 2 \times 2^{128} = 2^{129} \approx 6{,}8 \times 10^{38}$

En la Figura 14 se incluyen los valores correspondientes a estos rangos, empleando Precisión Simple,

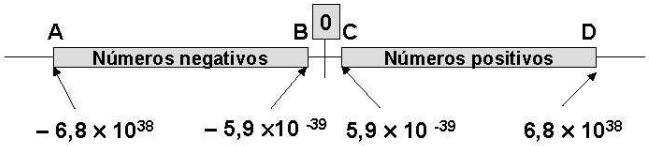

Figura 14

Naturalmente, con Precisión Doble, al emplearse mayor número de bits para el exponente, el rango es mucho mayor. En la Figura 15 se incluyen los valores correspondientes con Precisión Doble.

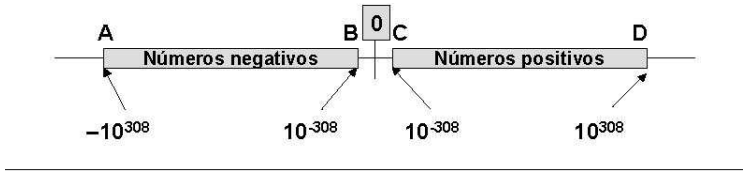

Figura 15

Otra característica importante a considerar en la representación en Coma Flotante es la precisión. Este concepto está relacionado con el número de cifras significativas de una representación. Con simple precisión, el número de cifras binarias significativas es 23, mientras que con doble precisión este número asciende hasta 52 dígitos binarios. Veamos su importancia con

Ejemplo: Se trata de efectuar la operación $(a - b) \cdot 2^{120}$, para los siguientes valores binarios de a y b :

a = 1,1100 0111 0001 1100 0111 0001 11

b = 1,1100 0111 0001 1100 0111 0001 01

Con 32 bits (23 cifras significativas, simple precisión) el resultado sería cero, pues se eliminarían las tres últimas cifras fraccionarias

Sin embargo, con 64 bits (52 cifras significativas, doble precisión) el resultado sería $2^{-25} \times 2^{120} = 2^{95} \approx 4 \times 10^{28}$.

Sobran los comentarios.

2.4.5.3.- Conversión a Coma Flotante

Aunque ya lo hemos visto con un ejemplo, establezcamos los pasos para pasar una cantidad expresada en decimal (o en cualquier otro sistema de numeración) a coma flotante. Lo primero que hay que hacer es expresar esta magnitud en binario en la forma normalizada, es decir, de la forma

$$\pm\ 1{,}\ldots \times 2^{exp}$$

donde

- ± será el signo (1 bit)
- 1,... será la mantisa (con 23 o 52 bits según la precisión)
- exp será la característica (con 8 o 11 bits, siempre en exceso a $2^{N-1} - 1$).

Ejemplo 1: Se trata de representar en Coma Flotante el número – 0,00072.

En binario tenemos que 0,00072 = 0, 0000 0000 0010 1111 0010 1111 1001 10
Normalizando como se ha explicado anteriormente:

$$1,\ 0111\ 1001\ 0111\ 1100\ 110 \times 2^{-11}$$

a) La mantisa será el valor fraccionario anterior (sin la coma):

0111 1001 0111 1100 110

(hemos calculado 19 bits suponiendo que para nuestro problema sea una precisión suficiente, pero, como la representación en Coma Flotante con simple precisión requiere 23 bits para la mantisa, deberán añadirse otros 4 bits, a cero).

b) El signo es negativo, es decir, 1

c) El exponente, característica, debe ser – 11 en exceso a 127, o sea: 01110100

La representación, con sus 32 bits, será:

1011 1010 0011 1100 1011 1110 0110 0000

que, en hexadecimal, sería BA3CBD60, bastante más cómoda que la expresión binaria anterior.

2.5.- REPRESENTACIONES DECIMALES

Con las representaciones anteriores se almacenan valores en forma binaria. Sin embargo, en la vida real la representación habitual es la decimal. Por ello el código BCD proporciona una excelente interfaz para los sistemas binarios. Ejemplos de estas interfaces son las entradas por teclado y las salidas digitales.

En las representaciones decimales cada dígito de un número decimal se representa, de alguna forma, por un conjunto de cuatro bits de manera que, sin realizar ningún cálculo aritmético se puede conocer el valor decimal correspondiente.

Las principales representaciones decimales son las siguientes:

- Códigos BCD (Binary Coded Decimal, o Decimal Codificado en Binario)

  BCD natural

  BCD Aiken

  BCD exceso tres
- Decimal Desempaquetado
- Decimal Empaquetado

Seguidamente analizaremos estas formas de representación, prescindiendo de BCD Aiken y BCD exceso tres, por su escasa utilización en la actualidad.

2.5.1.- CÓDIGOS BCD

Veremos únicamente la representación en el código BCD natural.

Para representar en binario los valores decimales 0 a 9, basta emplear cuatro bits: desde 0000 hasta 1001. En el código BCD natural, se asigna a cada dígito decimal la combinación de cuatro bits cuyo valor en binario coincide con el dígito correspondiente (Tabla 3). Así, en lugar del dígito 5, se representará el valor cinco en binario '1001' y en lugar del dígito 3 el valor binario '0011'. Siempre se emplean cuatro bits por cada dígito.

Ya sabemos que con 4 dígitos se pueden representar 16 números (desde 0000 hasta 1111), pero en el código BCD natural sólo se usan 10 de ellos. Las 6 combinaciones restantes no son válidas, y se corresponden a 1010,1011,1100,1101,1110,1111.

Decimal	BCD natural	Decimal	BCD natural
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Tabla 3

Si el número decimal a representar tiene más dígitos, simplemente se usan más grupos de cuatro bits: un grupo por cada dígito (Tabla 4).

Decimal	BCD natural	Binario
15	0001 0101	0000 1111
32	0011 0010	0010 0000
97	1001 0111	0110 0001

Tabla 4

Obsérvese en la figura cómo esta representación es diferente de cualquiera anterior. Así el valor quince, que en binario (con 8 bits) se representaría como 0000 1111, en BCD natural se representa como 0001 0101 (los cuatro primeros bits son la representación binaria del primer dígito '1', y los cuatro siguientes la del segundo dígito '5').

Es igualmente sencillo determinar el número decimal a partir del código BCD. Se comienza por el bit más a la derecha y se divide el código en grupos de 4 bits. Después se escribe el dígito decimal representado por cada grupo de 4 bits.

Ejemplo: Convertir a decimal los siguientes códigos BCD

a) 10000110
   1000 / 0110 → 86
b) 001101010001
   0011/0101/0001 → 351

2.5.1.1.- Suma en BCD

BCD es un código numérico y puede utilizarse en operaciones aritméticas. La suma es la más importante de estas operaciones, ya que las otras tres operaciones (resta, multiplicación y división) se pueden llevar a cabo utilizando la suma.

La suma de dos números expresados en BCD natural se efectúa de forma similar a la de dos números decimales, es decir, dígito a dígito. Pero al sumar en binario dos números cuyo valor varía entre 0 y 9, pueden presentarse tres situaciones diferentes:

- *La suma no es superior a 9*. Entonces el resultado obtenido es correcto y al efectuar la suma binaria no se ha producido acarreo. Es el primer caso del ejemplo de la figura: se suman los dígitos 5 y 3, resultado 8.
- *La suma es superior a 9 o se genera un acarreo en el grupo de 4 bits*. Entonces el resultado no es válido. Sin embargo, si a esta combinación se le suma el valor seis (0110) el resultado nos da un acarreo hacia el dígito superior, que se debe tener en consideración, y, además una combinación BCD válida (0010) y que se corresponde con la del resultado deseado.

En la (Figura 16) vemos dos ejemplos de suma en BCD.

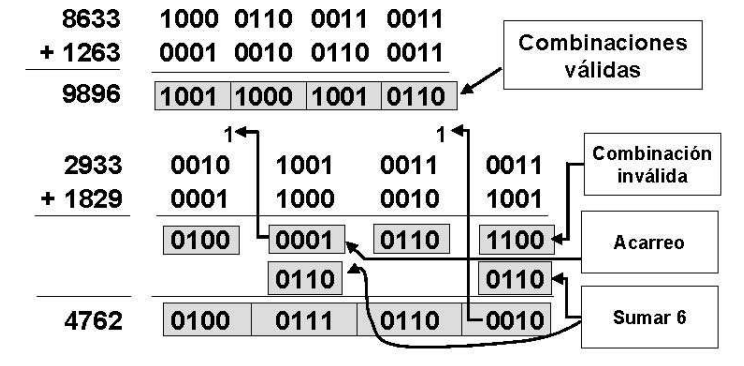

Figura 16

En el primer caso (8.633 + 1.263), las combinaciones obtenidas al sumar dígito a dígito son todas correctas y en ninguna de las sumas se produce acarreo. Por tanto, el resultado obtenido es correcto tal cual.

En el segundo caso (2.933 + 1.829), sin embargo nos encontramos con que

- La primera combinación obtenida (1100) no es válida. Debemos sumar 6, lo que nos produce una combinación correcta (0010) que nos da el primer dígito BCD de la suma (2) y además un acarreo que deberemos incluir en la siguiente suma.
- La segunda suma (incluido el acarreo anterior) da una combinación válida (0110) y no se produce acarreo. Tenemos ya el segundo dígito (6).
- La tercera suma da una combinación válida (0001) pero produce acarreo. Éste lo tomamos en consideración y, además, sumamos 6 a esta combinación. Obtenemos así el tercer dígito (0111). El tercer dígito será por tanto 7.
- La última suma (incluido el acarreo anterior) nos da una combinación válida (0100) y no hay acarreo, por lo que nos proporciona directamente el último dígito de la suma (4).

2.5.2.- REPRESENTACIÓN EN DECIMAL DESEMPAQUETADO

La representación en BCD natural no incluye signo. Este problema queda resuelto por las dos representaciones basadas en BCD natural que vamos a incluir ahora.

Tenemos, en primer lugar, la representación en Decimal Desempaquetado (Figura 17). En ella, cada dígito decimal ocupa un byte (es decir ocho bits). Los cuatro bits de la derecha de cada byte se corresponden directamente con su representación en BCD natural, mientras que los cuatro restantes tienen siempre el valor '1111' excepto en el primer byte. En éste tendrán el valor '1100' si el número a representar es positivo y el valor '1101' si el número es negativo.

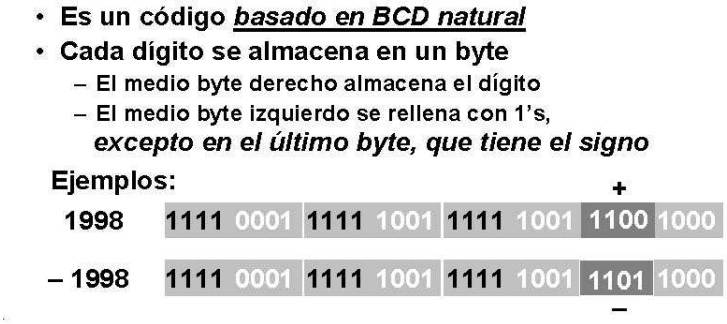

Figura 17

Ventaja:

- La principal ventaja de esta representación consiste en que emplea un byte para cada dígito decimal a representar, lo que facilita mucho el proceso de impresión o de presentación en pantalla de cantidades decimales.

Inconvenientes:

- Es una representación con muchos bits innecesarios. La mitad izquierda de cada byte es inútil (con excepción de la del primer byte que nos indica el signo), lo que supone una mayor cantidad de almacenamiento necesaria, con su correspondiente coste.
- Por otra parte, para efectuar operaciones aritméticas con estas cantidades hay que realizar unas operaciones previas de 'limpieza' de esta información innecesaria, lo que complica su realización.

2.5.3.- REPRESENTACIÓN DECIMAL EMPAQUETADO

La representación en Decimal Empaquetado, resuelve estos dos problemas.

- Permite trabajar directamente con cantidades en decimal (sin necesidad de pasarlas a binario) y
- No emplea más almacenamiento del necesario.

La solución es evidente: se eliminan los grupos de cuatro bits innecesarios (la parte izquierda de cada byte, excepto el primero), "empaquetando" (de ahí su nombre) dos dígitos decimales en cada byte. El signo, que evidentemente es necesario, se deja en el primer byte, aunque ocupando la mitad derecha (Figura 18).

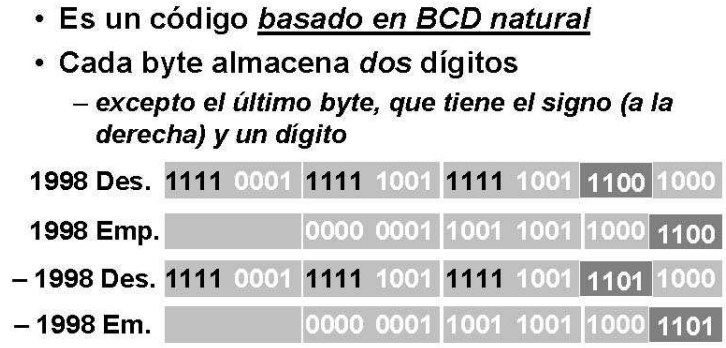

Figura 18

2.6.- SISTEMAS DE CODIFICACIÓN

Un ordenador es capaz de almacenar y tratar información de características muy diversas, desde información numérica hasta información audiovisual. En cualquier caso, el ordenador únicamente es capaz de manejar magnitudes binarias ('ceros' y 'unos'). Por tanto, se necesita convertir, de alguna forma, la información del mundo real a alguna forma que pueda ser manejada directamente por los ordenadores.

Este proceso, en general, (Figura 19) se conoce como Codificación de la Información.

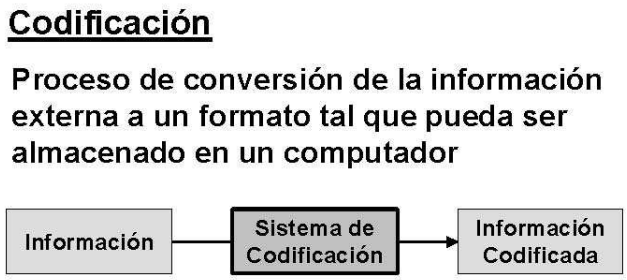

Figura 19

Existen muchos códigos especializados que se usan en los sistemas digitales. Algunos códigos son estrictamente numéricos, como BCD, y otros son alfanuméricos, es decir, se utilizan para representar números, letras, símbolos e instrucciones, como el código ASCII.

Los principales tipos de Sistemas de Codificación empleados son:

- ***Numéricos***: Codifican información numérica
  - MS, C-1, C-2, exceso 2n-1
  - Coma Flotante de Precisión Simple o Doble
  - BCD natural, BCD Aiken, BCD exceso-3
  - Decimal empaquetado o desempaquetado
- ***Alfanuméricos***: Codifican información alfanumérica
  - FIELDATA, ASCII, EBCDIC, UNICODE

Al tratar de códigos conviene precisar algunas definiciones:

- *Longitud*: es el número de bits empleados para representar un carácter. Consecuencia de la longitud es el número de caracteres de un código
- *Número de caracteres*: el valor máximo es $2^{longitud}$. En el caso, muy frecuente, de que la longitud sea 8, el número máximo de caracteres es 256.
- *Distancia* entre dos combinaciones binarias de un código: es el número de bits que deben

cambiarse en una combinación para obtener la otra; o, lo que es lo mismo, al número de bits diferentes que tienen esas dos combinaciones en particular. En la Figura 20 se incluyen algunos ejemplos con las distancias entre algunas combinaciones de un código.

- *Distancia de un código*: Aplicando la definición anterior se define como distancia de un código al menor valor de todas las distancias posibles entre combinaciones del código.

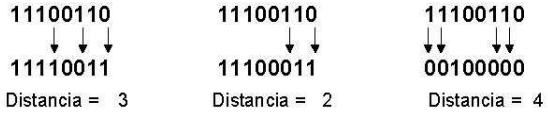

Figura 20

Ejemplo: Analicemos el código de la tabla 5:

0	00000	5	01010
1	00011	6	01100
2	00101	7	01111
3	00110	8	10001
4	01001	9	10010

Tabla 5

Se trata de un código en el cual se representan únicamente los diez dígitos decimales con las configuraciones que se indican. Este código emplea 5 bits para representar cada uno de los dígitos.

Si comparamos todas las combinaciones entre sí podemos encontrar únicamente distancias con un valor de 2 o de 4. Por tanto, la distancia del código es el menor de ellos; es decir, la distancia es 2.

2.6.1.- TÉCNICAS DE PROTECCIÓN CONTRA ERRORES

La información codificada está sujeta a posibles errores. En este contexto se entiende por error la modificación accidental de la información, sin la intervención directa o indirecta de ninguna persona. Por ejemplo, sería un error que el carácter 'A' almacenado en un disco se convirtiese en 'Q' "por las buenas", pero no lo sería que ocurriese lo mismo por una equivocación en un programa o por una acción malintencionada de alguna persona.

¿Se puede modificar la información "porque sí"? Hay una serie de factores externos y difícilmente evitables que, eventualmente, pueden producir un error. Desde radiaciones alfa que pueden alterar el contenido de la memoria o de algún dispositivo de almacenamiento hasta campos magnéticos que pueden alterar la información transmitida a algún dispositivo externo o a través de una línea de comunicaciones. El error se traduce en el cambio de algún bit de '0' a '1' o viceversa.

En principio, tal y como hemos considerado la representación de información hasta ahora, no es posible reconocer que se ha producido un error. Supongamos que tenemos codificado en C-2 (con ocho bits) el valor decimal '14'. Su representación sería '00001110'; si uno de los bits cambia accidentalmente de valor y el contenido de esos ocho bits pasa a ser, por ejemplo, '00011110', representará el valor '30', tan válido, desde el punto de vista de codificación, como el anterior. Estos errores son infrecuentes pero, de acuerdo con la Ley de Murphy, se producen y es preciso protegerse contra ellos. Para la protección se emplean determinadas técnicas en la codificación de la información:

- Existen códigos capaces de detectar que se ha producido un error, aunque no son capaces de saber dónde. Estos códigos pueden ser suficientes cuando el error se produce durante una transmisión: basta con repetirla, ya que la información se mantiene correcta en el punto de origen.
- Existen códigos con capacidad de corrección de errores. Si, por ejemplo, la letra 'A' se cambia accidentalmente por la 'Q', el código debe ser capaz de reconocer que se ha producido un error y además debe ser capaz de restituir la representación al valor correspondiente a la letra 'A'. Son códigos con capacidad de corrección de errores o, códigos autocorrectores.

2.6.2.- CÓDIGOS DE PARIDAD

Volviendo a nuestro código ejemplo de la Tabla 5, cuando la distancia de un código es 2 o más, no hay ninguna combinación válida que difiera de otra válida en un solo bit. Así, si en nuestro código se tuviese, digamos, el valor '4' ('01001') y se produjese un error que modificase uno de sus bits, produciendo, por ejemplo, la representación '01000', el código sería capaz de detectar que se había producido un error al reconocer que esa combinación '01000' no pertenece al conjunto de caracteres del código. Sin embargo, sin conocer el valor original, no podría conocer cuál de los bits era el que se había modificado indebidamente (en este caso podría ser cualquiera de ellos, excepto el primero).

Se puede afirmar que cualquier código cuya distancia sea 2 o superior, es capaz de detectar errores, aunque no que sea capaz también de corregirlos.

Si analizamos con cierto detalle nuestro código ejemplo, en la Figura 21 podemos observar que:

a) Los cuatro primeros bits se corresponden con el código BCD de cada uno de los dígitos.

b) El quinto bit se ha elegido de forma tal que el número total de bits que tienen el valor '1' en cada código, incluyendo este quinto bit, sea siempre par.

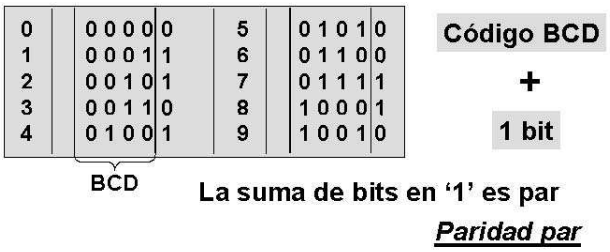

Figura 21

Los códigos con distancia dos, son los más simples con capacidad de detección de errores. Se denominan, en general, *códigos de paridad* y se obtienen añadiendo un bit al código básico (en nuestro ejemplo el BCD, en otros casos puede ser el ASCII, el Unicode o cualquier otro) de manera que el total de bits en '1' sea siempre par. Exactamente igual podría haberse elegido que el total de bits en '1' fuese siempre impar. En el primer caso se trata de paridad par, mientras que en el segundo estamos hablando de códigos de paridad impar.

Veamos un par de ejemplos (Figura 22). Supongamos que, utilizando nuestro código ejemplo (el BCD con paridad par), tenemos almacenado en un disco el valor '01001'.

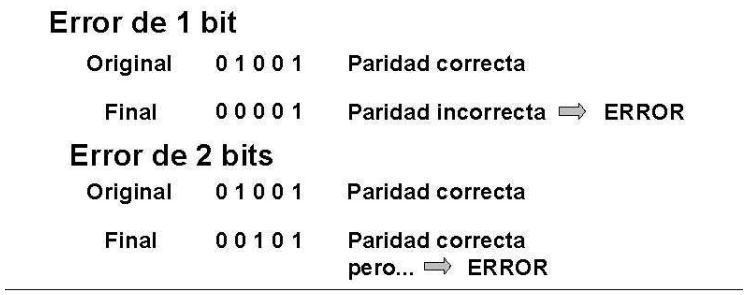

Figura 22

Si, al leer del disco recibiésemos el valor '00001', al comprobar su paridad se detectaría que el total de bits en '1' es impar, lo que nos indica que el código es inválido y que, por tanto se ha producido un error. Aunque no podemos conocer cuál debería ser el valor correcto. Supongamos ahora que recibimos el valor '00101'. Su paridad es correcta (el número total de bits en '1' es dos, o sea, par). El código no detecta, por tanto, ningún error. Pero, sin embargo, el valor se ha modificado durante la transmisión. ¿Qué significa esto?. Simplemente que nuestro código es capaz de detectar errores de un bit, pero no de dos bits. (También podría detectar los de tres bits).

En la práctica el no poder corregir errores de dos bits no es excesivamente grave ya que si la posibilidad de errores de un bit es muy reducida, la de errores de dos bits es prácticamente nula, aunque no podemos olvidar que un simple código de paridad 'se traga' los errores de dos bits. En cualquier caso

Un código de paridad no es capaz de corregir ningún tipo de error; únicamente puede detectarlos... y con limitaciones.

Por otra parte, no olvidemos que el introducir un código de paridad exige incrementar el número de bits necesarios para representar un dato. Si para BCD bastaría un código de cuatro bits, si queremos que pueda detectar errores, el código deberá tener, al menos, un bit más.

Existen dos tipos de escenarios:

- *Cuando se conserva en alguna parte el dato correcto y existe la posibilidad de recuperarlo*. En estos casos son muy útiles lo códigos con capacidad de detección de errores.
  Típicamente esto ocurre al transmitir datos, ya sea entre memoria y periféricos o a través de líneas de comunicación. Evidentemente la retransmisión del dato correcto supone una reducción en la velocidad de transmisión, que tiene mayor o menor importancia según la tasa de errores que se produzcan.

- *Cuando el dato correcto no puede recuperarse*. En estos casos el código detector tiene muy poca utilidad (aún le queda alguna, pues, al menos, se puede saber que es incorrecto). Lo que se requieren son códigos que tengan capacidad de corrección, es decir, que sean capaces no sólo de detectar que se ha producido un error (alteración de un bi), sino que sean capaces de recuperar directamente el valor correcto de ese bit. Los casos más habituales en los que se puede modificar accidentalmente un dato ocurren en la memoria o en los discos. Aunque la posibilidad de que se altere accidentalmente un bit es muy reducida, como la

capacidad de memorias y discos cada día es mayor, este fenómeno se produce con más frecuencia de la que podemos imaginar, e incluso no es demasiado infrecuente que se puedan alterar dos bits de un carácter simultáneamente.

En general, los códigos correctores de errores derivan del código básico de *Hamming* capaz

- de detectar y corregir cualquier error de un bit y
- de detectar, sin posibilidad de corrección, los errores de dos bits. Los códigos más avanzados son capaces de corregir errores de 2, o incluso de 3 bits.

Hay que notar que los sistemas más potentes suelen tener mayor capacidad de almacenamiento (memoria o disco) y mayor volumen de transferencias, por lo que el riesgo de que se produzcan errores es mayor (y de más trascendencia) por lo que suelen contar con códigos de corrección más potentes.

2.6.2.1.- Código de Hamming

En el código básico de Hamming se añaden varios bits (siempre más de uno) al código mínimo. Los bits imprescindibles para representar la información (por ejemplo, los 4 bits del código BCD) se denominan bits de información, mientras que los bits que se añaden para permitir la corrección se conocen como bits de paridad (Figura 23).

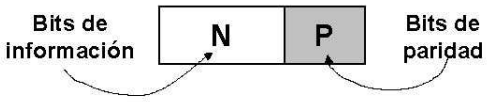

Figura 23

El número mínimo de bits de paridad necesarios para poder detectar y corregir errores de un bit viene dado por la expresión

$$2p \geq N + P + 1$$

En ella se ve que, cuando el número de bits de información es 4 (por ejemplo BCD), el número de bits adicionales debe ser 3, al menos; mientras que si el número de bits de información fuese 8 (como en el ASCII ampliado o Unicode 8) el número mínimo de bits de paridad debe ser 4, como mínimo ($24 \geq 8 + 4 + 1$). Esto significa un incremento del 50% en las necesidades de almacenamiento, con un coste adicional importante, lo que puede aconsejar no incluir este código en ciertos equipos sencillos. Códigos más potentes requieren aún más bits adicionales, con un mayor coste, lo que los restringe a equipos de grandes prestaciones, por lo que no será normal encontrarlos en un PC. Como veremos a continuación, el contenido de los bits de paridad nos indica si el dato es correcto o no, y en este último caso nos dirige al bit incorrecto.

Veamos primero cómo se forman los bits de paridad para el código básico de Hamming para BCD. Necesitaremos, como ya vimos, 4 bits de información y 3 bits de paridad: total 7 bits.

- Los bits de paridad ocupan las posiciones correspondientes a potencias sucesivas de dos: P1, P2, P4. (En caso de que el código no fuese el BCD y se requiriesen más bits de paridad, se continuaría con las posiciones 8, 16, etc.).
- Los bits de información ocupan las cuatro posiciones que dejan libres los bits de paridad: I3, I5, I6 e I7. Veamos un ejemplo (Figura 24): Supongamos que se desea codificar en BCD con código de Hamming el valor '9'. El BCD sería '1001' y estos bits ocuparían las posiciones I3, I5, I6 e I7, respectivamente. Falta conocer qué bits se deben colocar en las posiciones restantes.

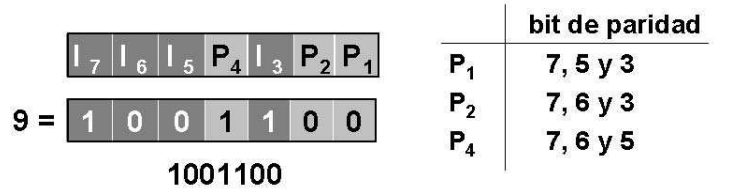

Figura 24

a) En la posición P1, se colocará el bit de paridad formado por los bits de información correspondientes a las posiciones I3, I5, e I7 (‘1’, ‘0’ y ‘1’). Para que esos tres bits más el correspondiente a la posición 1 den una paridad par, el contenido del bit de la posición P1 debe ser ‘0’ (total de bits en ‘1’ igual a 2, o sea par).

b) Análogamente, en la posición P2 se pondrá el bit de paridad formado por los bits de información I3, I6 e I7 (‘1’, ‘0’ y ‘1’). La posición 2 deberá tener, pues, un ‘0’.

c) Por último, la posición P4 tendrá el bit de paridad formado por los bits de información I5, I6 e I7 (‘1’, ‘0’ y ‘0’). La posición P4 deberá tener, pues, un ‘1’.

Nos hemos limitado a explicar el mecanismo de formación de los bits de paridad de este código, prescindiendo de cualquier demostración de tipo matemático sobre el mismo por escaparse del objetivo de la asignatura.

Veamos ahora cómo se produce la detección y posterior corrección de errores con este código. Supongamos que se tiene el valor ‘1011100’ según la Figura 25. Veamos si es correcto o no. Para ello se deben recalcular, según la regla vista anteriormente los bits de paridad (posiciones P1, P2, y P4) y comprobar si coinciden con los que aparecen:

a) Para la posición P1, el valor del bit de paridad debería ser ‘1’ (paridad par de los bits I3, I5, e I7). Pero vemos que, en realidad, el valor de este bit es ‘0’.

b) Para la posición P2, el valor del bit de paridad debería ser ‘0’ (paridad par de los bits I3, I6 e I7). El valor de este bit coincide.

c) Para la posición P4, el valor del bit de paridad debería ser ‘0’ (paridad par de los bits I5, I6 e I7). Pero vemos que el valor de este bit es ‘1’.

Lo anterior, en principio, no significa que estos bits sean o no correctos. Ahora bien, nos da la base para conocer lo que ocurre. Hecho este proceso, la regla para saber si hay un error y conocer dónde es la siguiente:

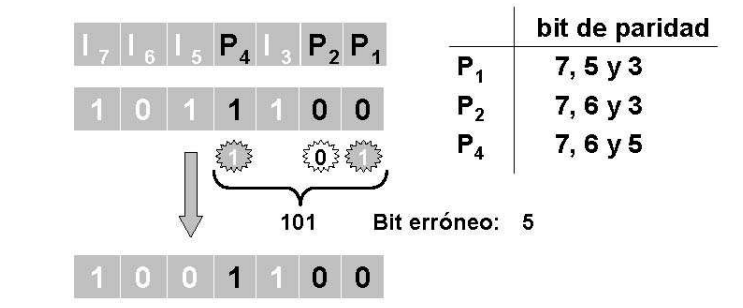

Figura 25

Si los tres bits de paridad calculados según las reglas anteriores coinciden con los valores que aparecen en las posiciones de bits de paridad, el dato es correcto, lo que no ocurre en nuestro caso. De lo contrario el dato no es correcto y para conocer cuál es el bit incorrecto se forma un número de control con tres dígitos en binario:

1. el primer dígito del número de control será un '0' cuando el bit de paridad calculado para la posición P4, coincida con el que realmente figure en el dato, y un '1' en caso contrario. En el ejemplo, no coincide, por tanto el primer dígito del número de control (por la izquierda) será un '1'.
2. para el segundo dígito, se repite el proceso con la posición P2. En el ejemplo coinciden el bit de paridad calculado con el que figura en el dato. Así el segundo dígito del número de control será un '0'. Así, los dos primeros dígitos del número de control serán '10'.
3. para el tercer y último dígito, se repite el proceso con la posición P1. En el ejemplo el bit de paridad calculado no coincide con el que figura en el dato. Así, el tercer dígito del número de control (por la izquierda) será un '1'. El valor binario de estos tres dígitos es, pues, '101'.

El valor decimal del número de control binario, es decir '5', nos indica que el bit incorrecto es el que ocupa la posición 5. Como este valor en nuestro dato es '1', y es incorrecto, el valor real y correcto debe ser '0'. En conjunto, el valor corregido, con todos sus bits, será '1001100'.

En la Figura 26 se presenta otro ejemplo, en el cual al calcular los bits de paridad que corresponden a las posiciones P1, P2, y P4, todos ellos resultan coincidentes con los que aparecen en el dato. Por tanto se generará un '0' para cada uno de los dígitos que se emplean para localizar el bit incorrecto, si lo hay.

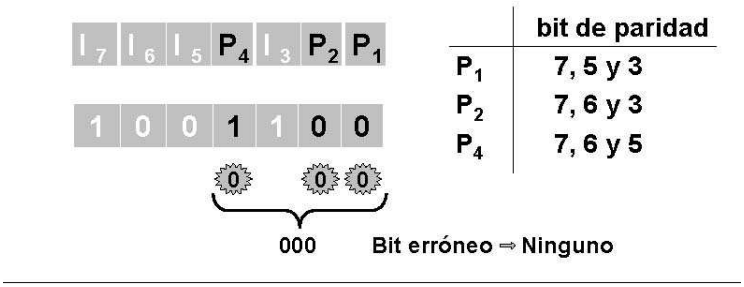

Figura 26

Cuando todos los dígitos de comprobación tienen el valor '0', acabamos de decir que el valor del código es correcto. Efectivamente '1001100' se corresponde con el valor '9' en código Hamming para BCD.

El ordenador deberá tener algún circuito capaz de realizar las operaciones que acabamos de indicar, circuito que veremos más adelante y que resulta ser realmente sencillo.

BIBLIOGRAFÍA RECOMENDADA:

• E. Alcalde y otros. Arquitectura de Ordenadores.- Mc Graw Hill. 1991 (Cap. 2)

• J.M. Angulo y otros. Sistemas Digitales y Tecnología de Computadores.- Paraninfo 2002

• M. Morris Mano y C.R. Kime. Fundamentos de Diseño Lógico y Computadoras.- Prentice Hall. 1998 (Cap. 1)

• W. Stallings. Organización y Arquitectura de Computadores.- Prentice Hall. 2000 (Cap. 8)