CAPÍTULO 4
ARITMÉTICA Y REPRESENTACIÓN DE LA INFORMACIÓN EN EL COMPUTADOR

Dos de los aspectos básicos que se presentan en el tratamiento de la información son cómo representarla (de lo cual dependerá sus posibilidades de manipulación) y cómo almacenarla físicamente. La primera se resuelve recurriendo a un código adecuado a las posibilidades internas del computador, que abordaremos en el presente capítulo. La segunda tiene que ver con los llamados soportes de información y es una cuestión que pertenece al ámbito del soporte físico del ordenador.

En la raíz de los desarrollos informáticos está el hecho de que todo dato puede ser representado por un conjunto de bits, circunstancia que permite a la ALU realizar un gran número de operaciones básicas utilizando su representación binaria. El paso a códigos binarios es una misión que el computador lleva a cabo automáticamente, por lo que el usuario puede despreocuparse de este proceso. Sin embargo es conveniente tener algunas ideas acerca de la forma como el computador codifica y opera internamente, cuestión indispensable para comprender determinados comportamientos de la máquina. Para ello empezaremos recordando algunos conceptos relativos al sistema de numeración binario y a las transformaciones entre éste y el sistema decimal.

4.1. SISTEMAS DE NUMERACIÓN EN INFORMÁTICA

Llamaremos sistema de numeración en base **b,** a la representación de números mediante un alfabeto compuesto por **b** símbolos o cifras. Así todo número se expresa por un conjunto de cifras, contribuyendo cada una de ellas con un valor que depende:

a) de la cifra en sí,
b) de la posición que ocupa dentro del número.

En el sistema de numeración decimal, se utiliza, **b** = 10 y el alfabeto está constituido por diez símbolos, denominados también cifras decimales:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} (3.1)

y, por ejemplo, el número decimal 278.5 puede obtenerse como suma de:

$$\begin{array}{r} 200 \\ 70 \\ 8 \\ 0.5 \\ \hline 278.5 \end{array}$$

es decir, se verifica que:

$$278.5 = 2\times10^{2} + 7\times10^{1} + 8\times10^{0} + 5\times10^{-1}$$

Cada posición, por tanto, tiene un peso específico (en el sistema decimal, cada posición además tiene un nombre):

	valor en el ejemplo	nombre
posición 0 peso b^0	8	**unidades**
posición 1 peso b^1	7	**decenas**
posición 2 peso b^2	2	**centenas**
posición -1 peso b^{-1}	5	**décimas**
....		

Generalizando, se tiene que la representación de un número en una base **b**:

$$N \equiv \ldots n_4\, n_3\, n_2\, n_1\, n_0\, .\, n_{-1}\, n_{-2}\, n_{-3} \ldots \quad (3.2)$$

es una forma abreviada de expresar su valor, que es:

$$N \equiv \ldots n_4 b^4 + n_3 b^3 + n_2 b^2 + n_1 b^1 + n_0 b^0 + n_{-1} b^{-1} \ldots \quad (3.3)$$

donde el punto separa los exponentes negativos de los positivos.

Nótese que el valor que tome **b** determina la longitud de la representación; así, por un lado, resulta más cómodo que los símbolos (cifras) del alfabeto sean los menos posibles, pero, por otra parte, cuanto menor es la base, mayor es el número de cifras que se necesitan para representar una cantidad dada. Como ejemplo veamos cual es el número decimal correspondiente de 175372, en base 8, (cuyo alfabeto es {0, 1, 2, 3, 4, 5, 6, 7} y recibe el nombre de sistema octal).

$175372)_8 = 1\times8^5 + 7\times8^4 + 5\times8^3 + 3\times8^2 + 7\times8^1 + 2\times8^0 =$
$= 6\times10^4 + 4\times10^3 + 2\times10^2 + 5\times10^1 + 0\times10^0 = 64250)_{10}$

4.1.1 DEFINICIÓN DEL SISTEMA BINARIO

En el sistema de numeración binario **b** es 2, y se necesita tan sólo un alfabeto de dos elementos para representar cualquier número: {0,1}. Los elementos de este alfabeto se denominan cifras binarias o bits. En la Tabla 4.1 se muestran los números enteros binarios que se pueden formar con 3 bits, que corresponden a los decimales de 0 a 7.

Tabla 4.1.- *Números binarios del 0 al 7*

binario	000	001	010	011	100	101	110	111
decimal	0	1	2	3	4	5	6	7

4.1.2 TRANSFORMACIONES ENTRE BASES BINARIA Y DECIMAL

Para *transformar un número binario a decimal:*
Basta tener en cuenta las expresiones (3.2) y (3.3), en las que $b = 2$.

Ejemplo 1:

Transformar a decimal los siguientes números binarios:

110100; 0.10100; 10100.001

$110100)_2 = (1\times2^5) + (1\times2^4) + (1\times2^2) = 2^5 + 2^4 + 2^2 = 32 + 16 +4 = 52)_{10}$
$0.10100)_2 = 2^{-1} + 2^{-3} = (1/2) + (1/8) = 0.625)_{10}$
$10100.001)_2 = 2^4 + 2^2 + 2^{-3} = 16+4+(1/8) = 20125)_{10}$

Observando el Ejemplo 1 se deduce que se puede transformar de binario a decimal sencillamente sumando los pesos de las posiciones en las que hay un 1, como se pone de manifiesto en el Ejemplo 2.

Ejemplo 2:

Transformar a decimal los números: $1001.001\ )_2$ y $1001101\ )_2$

$1001.001)_2$	1	0	0	1	0	0	1	$= 8 + 1 + 1/8 = 9125)_{10}$
pesos →	8	4	2	1	1/2	1/4	1/8	

$1001101)_2$	1	0	0	1	1	0	1	$= 64 + 8 + 4 + 1 = 77)_{10}$
pesos →	64	32	16	8	4	2	1	

Para *transformar un número decimal a binario*:

a) La parte entera del nuevo número (binario) se obtiene dividiendo la parte entera del número decimal por la base, 2, (sin obtener decimales en el cociente) de partida, y de los cocientes que sucesivamente se vayan obteniendo. Los residuos de estas divisiones y el último cociente (que serán siempre menores que la base, esto es, 1 o 0), en orden inverso al que han sido obtenidos, son las cifras binarias.

Ejemplo 3:

Pasar a binario el decimal 26

26 | 2
0   13 | 2
1   6 | 2
0   3 | 2
1   1

$26)_{10} = 11010)_2$

b) La parte fraccionaria del número binario se obtiene multiplicando por 2 sucesivamente la parte fraccionaria del número decimal de partida y las partes fraccionarias que se van obteniendo en los productos sucesivos. El número binario se forma con las partes enteras (que serán ceros o unos) de los productos obtenidos, como se hace en el siguiente ejemplo.

Ejemplo 4:

Para pasar a binario el decimal 26.1875 separamos la parte fraccionaria: 0.1875 y la parte entera: 26 (ya transformada en el Ejemplo 3).

$$\begin{array}{r} 0.1875 \\ \times 2 \\ \hline \mathbf{0}.3750 \end{array} \qquad \begin{array}{r} 0.3750 \\ \times 2 \\ \hline \mathbf{0}.7500 \end{array} \qquad \begin{array}{r} 0.7500 \\ \times 2 \\ \hline \mathbf{1}.5000 \end{array} \qquad \begin{array}{r} 0.5000 \\ \times 2 \\ \hline \mathbf{1}.0000 \end{array}$$

Por tanto, habiéndonos detenido cuando la parte decimal es nula, el número decimal 26.1875 en binario es:

$26.1875)_{10} = 11010.0011)_2$

NOTA: un número real no entero presentará siempre cifras después del punto decimal, pudiendo ser necesario un número finito o infinito de éstas, dependiendo de la base en que se represente; por ejemplo el número $1.6)_{10}$ representado en binario sería $1.100110011001...)_2$, requiriendo infinitas cifras para ser exacto, como por otra parte ocurre con muchos números representados en decimal.

4.1.3 CÓDIGOS INTERMEDIOS

Como acabamos de comprobar, el código binario produce números con muchas cifras, y para evitarlo utilizamos códigos intermedios que son bases mayores, que no se alejen de la binaria. Estos se fundamentan en la facilidad de transformar un número en base 2, a otra base que sea una potencia de 2 (2^2=4; 2^3=8; 2^4=16, etc.), y viceversa. Usualmente se utilizan como códigos intermedios los sistemas de numeración en base 8 (u octal) y en base 16 (o hexadecimal).

4.1.3.1 BASE OCTAL

Un número octal puede pasarse a binario aplicando los algoritmos ya vistos en 4.1.2. No obstante, al ser b=8=2^3, el proceso es más simple puesto que, como puede verse

$$n_5 2^5 + n_4 2^4 + n_3 2^3 + n_2 2^2 + n_1 2^1 + n_0 2^0 + n_{-1} 2^{-1} + n_{-2} 2^{-2} + n_{-3} 2^{-3} =$$
$$(n_5 2^2 + n_4 2^1 + n_3 2^0)\times 2^3 + (n_2 2^2 + n_1 2^1 + n_0 2^0)\times 2^0 + (n_{-1} 2^2 + n_{-2} 2^1 + n_{-3} 2^0)\times 2^{-3} =$$
$$(m_1)\times 8^1 + (m_0)\times 8^0 + (m_{-1})\times 8^{-1}$$

Cada 3 símbolos binarios (3 bits) se agrupan para formar una cifra de la representación en octal, por tanto en general puede hacerse la conversión fácilmente, de la forma siguiente:

Para transformar un número binario a octal se forman grupos de tres cifras binarias a partir del punto decimal hacia la izquierda y hacia la derecha (añadiendo ceros no significativos cuando sea necesario para completar grupos de 3). Posteriormente se efectúa directamente la conversión a octal de cada grupo individual de 3 cifras, y basta con memorizar la Tabla 4.1 para poder realizar rápidamente la conversión.

Así por ejemplo

$10001101100.1101)_2$ =	010	001	101	100	.	110	100	
	2	1	5	4	.	6	4	= $2154.64)_8$

De octal a binario se pasa sin más que convertir individualmente a binario (tres bits) cada cifra octal, manteniendo el orden del número original. Por ejemplo:

537.24 $)_8$ = 5 3 7 . 2 4
101 011 111 . 010 100 = 101011111.0101 $)_2$

Para *transformar un número de octal a decimal* se aplica la expresión (3.3) con *b*=8.

Para *transformar un número de decimal a octal* se procede de forma análoga a como se hizo para pasar de decimal a binario dividiendo o multiplicando por 8 en lugar de por 2.
Así se puede comprobar que 1367.25$)_8$ = 759.328125$)_{10}$ ó que 760.33$)_{10}$ =1370.2507...$)_8$

4.1.3.2 BASE HEXADECIMAL

Para representar un número en base hexadecimal (esto es, *b*=16) es necesario disponer de un alfabeto de 16 símbolos:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Tabla 4.2.- *Números binarios del 0 al 7*

hexadec.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
binario	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Al ser *b*=16=2^4, de modo similar al caso octal, cada símbolo hexadecimal se corresponde con 4 símbolos binarios (4 bits) y las *conversiones a binario* se realizan agrupando o expandiendo en grupos de 4 bits. Se pueden comprobar las transformaciones siguientes:

10111011111.1011011 $)_2$ = 0101 1101 1111 . 1011 0110
5 D F . B 6 = 5DF.B6$)_H$

1A7.C4 $)_H$ = 1 A 7 . C 4
0001 1010 0111 . 1100 0100 = 110100111.110001$)_2$

De la misma forma que manualmente es muy fácil convertir números de binario a octal, y viceversa, y de binario a hexadecimal, y viceversa, también resulta sencillo efectuar esta operación electrónicamente o por programa, por lo que a veces la computadora utiliza este tipo de notaciones intermedias como código interno o de entrada/salida, y también para visualizar el contenido de la memoria o de los registros.

Para *transformar un número de hexadecimal a decimal* se aplica la expresión (2.3) con *b*=16. Para pasar un número de decimal a hexadecimal se hace de forma análoga a los casos binario y octal: la parte entera se divide por 16, así como los cocientes enteros sucesivos, y la parte fraccionaria se multiplica por 16, así como las partes fraccionarias de los productos sucesivos.

Así se puede comprobar que $12A5.7C)_H = 4773.484375)_{10}$ ó que $16237.25)_{10} = 3F6D.4)_H$

4.2. OPERACIONES ARITMÉTICAS Y LÓGICAS

El procesamiento de la información incluye realizar una serie de operaciones con los datos; estas operaciones y las particularidades de las mismas en su realización por el computador son el objeto de los próximos apartados.

4.2.1 OPERACIONES ARITMÉTICAS CON NÚMEROS BINARIOS

Las operaciones aritméticas básicas (suma, resta, multiplicación y división) en sistema binario se realizan en forma análoga a la decimal aunque, por la sencillez de su sistema de representación, las tablas son realmente simples:

Tabla 4.3.- *Operaciones básicas en binario*

Suma aritmética	Resta aritmética	Producto aritmético
0 + 0 = 0	0 - 0 = 0	0 . 0 = 0
0 + 1 = 1	0 - 1 = 1 y debo 1 (*)	0 . 1 = 0
1 + 0 = 1	1 - 0 = 1	1 . 0 = 0
1 + 1 = 0 y llevo 1(*)	1 - 1 = 0	1 . 1 = 1

(*) *Llamado normalmente acarreo. En binario 1+1=10 (es 0 y me llevo 1), igual que en decimal 6+6=12 (es 2 y me llevo 1)*

Ejemplo 5:

Efectuar las siguientes operaciones aritméticas binarias:

```
 1110101      1101010       1010011    110.01    |10
+1110110          ×11           ×10    10        11.001
--------    ---------     ---------    
11101011      1101010       0000000    010
             1101010_      1010011_      10
 1101010    100111110      10100110      00010
-1010111                                    10
--------                                    00
 0010011
```

A partir del ejemplo anterior, se observa que multiplicar por $10)_2$ (es decir, por 2 en decimal) equivale a añadir un cero a la derecha, o desplazar el punto decimal a la derecha, siendo esto similar a multiplicar por $10)_{10}$ un número decimal. De la misma forma dividir por $10)_2 = 2)_{10}$ equivale a eliminar un cero a la derecha, o desplazar el punto decimal a la izquierda.

Por ejemplo:

$1010011)_2 \times 2 = 10100110)_2$ $\quad$ $1010100)_2 / 2 = 101010)_2$
$10101.01)_2 \times 2 = 101010.1)_2$ $\quad$ $110.01)_2 / 2 = 11.001)_2$
$1.101101)_2 \times 2^5 = 110110.1)_2$ $\quad$ $10101.101)_2 / 2^6 = 0.010101101)_2$

4.2.2 VALORES BOOLEANOS Y OPERACIONES LÓGICAS

Un dígito binario, además de representar una cifra en base dos, también puede interpretarse como un valor booleano o dato lógico (en honor a George Boole, citado en el capítulo 1), entendiendo como tal una cantidad que solamente puede tener dos estados (Verdadero/Falso, SI/NO, 1/0, etc.) y por tanto con capacidad para modelizar el comportamiento de un conmutador. Así, además de las operaciones aritméticas con valores binarios, se pueden llevar a cabo operaciones booleanas o lógicas en las que estos valores se consideran señales generadas por conmutadores. Las operaciones booleanas más importantes son:

OR lógico (también denominado unión, suma lógica (+) o función O),
AND lógico (también intersección, producto lógico (.) o función Y)
la complementación (¯) (o inversión, negación, o función NOT o NO).

Nótese que las dos primeras son operaciones de dos operandos o binarios mientras que la complementación es unaria. Estas operaciones se rigen según la Tabla 4.4.

Tabla 4.4.- *Operaciones lógicas*

OR
$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

AND
$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

NOT
$0 = \overline{1}$
$1 = \overline{0}$

Como puede observarse, el AND y OR lógicos se corresponden parcialmente con el producto y suma binarios, y lo más significativo, es la posibilidad de

implementar estas operaciones lógicas, y por tanto las aritméticas binarias, en forma de circuitos.

4.2.3 PUERTAS LÓGICAS

Necesitamos examinar la realización en hardware de algunas de las operaciones máquina básicas. Para poder hacer esto, vamos a diseñar y "construir" algunos de esos circuitos. A lo largo de este recorrido se irá poniendo de manifiesto la importancia de la lógica, en el proceso de diseño de componentes hardware.

Tenemos tendencia a considerar a los números y a la aritmética como las entidades básicas de la informática. Después de todo, los números y las operaciones aritméticas es lo primero que se estudia en los cursos elementales. Sin embargo, tanto la suma como el resto de las operaciones de la máquina se tienen en términos de la lógica. *Las computadoras trabajan con la lógica*. Sabemos que el sistema binario se utiliza para representar números (o, de hecho, también otro tipo de información) en la computadora. Sin embargo, utilizar la codificación binaria es equivalente a definir los números en términos de las entidades lógicas básicas **verdadero** y **falso**. Por ejemplo $5)_{10}$ es $101)_{2}$. Pero esto es lo mismo que decir **verdadero falso verdadero** si asimilamos 1=**verdadero** y 0=**falso.** Parece, pues, apropiado comenzar nuestro proyecto de construcción con las operaciones lógicas **and**, **not** y **or**.

Para ello, necesitamos entender la operación y conocer qué circuitos podemos construir. Las operaciones **and** y **or** son binarias: tiene dos entradas u operandos, y de ellos se obtiene una salida. Para poder determinar qué componentes pueden ser de utilidad en la construcción de estos circuitos **and** y **or**, conviene recordar que representamos la información como una secuencia de dos estados (verdadero/falso, 0/1). Por tanto, el elemento básico tiene que ser algún tipo de conmutador, es decir, algún dispositivo que pueda representar los dos estados, un simple interruptor[1] puede hacerlo. Trabajaremos ahora con electricidad para simular la lógica, por lo que utilizaremos hilos en lugar de variables lógicas, cada hilo tiene voltaje o no lo tiene (1/0).

En la Figura 4.1. pueden verse los circuitos buscados a los que denominaremos **puertas lógicas** (en general, dispositivos físicos que realizan operaciones lógicas).

---

[1] En la práctica un interruptor resulta demasiado lento, por lo que necesitamos algún tipo de conmutador que pueda cambiar de estado a velocidad electrónica. El transistor es este dispositivo; se trata de un conmutador que interrumpe o permite el paso de corriente eléctrica y utiliza la electricidad para realizar la conmutación. Este es el elemento activo de la CPU (más recientemente el circuito integrado; sin embargo, un circuito integrado no es más que una pequeña "tableta" de circuitería que incluye transitores y algunos otros componentes electrónicos básicos, como condensadores y resistencias).

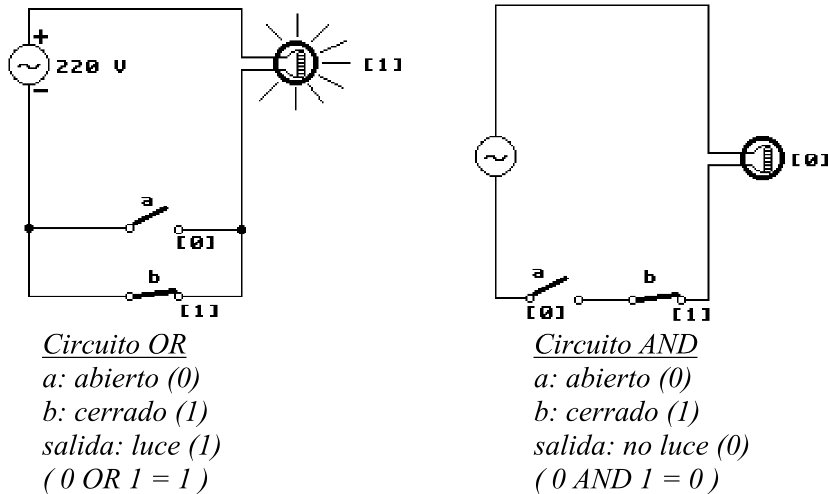

Fig. 4.1. *Circuitos eléctricos cuyo comportamiento puede describirse mediante las funciones OR y AND.*

Obsérvese que existen dos operandos de entrada y uno de salida. En el caso del circuito **and**, solo si los dos conmutadores están cerrados, (las entradas son **verdadero**) existirá voltaje a la salida, lo que se interpreta como **verdadero** (o 1). Análogamente, se reproduce con este circuito, el resto de filas de la tabla de verdad de la operación **and**.

Otra operación muy utilizada es el llamado or-exclusivo, simbolizada por **xor**, se definido mediante la ecuación

a **xor** b = (a **or** b) **and not** (a **and** b)

Informalmente significa "a o b, pero no ambos". Obsérvese que no es necesario construir un circuito diferente para ella, ya que un circuito que realice esta función puede conseguirse enlazando puertas elementales **not**, **and** y **or**, combinándolas como se hace en el lado derecho de la ecuación que la define.

4.2.4 ARITMÉTICA CON PUERTAS LÓGICAS

Las operaciones lógicas básicas, raramente se usan de forma aislada. Se puede imaginar las combinaciones de estas operaciones como conexiones de una puerta lógica a la entrada de otra de tal manera que al atravesar el circuito una corriente eléctrica, se van abriendo o cerrando hasta obtener la operación deseada. Algunos de los circuitos lógicos, incluso en los computadores más sencillos, son extremadamente complejos y no es objeto de este curso entrar en este tema, propio

de arquitectura de computadores, sin embargo es importante retener la idea de que los elementos lógicos pueden combinarse para realizar operaciones mas complejas y que la tabla de verdad de una combinación de ellas puede determinarse a partir de las tablas de las puertas elementales.

A modo de ejemplo, ahora que conocemos las puertas lógicas básicas, abordamos el diseño de una unidad que sea capaz de sumar dos enteros en binario. Por ejemplo, podríamos querer realizar la suma de la tabla 4.5, siguiendo el algoritmo de suma tradicional, sumando primero los números de la columna de más a la derecha. Para realizar este proceso, primero calculamos la suma de la primera columna, sumando los bits 1+1. Después de hacer eso, resulta evidente que se nos produce un arrastre. Por tanto, la segunda cosa que debemos hacer para conseguir sumar enteros es calcular el arrastre.

Tabla 4.5.- *Suma en binario*

001001
010101
011110

Esta primera suma nos confirma que la suma binaria debe considerar una doble salida: el resultado de aplicar la tabla de sumar y los posibles acarreos ("llevar" de una posición a otra). Así, las reglas para la suma de dos dígitos binarios se dan en la Tabla 4.6:

Tabla 4.6.- *Resultados de la suma en binario*

Sumandos	Suma	Acarreo
0 0	0	0
0 1	1	0
1 0	1	0
1 1	0	1

Puede notarse que la columna Suma puede obtenerse mediante una puerta XOR mientras que la del acarreo coincide con una puerta AND. En la Figura 4.2 se muestra un circuito lógico con estas puertas. Un circuito de este tipo se denomina semisumador.

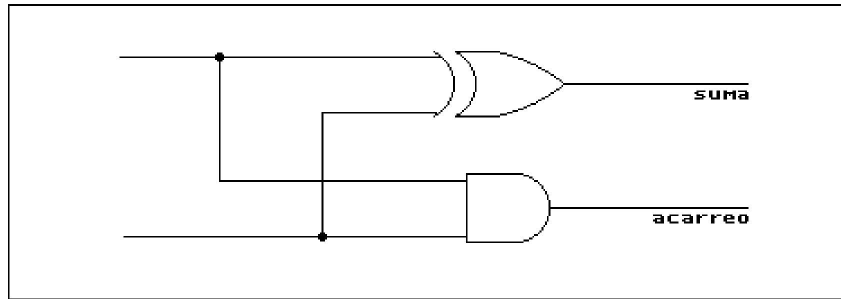

Fig. 4.2. *Esquema de un semisumador La puerta de arriba representa la puerta lógica XOR y la de abajo la puerta lógica AND*

Sin embargo, todavía no hemos terminado. Volviendo a la suma anterior, recordemos que debemos continuar con el proceso de suma, procediendo de derecha a izquierda a través de una serie de bits. El problema es que algunas de esas sumas producen un arrastre que debe incluirse en la siguiente suma de la izquierda. En la tabla 4.5., esto ocurre en la primera suma, es decir, la suma de los dígitos de más a la derecha, puesto que 1+1=0 mas un arrastre de 1. Este bit arrastrado debe considerarse en la suma siguiente. ¡Parece que sólo hemos realizado la mitad del trabajo!. Si queremos extender nuestro semisumador a un sumador, el problema que nos encontramos es que necesitamos poder sumar a la vez tres bits, en lugar de dos. Necesitamos una circuiteria con tres entradas (el arrastre y los dos operandos) y dos salidas (el arrastre de salida y el bit suma). Esto puede hacerse con dos semisumadores y una puerta **or**, como se muestra en la Figura 4.3.

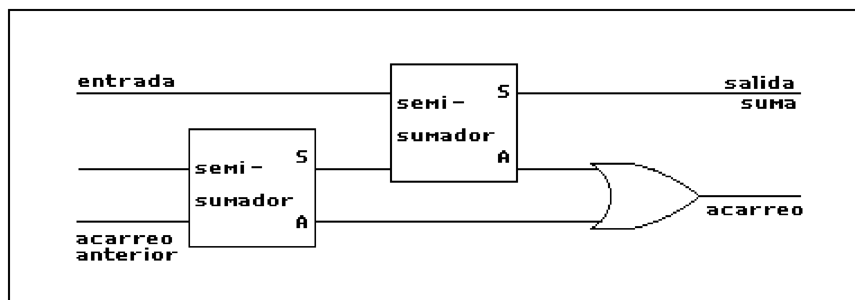

Fig. 4.3. *Esquema de un sumador completo, construido a partir de dos semisumadores. La puerta que proporciona el resultado del acarreo es una puerta lógica OR*

Es fácil ver cómo el sumador produce el bit suma. El primer semisumador produce el dígito resultante de la suma de los operandos y el segundo semisumador suma a este resultado el bit de arrastre proveniente del sumador anterior. Que el sumador produce el bit de arrastre correcto es un poco más difícil de ver. Básicamente, el

sumador debe producir un bit de arrastre 1, bien si la suma de los operando es mayor que 1, bien cuando el resultado que se produce al sumar el arrastre anterior es mayor que 1. En términos del diagrama de la Figura 4.3, significa que siempre que cualquier semisumador produzca un arrastre de 1, se debe producir una salida 1 del bit de arrastre. Este es exactamente el efecto que produce la puerta **or.** Esta explicación no constituye una demostración de que el sumador trabaje correctamente, ya que para demostrar que el circuito es correcto, necesitaríamos aplicarle cada uno de las ocho posibles combinaciones para los valores de los operandos y del bit de arrastre de entrada.

Nuestro objetivo de diseñar una unidad operativa para sumar enteros, no está aún alcanzado, puesto que nuestro sumador todavía solo suma correctamente una posición de un número binario. Sin embargo, puesto que sabemos que el proceso de suma es idéntico para todas las posiciones, es fácil construir la unidad operativa, pues basta encadenar juntos un número apropiado de sumadores. Por ejemplo, si nuestros números binarios tuvieran 4 bits, podemos construir un sumador de números de 4 bits utilizando cuatro sumadores en cascada. Este sumador puede verse en la Figura 4.4. En este diagrama, las tres entradas para cada bit se encuentran en la parte de arriba de la caja negra correspondiente, y las dos salidas se encuentran en la parte de abajo. Obsérvese que los sumadores se han conectado de manera que simulan la forma en que realizamos la suma binaria; el bit de arrastre de cada sumador se conecta al bit de entrada de arrastre del sumador de su izquierda. Este diseño se conoce como *sumador helicoidal*, debido a la forma en que se produce la propagación hacia la izquierda del bit de arrastre. De forma análoga, es posible construir sumadores para enteros con mayor número de bits.

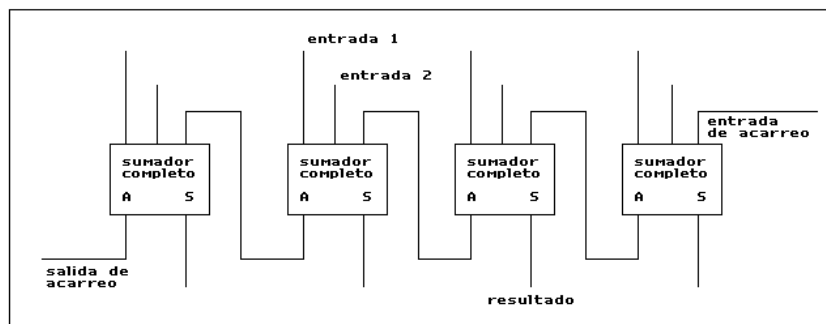

Fig. 4.4. *Esquema de un sumador en paralelo para cuatro bits.*

Nótese que existe un retraso asociado al paso de los datos a través de cada puerta lógica de un circuito. Aunque los circuitos de la figura operan en paralelo, puede ser necesario pasar un acarreo a través de todos los sumadores del circuito. Lo que supone un retraso considerable y la suma deja de ser “en paralelo”. Afortunadamente existen procedimientos para resolver estos problemas.

4.3. REPRESENTACIÓN DE INFORMACIÓN EN EL COMPUTADOR

Empecemos recordando que toda información que maneja y almacena el computador, (programas y datos) se introduce por los mismos dispositivos, utilizando un mismo alfabeto o conjunto de caracteres. Para evitar este tipo de problemas toda comunicación con el computador se realiza de acuerdo con los caracteres que admiten sus dispositivos de Entrada/Salida, y solo a partir de ellos, pueden los usuarios expresar cualquier dato o instrucción. Es decir, toda instrucción o dato se representará por un conjunto de caracteres tomados del alfabeto definido en el sistema a utilizar, lo que origina que algunos caracteres a veces no puedan reconocerse (la letra ñ por ejemplo). Los caracteres reconocidos por los dispositivos de entrada/salida suelen agruparse en cuatro categorías, dejando aparte los caracteres gráficos:

Caracteres alfabéticos: Son las letras mayúsculas y minúsculas del abecedario:

A, B, C,...,X,Y,Z, a,b,c,...,x,y,z

Caracteres numéricos: Están constituidos por las diez cifras decimales:

0,1,2,3,4,5,6,7,8,9

Caracteres de control: Representan órdenes de control, como el carácter indicador de fin de línea o el carácter indicador de sincronización de una transmisión o de que se emita un pitido en un terminal, etc. Muchos de los caracteres de control son generados e insertados por el propio computador y otros lo son por el usuario como por ejemplo, el carácter producido por la tecla de borrar. Estos caracteres suelen carecer de representación visual o impresa, pero desencadenan diferentes procesos cuando aparecen.

Caracteres especiales: Son los símbolos no incluidos en los grupos anteriores, entre otros, los siguientes:

) (, ; . : -_ ! * + = Ç ¿ ? ^ SP (espacio blanco, que separa dos palabras).

A los caracteres que no son de control, se les denomina **caracteres-texto** y al conjunto de los dos primeros tipos se le denomina conjunto de **caracteres alfanuméricos.**

4.3.1 LA CODIFICACIÓN EN INFORMÁTICA

Vamos a presentar ahora la forma como se codifican tanto los datos como las instrucciones, por lo que distinguiremos entre unos y otras aunque todas ellas estén formados por caracteres. A la hora de codificar información (sean datos o instrucciones), parece razonable aprovechar al máximo la memoria principal del computador; por ello, el número de bits de una palabra es normalmente un múltiplo entero del numero de bits con que se representa un carácter; desde el punto de vista de la CPU, los intercambios de información con la memoria se hacen por palabras y por caracteres y de esta forma para escribir o leer un dato o instrucción almacenado en la memoria principal, basta con proporcionar la dirección de la palabra correspondiente, para que se efectúe esta operación de escritura o lectura en paralelo (de forma simultánea todos los bits que la constituyen) gracias al bus.

4.3.1.1 CÓDIGOS DE ENTRADA/SALIDA

Los códigos de entrada/salida (E/S) o códigos externos son códigos que asocian a cada carácter (alfanumérico o especial) una determinada combinación de bits. En otras palabras, un código de E/S es una correspondencia entre el conjunto de todos los caracteres:

0,1,2,...9,A,B,...Y,Z.a,b,...y,z,*,”,%,...

y un conjunto de n-uplas binarias pertenecientes a: $\{0,1\}^{n,}$

El número de elementos, m, del primer conjunto depende del número de caracteres que el dispositivo o sistema informativo utilice y n dependerá a su vez de m, ya que con n bits se puede codificar $m=2^n$ símbolos o caracteres distintos, con lo que n debe ser el menor número entero que cumpla:

$$n \geq \log_2 (m) = 3.32 \log (m)$$

Tabla 4.7.- *Equivalencia entre valor decimal y carácter según el código ASCII*

Valor dec.	Carácter de control	Significado	Valor decim.	Carác-ter	Valor decim.	Carác-ter	Valor decim.	Carác-ter
0	NUL	Nulo	32		64	@	96	`
1	SOH	Comz. de cabecera	33	!	65	A	97	a
2	STX	Comienzo de texto	34	“	66	B	98	b
3	ETX	Fin de Texto	35	#	67	C	99	c
4	EOT	Fin de Transm	36	$	68	D	100	d
5	ENQ	Pregunta	37	%	69	E	101	e
6	ACK	Confirmación posit.	38	&	70	F	102	f
7	BEL	Pitido	39	‘	71	G	103	g
8	BS	Espacio Atrás	40	(	72	H	104	h
9	HT	Tabulador Horiz.	41	)	73	I	105	i
10	LF	Salto de Línea	42	*	74	J	106	j

11	VT	Tabulador Vertical	43	+	75	K	107	k
12	FF	Salto de Página	44	,	76	L	108	l
13	CR	Retorno de Carro	45	-	77	M	109	m
14	SO	Shift Out	46	.	78	N	110	n
15	SI	Shift In	47	/	79	O	111	o
16	DLE	Escape unión datos	48	0	80	P	112	p
17	DC1	Contrl dispositivo 1	49	1	81	Q	113	q
18	DC2	Contrl dispositivo 2	50	2	82	R	114	r
19	DC3	Contrl dispositivo 3	51	3	83	S	115	s
20	DC4	Contrl dispositivo 4	52	4	84	T	116	t
21	NAK	Confirm. negativa	53	5	85	U	117	u
22	SYN	Sincronización	54	6	86	V	118	v
23	ETB	Fin Bloque de Texto	55	7	87	W	119	w
24	CAN	Cancelar	56	8	88	X	120	x
25	EM	Fin del Medio	57	9	89	Y	121	y
26	SUB	Sustitución	58	:	90	Z	122	z
27	ESC	Escape	59	;	91	[	123	{
28	FS	Separad. de ficheros	60	<	92	\	124	|
29	GS	Separad. de grupos	61	=	93	]	125	}
30	RS	Separad de registros	62	>	94	^	126	~
31	US	Sep. de unidades	63	?	95	_	127	Delete

Nótese que con n=7, se puede representar hasta 128 caracteres, con lo que se puede codificar además de los caracteres habituales, determinados caracteres de control y gráficos. El más usado de estos códigos es el ASCII (American Standard Code for Information Interchange) representado en la Tabla 4.7, que incluye además un octavo bit para detectar posibles errores de transmisión o grabación.

4.3.1.2 DETECCIÓN DE ERRORES

La detección y corrección automáticas de los errores que se pueden producir, especialmente durante la transmisión de datos, es una cuestión que preocupa y como acabamos de indicar interviene incluso en la propia estructura del código. Una técnica muy valiosa, que sirve de gran ayuda para conseguir este objetivo, es el concepto de **código redundante** [2]. Un código redundante es aquel que contiene una cierta cantidad de información adicional embebida en la expresión codificada de los datos, que permite determinar, a partir del análisis de esta expresión, si los datos han sido codificados (o transmitidos) correctamente.

Los códigos redundantes más sencillos y comunes requieren la inclusión de un bit de paridad en el código de cada dato. Así el bit de paridad se pone a 0 ó a 1, según que el número de unos en el dato sea par, cuando se emplea paridad par, o impar, cuando se emplea paridad impar. Por ejemplo, empleando paridad par y el bit más

[2] Un ejemplo de código redundante es el N.I.F., en el que la letra depende de los dígitos del D.N.I. (en realidad es el resto de dividir el número del D.N.I. por 23, asignando a cada número del 0 al 22 una letra distinta). Con esta letra adicional es fácil detectar algunos errores de escritura (obviamente no todos).

significativo como bit de paridad, el número entero 0110101, al tener cuatro unos pone el bit más a la izquierda (llamado **bit más significativo**) a 0 para codificarse como el 00110101; mientras que el número 1001100, al tener tres unos, pone un 1 como bit más significativo para formar el 11001100 y tener un número par de unos.

De esta manera es fácil comprobar si la paridad de un dato es correcta (comprobación de paridad). Estas comprobaciones se realizan sobre todo cuando se han transmitido los datos para entrada o salida, hacia o desde un dispositivo de almacenamiento masivo o a través de una red de comunicaciones, donde los errores son más probables. Así por ejemplo, si recibimos el código 01001100 en un código con paridad par, podemos afirmar que ha habido un error al tener un número impar de unos.

4.3.1.3 ENCRIPTADO DE DATOS

Para muchas aplicaciones es esencial que los datos almacenados en disco o transmitidos mediante redes de comunicaciones tengan un cierto nivel de confidencialidad. Una técnica que proporciona un cierto grado de seguridad es el encriptado de los datos, que consiste en modificar los bits que los representan de modo que no sigan ningún código estándar y por consiguiente requieran una decodificación secreta antes de poder ser interpretados. En todos los casos en que los datos se encriptan de esta forma, la codificación y decodificación se realiza de manera automática por el hardware o el software del sistema.

Las técnicas de encriptado son muy variadas. Algunas utilizan algoritmos que codifican y decodifican los datos, empleando números aleatorios. Otras están basadas en los restos que se obtienen cuando se dividen los valores de los datos entre grandes números primos. La clave para códigos de esta naturaleza está constituida por un gran número entero (con más de 100 dígitos decimales) que es el producto de dos números primos, uno de los cuales se emplea para la división. Si el número clave es desconocido para el intruso el proceso de factorización es muy difícil, y toma tanto tiempo que, incluso utilizando los computadores más potentes, cuando la información se decodifica ya es inútil, por obsoleta.

4.3.2 REPRESENTACIÓN INTERNA DE DATOS

La procedencia de los datos a ser procesados, puede ser muy distinta y tener su origen en unidades muy diferentes del computador. Así, puede ser un carácter leído de un teclado, estar almacenado en un disco o ser un número que se encuentra en memoria principal. En cualquier situación, el dato tiene la misma

representación que dependerá de su naturaleza, que como sabemos puede ser: numérica (enteros y reales), lógica y alfanumérica.

Los códigos de E/S, presentan problemas cuando los aplicamos a los datos numéricos, con el objetivo de operar aritméticamente con ellos, ya que desafortunadamente su representación binaria, obtenida por un código de E/S, no es adecuada para estas operaciones. Por ejemplo basta con comparar la notación de 253 en código ASCII, con bit de paridad. 253= 10110010 00110101 00110011 y su representación binaria: $253=11111101)_2$ para darnos cuenta que el código de E/S, utiliza un número excesivo de bits respecto a la representación del número en sistema binario. La justificación de ello es que un código como el ASCII no está pensando para representar solamente datos numéricos.

De acuerdo con lo anterior, existe la necesidad de llevar a cabo una conversión de la representación simbólica de E/S de los datos a otra, que denominamos representación interna, que depende tanto de las características del computador, como del uso que el programador desee hacer con los datos. Con ello abrimos la posibilidad de que los datos puedan codificarse de forma distinta, según estemos en un proceso de E/S o consideremos los datos ya almacenados en una de las memorias del computador. Ello supone la aparición de un nuevo problema: un mismo conjunto de bits puedan interpretarse de forma distinta según sea la situación, así por ejemplo, el valor binario 1010011 puede significar: el valor de 83 si lo consideramos como un entero positivo, el carácter "S" en ASCII, y ello sin contar que podría ser una instrucción en vez de un dato, en cuyo caso, su interpretación dependería del lenguaje máquina del procesador.

Afortunadamente, esta aparente complejidad se simplifica, gracias a la introducción del concepto de tipo de dato que desarrollaremos al final de este capítulo, de forma que para todo dato manejado, obligatoriamente se ha tenido que definir previamente la naturaleza de su tipo. Esta definición implicará que su representación interna será una u otra, al tiempo que determina, como ya veremos, las operaciones que podemos realizar a partir de este dato concreto.

4.3.2.1 REPRESENTACIÓN DE DATOS ALFANUMÉRICOS

Los datos de tipo alfanumérico se representan internamente con el mismo valor binario que le corresponde al dato, según el código de E/S que se utilice. Por tanto en el caso alfanumérico, no hay diferencia entre su representación interna y la derivada de su código de E/S.

4.3.2.2 REPRESENTACIÓN INTERNA DE DATOS NUMÉRICOS

El ordenador distingue entre números de tipo entero (sin decimales) y de tipo real (con decimales) y utiliza representaciones internas distintas para ambos tipos. Sin embargo, cualquiera que sea la forma empleada para representar los números en un computador, hay siempre unos límites, superior e inferior, al valor que pueda ser representado. Estos límites dependen tanto de la forma como se representen como del número de bits asignados para cada número. Se emplea el término **overflow** (desbordamiento) cuando una operación produce un número tan grande que se sale de estos límites (el término **underflow** se usa en relación al límite inferior). Por ejemplo, si un computador empleara ocho bits para almacenar enteros positivos, el número más grande que podría almacenar sería 11111111 (binario) =255 (decimal). Cualquier intento de almacenar un número de más de ocho bits produciría necesariamente un desbordamiento.

Al objeto de reducir los errores de desbordamiento, y de reducir el espacio que se puede malgastar, (reservando un número excesivo de bits si éstos no son necesarios), existe una cierta flexibilidad en la representación. Así por ejemplo, la mayor parte de los computadores disponen de la posibilidad de representar los números en simple y doble precisión, de forma que los números en doble precisión ocupan más bits por lo que son más exactos. Afortunadamente los lenguajes de programación permiten a los programadores tener la posibilidad de elegir el tipo de representación más adecuada para cada número en función de como lo vayan a manejar a lo largo del programa.

4.3.2.2.1 Representación de datos de tipo entero

Esta representación se basa en el sistema de numeración binario, basado en la posición de cada dígito, en una palabra con un número de bits determinado. Estos datos numéricos, se suelen denominar de punto fijo, en contraposición al punto flotante, que utilizaremos para los datos numéricos reales. Existen dos alternativas de representación, según tengan o no signo, reservando en este último caso el bit más representativo para indicar una cantidad negativa. Los límites de valores enteros que podemos representar, sin caer en un overflow, dependen naturalmente de la longitudes de palabra que utilicemos. Es posible definir distintas clases de datos de tipo entero de forma que cada una de ellas tenga un rango numérico distinto. Así por ejemplo, con 16 bits se pueden representar los valores de 0 a 65536, y con 32 bits se va desde 0 a 4294967296.

Nótese que al operar con aritmética de números enteros, se puede producir un overflow cuando el resultado del cálculo excede el rango de números que pueden representarse. No hay forma de prevenir los overflows, todo lo que los sistemas

operativos pueden hacer es detectarlos cuando ocurren y avisar al usuario de esta circunstancia.

4.3.2.2.2 Representación en Complementos

Para representar un número entero negativo habitualmente recurrimos a añadir el signo. Sin embargo hay otras alternativas para facilitar su utilización. Normalmente se utiliza una representación diferente en binario, según que el número sea positivo o negativo, de esta forma, como se verá más adelante, las sumas y restas quedan reducidas a sumas, independientemente de los signos de los operandos. Este sistema de representación es de sumo interés, ya que al utilizarlo se reduce la complejidad de los circuitos de la ALU, pues no son necesarios circuitos específicos para restar.

Llamaremos **complemento a la base de un número**, *N*, al número que resulta de restar cada una de las cifras del número *N* a la base menos uno del sistema que se esté utilizando y posteriormente sumar uno a la diferencia obtenida. A partir de él, se puede dar la siguiente regla: Para *restar dos números se puede sumar al minuendo el complemento a la base del sustraendo despreciando, en su caso, el acarreo del resultado*.

Veamos, ejemplos para el caso decimal y binario:

a) En decimal (complemento a 10)

- Complemento a 10 del número 63 : 37.
- Complemento a 10 del número 16 : 84. En efecto:

```
   99                      99
 - 63                     -16
 ----                    ----
   36  +1 = 37             83  +1 = 84
```

- Supongamos que queremos efectuar las operaciones: 77-63 y 97-16. Las podemos realizar de dos formas, directamente: 77-63=14; 97-16=81 o utilizando el complemento del sustraendo y despreciando los acarreos finales:

```
   77        77              97      97
 - 63      + 37            - 16    + 84
          ------                  ------
          (1)14                   (1)81
```

b) En binario (llamada representación en complemento a 2):

- Complemento a 2 del número 1010 : 0110.
-

- Complemento a 2 del número 0011 : 1101. En efecto:

```
  1111                          1111
- 1010                        - 0011
------                        ------
  0101  +1= 0110                1100  +1 = 1101
```

- Supongamos que queremos efectuar las operaciones: 1100-1010 y 0111-0011. Las podemos realizar de dos formas, directamente o utilizando el complemento del sustraendo y despreciando los acarreos finales:

```
  1100      1100                0111      0111
- 1010    + 0110              - 0011    + 1101
          -------                       -------
          (1)0010                       (1)0100
```

Naturalmente los ceros a la izquierda del número no son necesarios y sólo se han colocado por motivos de claridad en la exposición.

En el ejemplo anterior, observamos que *para transformar un número binario, N, a complemento a 2 basta con cambiar los ceros por unos y los unos por ceros de N y sumar 1 al resultado*. Esto es siempre cierto y por tanto, no es necesario hacer las restas anteriores para calcular el complemento a 2, y se pueden restar dos números sólo efectuando sumas.

La representación interna más común para los números enteros con signo es la binaria de complemento a dos, donde si el bit más significativo es un 0 indica un valor positivo y si es un 1 indica un entero negativo cuyo valor absoluto es el complemento a 2 del número binario representado. Así por ejemplo, con 8 bits, tendríamos que el número 00110011 en binario representa el valor +51, mientras que el número 11001001 representa el entero negativo -55, ya que el bit más significativo a 1 indica que es negativo y debemos hacer el complemento a 2 de 11001001 que es 00110111 (55 en decimal) De este modo, los valores que se pueden representar van de -32768 a +32767 para el caso de 16 bits y de -2147483648 a +2147483647 usando 32 bits.

Quedaría la cuestión de como construir un circuito que calcule el complemento a dos, sabemos que el complemento a dos es igual al complemento a uno al que se le suma 1, basta con construir un circuito que calcule el complemento a uno. Para ello basta con poner una serie de puertas **not** en paralelo, con una señal complementaria que sirve para “disparar” la función. Es decir, cuando aparece un 1 en la línea de señal, el complemento de todas las líneas de entrada se envía al de las salidas. En caso contrario, los bits de entrada se envían a la salida sin sufrir alteración.

4.3.2.2.3 Representación de datos de tipo real

Al escribir un número en forma decimal, la cantidad de cifras decimales indica la precisión del mismo. Sabemos que muchos reales no pueden representarse mediante un número finito de cifras decimales. La precisión de una fracción decimal es una medida de cuánto se aproxima la representación al valor exacto del número. Por ejemplo, en base diez, el número racional 1/3 no puede representarse de forma exacta. Sin embargo, la representación con cuatro decimales 0.3333 es más precisa que la que sólo emplea dos: 0.33.

En las CPU las fracciones tienen necesariamente que almacenarse mediante un número finito de dígitos binarios. Como en el caso de las fracciones decimales, esto limita su precisión e implica que los cálculos que empleen estos números, raramente proporcionarán un resultado exacto. Para representar números reales se recurre a las técnicas basadas en el uso de **números en punto flotante**[3] que permiten expresar un rango mayor de los números que emplean un número dado de bits. Este método es similar al empleado para representar números en base diez en notación científica, denominada **método estándar,** que todo usuario de una calculadora de bolsillo conoce. Un número, en método estándar, consta de dos partes: la primera es una cantidad del intervalo [1, 10[, con posibles decimales, y la segunda, una potencia de diez. Por ejemplo:

$5.75 \times 10^4 = 57\ 500$
$6.7 \times 10^{-5} = 0.000\ 067$

Nótese que el orden de magnitud del número está determinado por la potencia de diez, y el número de cifras significativas, o precisión de número, está determinado por el número de cifras decimales en la primera parte.

Para la representación interna de números en punto flotante, se emplea el mismo principio, aunque en base dos. Así como en decimal, la primera parte es un numero comprendido entre 10^0 y 10^1 , en binario un número se expresa como el producto de dos partes: La primera es una fracción entre 1 (2^0) y 2 (2^1), llamada **mantisa**, y la segunda, una potencia de 2 llamada, **exponente**. Obsérvese, que al exigir que la mantisa esté comprendida entre 1 y 2, ello supone que escribiremos siempre los números de la forma:

$1.xxxxx * 2^{exponente}$

---

[3]En España, la parte decimal se separa por una coma. Sin embargo, en los países anglosajones, y por ello en la terminología informática, la parte decimal se señala con un punto. Es por ello, y para evitar confusiones, que en este libro empleamos el término punto flotante y la notación con punto en lugar de la coma.

donde el numero de dígitos a la izquierda del punto en la mantisa, dependerá de los bits que se hayan asignado a la representación interna de la misma)

Ejemplo 6:

Expresar en forma de mantisa y exponente, los siguientes números binarios:

$100100 = 1.001 * 2^5$
$111.101 = 1.11101 * 2^2$
$0.00111 = 1.11 * 2^{-3}$

Como consecuencia de este convenio notacional, siempre estará presente, en la representación del numero binario, el 1 de la parte entera de la mantisa, por lo que podemos prescindir de su representación interna, pues siempre se supone que existe; con ello nos ahorramos un bit en la representación binaria del mismo. Ello nos hace insistir en la diferencia que existe entre el concepto de mantisa y la representación interna que de ella hace el computador. Afortunadamente las unidades operativas de la ALU, están diseñadas teniendo en cuenta esta diferencia. Afinando mucho se observará que el número cero no puede ser representado internamente, ello se soluciona dando una codificación especial para el 0, que depende de cada ordenador y de cada lenguaje en particular.

Los siguientes ejemplos emplean cuatro bits para cada una de las partes, representadas, empleando el método de codificación conocido por **signo-y-magnitud**, donde un bit indica el signo y el resto de bits indican el valor absoluto, tanto de la mantisa como del exponente:

MANTISA				**EXPONENTE**				**valor**
Signo	1/2	1/4	1/8	Signo	4	2	1	
0	0	0	0	0	0	0	1	$= 1 \times 2^1 = 2$
0	1	0	1	0	1	0	0	$= (1 + 5/8) * 2^4 = 26$
0	1	1	0	1	0	1	0	$= (1 + 3/4) * 2^{-2} = 7/16$
1	1	1	1	0	1	1	0	$= -(1 + 7/8) * 2^6 = -120$

La forma de representar los números en punto flotante varían notablemente entre los distintos tipos de computadores. La mantisa se codifica en signo-y-magnitud o en complemento a dos. Además, el exponente se codifica a veces en **exponentes sesgados**. En este método, se resta un valor fijo al código, que representa al exponente, para determinar su valor real. Por ejemplo, si se emplean ocho bits para representar el exponente, los valores almacenados pueden variar entre 0 y 255. Sin embargo si se resta un valor prefijado, 128 (el sesgo), el rango de exponentes es de -128 a 127.

El número de bits que se asignan a cada parte de un número de punto flotante también varía entre los distintos computadores. El principio general es emplear el doble o el triple de bits para la mantisa que para el exponente. Veamos un ejemplo de codificación, empleando once bits para la mantisa y cinco para el exponente, signo-y-magnitud, para ambas partes del número.

MANTISA

Signo	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024
0	1	0	1	1	0	0	0	0	0	0

EXPONENTE

Signo	8	4	2	1
0	1	1	0	0

$=(1 + 1/2 + 1/8 + 1/16) \times 2^{12} = 11/16 \times 4096 = 6912$

El mismo número con un exponente sesgado (con desplazamiento 16) es:

MANTISA

Signo	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024
0	1	0	1	1	0	0	0	0	0	0

EXPONENTE

16	8	4	2	1
1	1	1	0	0

El exponente almacenado es $11100)_2$= 28, 12 cuando se le resta el sesgo 16.

En cualquier caso, una mayor cantidad de bits en el exponente permite representar un rango mayor de valores, mientras que una cantidad mayor de bits, dedicados a la mantisa permite representar el número con mayor precisión (con más cifras significativas).

Es por ello que se habla de números reales de **simple precisión** y de **doble precisión**, al utilizar el doble de bits para representarlos. El número de bits destinados a la mantisa y al exponente en cada caso dependerá de cada computador. Por ejemplo utilizando un total de 4 bytes en simple precisión se representan números en el rango de $\pm 3.4 \cdot 10^{38}$ con 7 cifras significativas, mientras que en doble precisión con 8 bytes el rango va de $\pm 1.7 \cdot 10^{308}$ con 15 cifras significativas.

4.3.2.2.4 Aritmética en punto flotante

Operar con números de punto flotante tiene una complejidad mayor que la que vimos para números enteros, por lo que conviene que repasemos la forma como se realizan las operaciones básicas, de lo cual sacaremos conclusiones importantes:

Suma de dos números

Dados dos números, x, y, expresados en punto flotante (recordando que sus mantisas, pertenecen a [1, 2[) se pueden escribir de la forma:

$x = d_1 * 2^n \qquad y = d_2 * 2^m$

la aritmética que utilizamos manualmente cumple que

si m = n $\qquad x+y = d_1\, 2^n + d_2\, 2^n = (d_1 + d_2)\, 2^n$

Por tanto, si los números tienen igual exponente, su suma tiene como mantisa la suma de las mantisas y como exponente el mismo. Téngase en cuenta que el resultado de $(d_1 + d_2)$ puede ser mayor que 2, en cuyo caso habrá que modificar debidamente su representación, esto es, si la suma de las mantisas “no cabe” en el lugar asignado, basta con despreciar los dígitos sobrantes menos significativos (“truncar”) e incrementar en uno el exponente de acuerdo con la siguiente propiedad:

$d * 2^n = (d/2) * 2^{n+1} = (d/2^2) * 2^{n+2}$

Esta misma propiedad, es la que nos permite conseguir que los dos números puedan expresarse con el mismo exponente, en caso de que no les coincidiera en la representación original. En el siguiente ejemplo, vamos a analizar esta operación.

Ejemplo 7:

Sumar los decimales 26.1875 y 12.8, expresándolos en punto flotante, en un ordenador que reserva 1 bit de signo y 5 bits para su mantisa y 1 bit de signo y 3 bits para el exponente:

En primer lugar hay que pasarlos a binario (Ver Ejemplo 4) y representarlos de acuerdo con las condiciones del ordenador que manejamos, efectuando los truncamientos que pudieran ser necesarios:

$26.1875 = 11010.0011)_2 \qquad = 1.10100)_2 * 2^4$

$12.8 = 1100.110011001..)_2 = 1.10011)_2 * 2^3$

	Mantisa						Exponente			
	Signo	1/2	1/4	1/8	1/16	1/32	Signo	4	2	1
26.1875	0	1	0	1	0	0	0	1	0	0
12.8	0	1	0	0	1	1	0	0	1	1

Una vez representados internamente, como el segundo número tiene el exponente menor, se desplaza la mantisa un lugar a la derecha (perdiendo el último 1). Esto es:

$1.10011)_2 * 2^3 = 0.110011)_2 * 2^4 =$ (truncando) $0.11001)_2 * 2^4$

Una vez igualados los exponentes, se suman las mantisas, obteniendo:

$[1.10100)_2 * 2^4] + [0.11001)_2 * 2^4] = [1.10100)_2 + 0.11001)_2] * 2^4 = 10.01101)_2 * 2^4$

Puesto que la mantisa desborda al intervalo [1, 2[, truncamos los bits que no pueden representarse en el espacio disponible de la mantisa, quedando:

$10.01101)_2 * 2^4 = 1.001101)_2 * 2^5 =$ (truncando) $1.00110)_2 * 2^5$

Al final de la operación tendremos:

Mantisa						Exponente			
Signo	1/2	1/4	1/8	1/16	1/32	Signo	4	2	1
0	0	0	1	1	0	0	1	0	1

Nótese que el numero arriba representado, equivale al decimal 38, cuando el resultado de la suma es: 26.1875 + 12.8 = 38.9875. La diferencia entre el resultado hallado y el correcto se debe a los dígitos perdidos en el proceso. La conclusión es importante: la suma de dos números en punto flotante origina inexactitudes.

Producto de dos números:

Dados dos números en punto flotante, esta operación consiste en multiplicar las mantisas de los números y sumar los exponentes, transformando el resultado en el caso de que la mantisa desborde el intervalo [1, 2[. El signo del producto viene dado por la utilización de una puerta xor sobre los bit de signo (0+0=0 ; 1+0=1 ; 0+1=1 ; 1+1=0)
Al igual, que en el caso de la suma, vamos a analizar esta operación a través de un ejemplo.

Ejemplo 8:

Multiplicar los decimales 26.1875 y 6.4 en el mismo ordenador, usado en el ejemplo anterior.

$26.1875 = 11010.0011)_2 = 1.10100)_2 * 2^4$
$6.4 = 110..110011001..)_2 = 1.10011)_2 * 2^2$

Obsérvese que al ser el segundo factor, la mitad del sumando, utilizado en el ejemplo anterior (6.4 = 12.8), la diferencia en binario entre ambos, se reduce a la posición del punto.

$[1.10100)_2 * 2^4] * [1.10011)_2 * 2^2] = [1.10100)_2 * 1.10011)_2] * 2^{4+2}$
$= 10.1001011100)_2 * 2^6 =$

Puesto que la mantisa debe pertenecer al intervalo [1,2[, en este ejemplo es necesario desplazar la mantisa una posición a la derecha, e incrementar el exponente en una unidad, para representar adecuadamente el producto; el resultado se trunca al numero de bits disponibles para la mantisa:

$= 10.1001011100)_2 * 2^6 = 1.01001011100)_2 * 2^7 =$ (truncando) $1.01001)_2 * 2^7$

Al final de la operación tendremos:

Mantisa						Exponente			
Signo	1/2	1/4	1/8	1/16	1/32	Signo	4	2	1
0	0	1	0	0	1	0	1	1	1

El numero arriba representado equivale al decimal (1 + 1/4 + 1/32) * 128 = 164, mientras que el resultado del producto es 26.1875 * 6.4 = 167.6, de donde se concluye que el producto de dos números en punto flotante también origina inexactitudes.

Comparando la aritmética en punto flotante con la aritmética en punto fijo obtenemos dos conclusiones:

- La aritmética en punto flotante produce errores de truncamiento
- La aritmética en punto flotante emplea más cálculo que la aritmética binaria para la misma operación matemática.

Por tanto, la utilización de números reales producirá muchos más errores de cálculo que el uso de números enteros, a la vez que empleará mayor tiempo de cómputo,

por lo que siempre que se pueda, es aconsejable utilizar números enteros. La utilización de coprocesadores matemáticos, reduce el tiempo de cómputo, aunque no los errores de truncamiento, que solo pueden ser reducidos incrementando el numero de bits destinados a la representación de la mantisa.

4.3.2.2.5 Limitaciones de la representación y la aritmética en el computador

Hemos visto que, siendo una herramienta muy potente y casi imprescindible para el cálculo, el uso del computador para representar valores numéricos y operar con ellos, presenta ciertas limitaciones que no deben olvidarse y que por su importancia pasamos a resumir:

rango: según la representación escogida, los valores que se pueden representar van desde un mínimo hasta un máximo. Esta limitación existe tanto con números enteros como con números reales.

precisión: con números reales, la limitación del número de dígitos significativos implica que no todos los valores puedan representarse exactamente. De hecho, dentro del rango de cada representación, habrá un número finito de representaciones exactas, y los números reales se representarán por la más próxima de aquellas.

overflow/underflow: estos errores se producen cuando al realizar una operación con dos números en una representación determinada (forzosamente la misma para los dos) el valor resultante cae fuera del rango de la misma y por tanto no puede ser codificado en dicha representación (o es codificado incorrectamente). El Sistema Operativo es el encargado de avisar de que se ha producido uno de estos errores.

truncamiento: estos errores (o inexactitudes) se producen en las operaciones con números reales donde la limitación en el número de bits que guarda la mantisa limita la precisión del resultado, en el cual se pierden las cifras menos significativas.

Acabemos esta sección, haciendo notar que tanto el software como el hardware juegan un papel importante en esta aritmética. Operaciones como la suma de números enteros se realizan directamente por hardware, mientras que las operaciones que se realizan en función de otras operaciones son supervisadas por el software de la máquina. De esta forma cada tipo de computador tiene su propia mezcla de hardware y software para la ejecución de las operaciones aritméticas. Este no es un tema menor, ya que está ligado al juego de instrucciones que incorpora la circuiteria del procesador. Como ya vimos, a la hora de seleccionar

una CPU hay que optar entre CISC, una máquina compleja que sea capaz de decodificar y ejecutar una mayor variedad de instrucciones o RISC un procesador más simple con un conjunto limitado de instrucciones más utilizadas. Ambas opciones tienen sus ventajas y sus inconvenientes y en la actualidad existen máquinas con una u otra arquitectura.

4.3.2.3 REPRESENTACIÓN INTERNA DE DATOS LÓGICOS

Los datos de tipo lógico representan un valor binario, es decir falso (0) o verdadero (1). Su representación varia de un computador a otro, sin embargo lo mas común es representar el 0 lógico haciendo 0 todos los bits de la palabra y el uno lógico con que al menos un bit de la palabra sea 1.

4.3.3 REPRESENTACIÓN INTERNA DE PROGRAMAS

Desde el punto de vista de la representación, vamos a distinguir entre un programa fuente y uno ejecutable. Un programa fuente representa las acciones que debe realizar el computador expresadas en un determinado lenguaje de alto nivel y como tal, es una información más, que se interpreta como un texto compuesto de caracteres. Estos son codificados al introducirlos al computador según el correspondiente código de E/S (usualmente el código ASCII) que más tarde el traductor se encargará de transformar en un ejecutable, que se representa de acuerdo con unos formatos previamente establecidos por el diseñador de la máquina.

Un programa ejecutable representa las acciones a tomar ya codificadas como conjunto de instrucciones máquina, y por tanto incomprensibles para el hombre, al estar codificados con los formatos preestablecidos para la máquina concreta. El programa adopta la forma de una sucesión de bits que engloba tanto el código binario de las instrucciones máquina como datos o direcciones utilizados en el programa.

Como sabemos, la forma que adopta habitualmente el código binario es la de bloques o campos. El primero de ellos es el código de operación y después de este pueden haber más campos, que indica la acción correspondiente a la instrucción. Estos campos dependerán de la operación que se trate. La representación de cada instrucción debe adecuar su formato a la función que realiza, así:

- Las instrucciones de transferencia deben indicar en los campos adicionales, la fuente y el destino de su transferencia, que tendrá que ver con alguno de los siguientes lugares:

- Un registro de la ALU, dando su número.
- La memoria principal, especificando el lugar (posición) de la memoria donde se encuentra el dato con el que hay que operar o que hay que transferir o donde hay que llevarlo.
- Un dispositivo de entrada o de salida, dando el número del dispositivo.

• En las instrucciones aritmético-lógicas, hay que indicar dónde se encuentran los operandos y dónde hay que depositar el resultado.

• Para codificar una instrucción que bifurque, habrá que indicar en ella la dirección a la que hay que saltar (esto es donde se encuentra la próxima instrucción a ejecutar) o donde se encuentra esa dirección de salto.

• Las instrucciones de control son mas sencillas de representar, pues dan una orden, que normalmente no depende de ningún parámetro (y puede no necesitar más campo que el código de operación).

Al contrario de lo que ocurre con la representación interna de datos, en el caso de la representación de instrucciones su relación con la longitud de palabra del computador es mucho menos rígida, debido a los distintos tipos de instrucciones que existen. Una instrucción máquina puede ocupar una o varias palabras de memoria y en algunos computadores, con palabra de varios bytes, se pueden empaquetar mas de una instrucción en una misma palabra. Además, la relación que existe entre el lenguaje máquina y el hardware hace que la longitud y composición de las instrucciones varíen considerablemente de un procesador a otro, aunque siempre manteniendo las características comunes que acabamos de describir.

4.4. EL CONCEPTO DE TIPO DE DATO

A lo largo del capítulo hemos establecido tanto cómo se representan internamente en el computador los datos enteros, reales, lógicos y caracteres, como la forma de operar con ellos. Más concretamente, hemos desarrollado distintos procedimientos de suma y producto, según sean los datos de punto fijo o flotante. Por otro lado, hemos visto que las variables son una forma de representar, a alto nivel, las posiciones de memoria donde se guardan sus valores. Llegados a este punto se hace evidente la necesidad de utilizar algún mecanismo para que auxilie al ordenador en dos tareas: 1) Que al consultar el contenido de una variable o alterar su valor, pueda saber cómo interpretar el valor contenido en la/s celda/s de memoria correspondientes a esta variable. 2) Que dada una expresión aritmética, exista un mecanismo para que pueda interpretar qué tipo de operación le está permitido llevar a cabo con los operandos que le proporciona, en cada momento, la ejecución del programa (p.e. la suma entera o la suma en punto flotante).

El mecanismo para resolver ambas cuestiones se basa en la introducción del concepto de **tipo de dato**; entenderemos como tal, tanto la capacidad de interpretación de un patrón de bits que representan datos, como las operaciones que pueden ser llevadas a cabo por estos datos. Afortunadamente durante la parte declarativa de un programa, existe la posibilidad de definir por parte del programador la naturaleza de los datos que va a utilizar (p.e. real o entero) y, en ciertos casos, la representación interna de cada uno de ellos (p.e. real de simple o doble precisión), lo cual determina automáticamente las operaciones permitidas entre ellos y la forma como se realizan éstas. Por tanto, los lenguajes de programación deben facilitar este proceso de especificación, de forma que antes de utilizar una variable, ésta deba *declararse* como perteneciente a un determinado tipo. De esta forma, el computador no tendrá ninguna ambigüedad en la representación interna de los valores de la variable ni en las operaciones a realizar con ellos. Veamos el siguiente ejemplo:

caso a	caso b	caso c	caso d
tipo x: entero	tipo x: real	tipo x: caracter	tipo x: entero
tipo y: entero	tipo y: real	tipo y: caracter	tipo y: entero
tipo z: entero	tipo z: real	tipo z: caracter	tipo z: caracter
z←x*y	z←x*y	z←x*y	z←x*y

En el caso a), los valores de x, y, z serán representados internamente utilizando la representación en punto fijo, y el producto se realizará con aritmética binaria. En el caso b), los valores de las variables se representarán utilizando mantisa y exponente, y el producto se realizará con aritmética de punto flotante. En el caso c), la expresión entre las variables no tiene sentido, puesto que no podemos multiplicar dos caracteres. En el caso d), la expresión z←x*y tampoco tiene sentido, puesto que aunque es posible multiplicar los valores de x e y, su resultado es entero y no puede asignarse a una variable de tipo caracter.

Así, la declaración de un tipo de dato en un programa, permite:

- Una forma inequívoca de representar, interpretar y tratar la información.

- Ahorrar memoria (eligiendo el tipo que, pudiendo representar los valores de una variable, ocupa menos memoria)

- Efectuar las operaciones de la forma más eficiente (recuérdese que aunque las operaciones parecen las mismas, la aritmética en punto flotante es más costosa que la binaria).

- Evitar errores de truncamiento, utilizando la aritmética en punto flotante sólo en los casos en que sea imprescindible.

- Detectar errores en las expresiones, sin conocer cual es su valor concreto en un momento dado (casos c y d en el ejemplo anterior).

En algunos casos, existe una cierta flexibilidad y es posible efectuar operaciones que involucren variables que no sean del mismo tipo. Consideremos la siguiente situación:

Precio y Tasa: tipo entero
Total: tipo real
Total ← Precio + Tasa

en este caso el compilador utiliza la suma entera y su resultado lo recodificará en un formato de punto flotante, antes de asignarlo a Total. Esta conversión implícita entre tipos se llama coerción y depende de cada lenguaje el que avise de su existencia.

Como veremos, los lenguajes de programación incorporan las definiciones de tipos de datos, sin que el programador tenga que preocuparse por su representación interna, por las operaciones permitidas o por la forma de llevarlas a cabo, ya que todo ello queda *encapsulado* dentro de la instrucción tipo, facilitando en gran medida la tarea de programación a alto nivel.

Veamos como se declaran las variables en los lenguajes que estamos considerando:

FORTRAN	**BASIC**
tipo lista de variables *tipo* es INTEGER REAL DOUBLE PRECISION LOGICAL COMPLEX CHARACTER * n (n, longitud de la cadena)	**DEFINT** - enteras **DEFINT** - simple precisión **DEFDBL** - doble precisión **DEFSTR** - cadena **DEF** *tipo rango letras, rango letras...* *tipo*: INT, SNG, DBL, STR

PASCAL	C
var lista_de_nombres1: tipo1;	`tipo1 lista_de_nombres1;` `tipo2 lista_de_nombres2;`

lista_de_nombres2: tipo2;

(Existe la posibilidad de definir tipos por el programador).

Es de destacar que, para obtener la mayor eficiencia en la representación de los datos y su manejo, suelen suministrarse muchos tipos posibles. Por ejemplo, el lenguaje Pascal tiene 5 tipos de enteros y 5 tipos de reales. Por poner un ejemplo más concreto, un determinado compilador de lenguaje C con palabras de 16 bits permite los siguientes tipos de datos simples numéricos

char	1 byte	complemento a 2	enteros de -128 a 127
unsigned char	1 byte	sin signo	enteros de 0 a 255
int	2 bytes	complemento a 2	enteros de -32 768 a 32 767
unsigned int	2 bytes	sin signo	enteros de 0 a 65 535
long	4 bytes	complemento a 2	enteros de -2 147 483 648 a 2 147 483 647
unsigned long	4 bytes	sin signo	enteros de 0 a 4 294 967 295
float	4 bytes	simple precisión	reales (7 dígitos) en $\pm 3.4 \cdot 10^{38}$
double	8 bytes	doble precisión	reales (15 dígitos) en $\pm 1.7 \cdot 10^{308}$
long double	10 bytes	doble precisión	reales (19 dígitos) en $\pm 1.2\ 10^{4932}$

Los vistos a lo largo de este capítulo son los llamados **datos de tipo simple:** enteros, lógicos, reales y alfanuméricos (algunos lenguajes como el C tienen un tipo simple adicional, el tipo puntero, que es un valor numérico entero sin signo para guardar una dirección de memoria - se dice que 'apunta' a una posición de la memoria). A partir de la agrupación de datos de tipo simple se pueden construir **tipos compuestos**, dotados de una cierta estructura. Sin embargo desde el punto de representación interna, el computador lo tratará siguiendo los principios que hemos descrito hasta aquí.

4.1. SISTEMAS DE NUMERACIÓN EN INFORMÁTICA 137

4.1.1 DEFINICIÓN DEL SISTEMA BINARIO..139
4.1.2 TRANSFORMACIONES ENTRE BASES BINARIA Y DECIMAL.........139
4.1.3 CÓDIGOS INTERMEDIOS 141

4.2. OPERACIONES ARITMÉTICAS Y LÓGICAS 143

4.2.1 OPERACIONES ARITMÉTICAS CON NÚMEROS BINARIOS 143
4.2.2 VALORES BOOLEANOS Y OPERACIONES LÓGICAS........................144
4.2.3 PUERTAS LÓGICAS...145
4.2.4 ARITMÉTICA CON PUERTAS LÓGICAS...146

4.3. REPRESENTACIÓN DE INFORMACIÓN EN EL COMPUTADOR ..150

4.3.1 LA CODIFICACIÓN EN INFORMÁTICA 150
4.3.2 REPRESENTACIÓN INTERNA DE DATOS...153
4.3.3 REPRESENTACIÓN INTERNA DE PROGRAMAS 165

4.4. EL CONCEPTO DE TIPO DE DATO..166