Arquitectura y Organización de Computadoras

Unidad 2: La información en una computadora: Tamaños principales: bit, byte, palabra, doble palabra. Aritmética del procesador. Representaciones numéricas de punto fijo. Operaciones. Representación y aritmética de números en punto flotante. Codificación de la información numérica y alfabética. Otros sistemas de codificación numéricos y alfanuméricos. Códigos redundantes: Concepto, tipos.

La información en una computadora

Las personas estamos acostumbradas a contar objetos y a realizar operaciones utilizando el sistema decimal. Este es uno de los muchos métodos de representación de la información que existe. Sin embargo, las computadoras no lo pueden manejar porque están formadas por dispositivos digitales, es decir, todos sus circuitos internos trabajan en forma binaria (encendido-apagado, si-no, 0-1). Esto implica que tengan que emplear otro sistema para gestionar la información: el sistema binario (sistema numérico con base 2, por lo que sólo utiliza dos símbolos: el 0 y el 1). Estos unos y ceros no son más que impulsos eléctricos con un determinado voltaje (por ejemplo: 3,3 voltios para el 1 y 0 voltios para el 0).

BIT

Teniendo en cuenta que las computadoras manejan un lenguaje binario, analizaremos los DÍGITOS BINARIOS como la unidad más elemental de información para la comunicación de datos. Se denomina BIT (contracción de BINARY DIGIT) al dígito binario que toma el valor 0 ó 1. Representa la información correspondiente a la ocurrencia de un suceso de entre dos posibilidades distintas: prendido o apagado, abierto o cerrado.
Un aspecto fundamental en la codificación binaria es determinar la cantidad de BITS necesarios para representar la información, de manera que podamos identificar una entre varias posibles.
Como un bit puede ser 1 o 0, podremos utilizarlo para seleccionar una información entre dos. Con dos bits, una entre cuatro; tres bits una entre ocho; etc. Las posibilidades aumentan como potencias de dos:

Un bit --------> $2^1 = 2$ posibilidades
Dos bits --------> $2^2 = 4$ posibilidades
Tres bits --------> $2^3 = 8$ posibilidades

Si quisiéramos conocer cuantos bits necesitamos para una de ocho situaciones utilizamos logaritmos en base 2 → $\log_2 8 = 3$.

En general el número de bits (B) que necesitaremos para poder codificar una determinada cantidad de información (N) estará determinado por:

$$B = \text{Log}_2 N$$

Como ejemplo, para poder representar en forma binaria los 26 caracteres de nuestro alfabeto necesitaríamos:

$$B = \text{Log}_2 26 = 4{,}7 \text{ bits} \Longrightarrow B = 5 \text{ bits}$$

BYTE

A pesar de que el ordenador maneja internamente la información en binario, no acepta entradas ni muestra salidas de esa forma. Para que la información de E/S sea comprensible y fácil de manejar, se utiliza un conjunto de símbolos, denominados caracteres. Se establece una correspondencia entre la representación interna binaria y el caracter externo, al que se le asigna una combinación única de bits que lo diferencia del resto. Casi todos los ordenadores utilizan internamente una agrupación de 8 bits, denominada BYTE, para representar cada carácter. Habitualmente byte se utiliza como sinónimo de caracter.

Palabra

Normalmente hay acuerdo acerca del significado de los términos “bit” y “byte”, pero no así sobre el concepto de “palabra”, el que depende de la arquitectura particular de cada procesador. La palabra o Word es un conjunto de uno o más bytes que la CPU trata como unidad. Es la unidad de información procesada por la UAL.
Los tamaños de palabra típicos son de 16, 32, 64 y 128 bits, siendo el de 32 bits el más común. Intel considera el tamaño de la palabra como de 16 bits y la doble palabra como de 32 bits.
Recordemos brevemente cómo se realiza el procesamiento de la información: las instrucciones y los datos pasan a través de buses a la memoria (MP). Dichas instrucciones son ejecutadas, una por vez, para lo cual primero cada una, por un bus, llega al registro de instrucciones (RI) de la UCP, donde permanece mientras se ejecuta, para que la UC interprete qué operación debe ordenar. Luego, el dato a operar por dicha instrucción llega desde la memoria a un registro acumulador (AX) del procesador, antes de ser operado en

la ALU, a fin de obtener un resultado. Este puede sustituir en el registro AX al dato ya operado, y luego pasar a la memoria si una instrucción así lo ordena.
El procesamiento de los datos que están en la MP será más rápido si en cada acceso a ella, el número de bits que la CPU puede tomar o enviar en paralelo es lo mayor posible. Es como llevar a la boca alimentos con una cuchara: cuando más grande sea ésta, más rápido se ingerirá la comida (a igualdad de cucharadas por minuto). Pero del mismo modo que existe un límite físico respecto de la cantidad máxima de alimento que se puede ingerir, en cada modelo de UCP se tiene una cantidad máxima de bits que ésta puede tomar por vez de la MP para procesar, que en general es múltiplo de 8.
La denominada "palabra" de un procesador es un indicador del número máximo de bits que puede tener un dato a ser operado por la UAL del mismo. Expresa también el número de bits de dato que maneja simultáneamente la UCP.
Si la palabra de un procesador es 32 bits, implica que la UAL puede sumar o restar dos números de 32 bits, y que el resultado que genere tendrá también 32 bits.
El tamaño o longitud de palabra determina además el ancho de los buses internos de la CPU por donde se transmiten las palabras, y el de los registros donde una palabra puede almacenarse. Es decir, son compatibles todos los elementos internos de la CPU.

Representación interna de la información

Sabemos que la información debe estar representada por patrones de "1" y "0". Sin embargo, existen muchas maneras de hacerlo, es decir, muchas maneras de **codificar los datos**. Algunos de los ejemplos más utilizados son: la codificación de números de punto fijo (con o sin signo), la de los números reales (conocida como Coma Flotante), y la de los caracteres requeridos para la impresión de texto.

Representación numérica de Punto Fijo

En esta representación la coma está ubicada siempre en el mismo lugar. Al representar con este formato, no se almacena la coma. Sólo se supone que está en un lugar determinado.

Respecto a un método de representación se deben hacer las siguientes consideraciones: capacidad de representación, rango y precisión.

Rango y Precisión en sistemas restringidos a n bits

El rango y la precisión son conceptos importantes en la arquitectura de computadoras, debido a que ambos son elementos finitos en la implementación de la arquitectura, mientras que son infinitos en el mundo real, por lo que el usuario debe tener en claro las limitaciones que surgen al tratar de representar información externa con formato interno.

El **rango** está dado por el número mínimo representable y el número máximo representable. En un sistema binario sin signo (BSS) el rango es $[0 \ldots (2^n - 1)]$, mientras que en un sistema binario con signo (BCS) el rango es $[(-2^{n-1} + 1) \ldots (2^{n-1} - 1)]$

La **precisión** es la mínima diferencia entre un número representable y el siguiente.

La **capacidad de representación** es la cantidad de números que se pueden representar. Está dada por b^n., donde b es la base del sistema en cuestión.

Ej. 1: si utilizamos 3 dígitos, con la coma entre el primero y el segundo, en el sistema decimal tendríamos un rango de 0,00 hasta 9,99 [0,00 ... 9,99]. La precisión de la representación es de 0,01, y el error es la mitad de la diferencia entre dos números consecutivos, como 4,01 y 4,02 es decir 0,01/2= 0,005. Implica que cualquier número de este rango puede representarse en este formato con una aproximación de hasta 0,005 de su valor real o preciso.

Ej. 2: en un sistema de representación binario sin signo, con cinco dígitos, el número máximo representable es 31 (b^n-1). El rango es [0, 31], es decir [00000, 11111] y la capacidad de representación 2^5 números, es decir, 32 números.

Sistemas de numeración posicionales

La base o raíz de un sistema de numeración define el rango de valores posibles que pueden adoptar sus dígitos.

La expresión general que permite determinar el valor de un número en un sistema de numeración de base *k* y en formato de punto fijo es la siguiente:

$$Value = \sum_{i=-m}^{n-1} b_i \cdot k^i$$

El valor del dígito que ocupa la posición *i* está representado por b_i. Existen en este caso *n* dígitos a la izquierda de la coma fraccionaria y *m* dígitos a su derecha. Esta forma de representación de un número, en la que cada posición tiene asignado un determinado valor, se denomina **sistema de numeración posicional**.

Supongamos la expresión $541{,}25_{(10}$, con n=3, m=2, k=10

$$\mathbf{541{,}25_{(10} = 5 \times 10^2 + 4 \times 10^1 + 1 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}}$$
$$\mathbf{= \quad 500 \; + \; 40 \quad + \quad 1 \quad + (2/10) \; + (5/100) \; = \; (541{,}25)_{10}}$$

Si en forma similar se considera el número binario $1010{,}01_{(2}$, en el que n=4, m=2 y k=2

$$\mathbf{1010{,}01_{(2} = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}}$$
$$\mathbf{= \quad 8 \quad + \quad 0 \quad + \quad 2 \quad + \quad 0 \quad + \quad 0{,}25 \quad = 10{,}25_{(10}}$$

En estos sistemas de numeración posicionales se define el bit con el mayor peso asociado como **bit más significativo** (MSB, *most significant bit*), y el bit de menor peso como **bit menos significativo** (LSB, *least significant bit*).

Conversión entre sistemas

De binario a decimal

El procedimiento visto anteriormente permite convertir un número expresado en un sistema de numeración de base cualquiera al sistema de numeración decimal, mediante la utilización de una representación polinómica. Se trata de multiplicar cada dígito por el peso asignado a su posición y luego sumar los valores para obtener el número convertido.

De decimal a binario

La forma más sencilla de convertir números que contengan tanto parte entera como fraccionaria, consiste en operar cada una de sus partes por separado.
Ej.: 23,375 a binario ➔ 23,375 = 23 + 0,375

Parte entera – Método de los restos

La forma polinómica general para la representación de un número entero binario es:

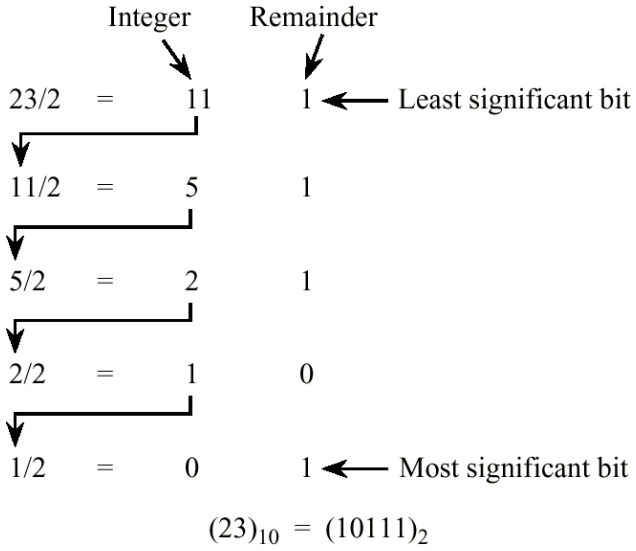
$b_i \times 2^i + b_{i-1} \times 2^{i-1} + ... + b_1 \times 2^1 + b_0 \times 2^0$

Si se divide el número entero por 2, se obtendrá:

$b_i \times 2^{i-1} + b_{i-1} \times 2^{i-2} + ... + b_1 \times 2^0$

con un resto de b_0. Como resultado de dividir por 2 el entero original se obtiene el valor del primer coeficiente binario b_0. Si se repite el proceso, se obtiene el segundo coeficiente b_1. Este procedimiento forma la base del **método de los restos**. Continúa hasta obtener 0 como cociente. Los restos obtenidos se unen en el orden indicado por la figura, teniendo en cuenta el MSB y LSB.

En general, todo número entero expresado en el sistema decimal puede convertirse a cualquier otro sistema dividiéndolo reiteradamente por la base del sistema de numeración al que se lo quiere convertir.



$(23)_{10} = (10111)_2$

Parte fraccionaria - Método de las multiplicaciones

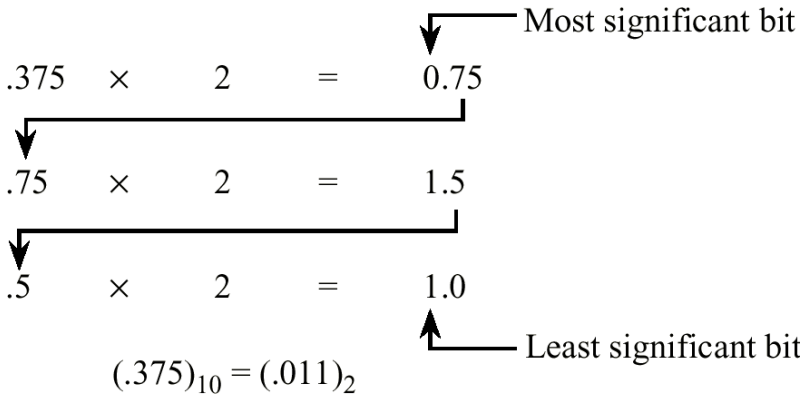
La conversión puede resolverse multiplicando sucesivamente la fracción por 2.
La forma polinómica general para la representación de una fracción binaria es:

$b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + ...$

Si se multiplica dicha expresión por 2, se obtiene

$b_{-1} + b_{-2} \times 2^{-1} + b_{-3} \times 2^{-2} + ...$

Se determina así el coeficiente b_{-1}. Reiterando el proceso sobre la fracción remanente, se obtendrán los sucesivos b_i. Este proceso continúa hasta obtener una fracción nula, o hasta que se haya alcanzado el límite de precisión requerida.



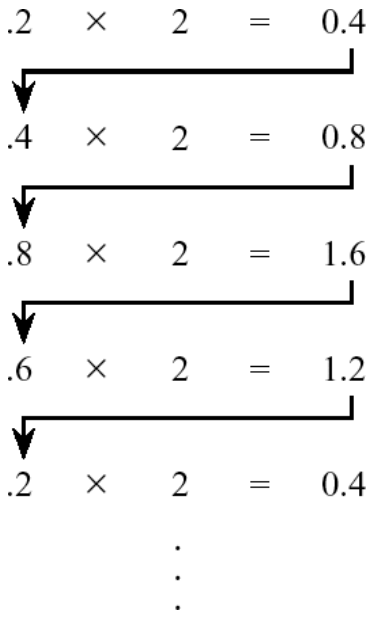
$(.375)_{10} = (.011)_2$

Por último, se combinan las partes entera y fraccionaria para obtener el resultado final:

$23{,}75 = 10111{,}011_{(2}$

Este proceso, denominado **método de las multiplicaciones**, puede ser utilizado para convertir números fraccionarios entre distintos sistemas de numeración, multiplicando por la base correspondiente. El multiplicador coincide con la base del sistema numérico de destino.

Si bien el método funciona con todos los sistemas de numeración, el proceso puede llegar a producir pérdidas de precisión. No todas las fracciones representadas en el sistema de numeración decimal pueden tener como equivalente un número racional exacto en el sistema binario. Por ej. el proceso de conversión del número decimal 0,2 (ver figura), llega a un punto en que se repite en forma periódica



Representación de números en los sistemas binario, octal y hexadecimal

Si bien los números binarios reflejan la realidad de la representación interna de los números en la computadora, tienen como desventaja el hecho de requerir mayor cantidad de dígitos para representar un número que cualquier otro sistema de numeración posicional. Además, es más probable que se cometan errores al escribir binarios debido a la gran cantidad de ceros y unos que se deben utilizar. Por estas razones es una práctica común la conversión a los sistemas octal (base 8) o hexadecimal (base 16), sistemas estrechamente vinculados al binario. Esta relación está dada por el hecho de ser estas bases potencias de 2 (la menor de todas ellas). Por otra parte, la conversión entre estos sistemas es trivial, y mucho más sencilla que convertir desde o hacia el sistema decimal.
Para el sistema hexadecimal se requieren seis dígitos más que en el decimal. Se utilizan letras, de A a F.

Binario (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)
000	0	0	0
001	1	1	1
010	2	2	2
011	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Al comparar las columnas de la tabla, se observa la necesidad de 3 bits para representar en binario cada uno de los dígitos del sistema octal, y cuatro para el sistema hexadecimal. En general, se requieren *k* bits para representar en binario un dígito del sistema de numeración de base 2^k, siendo *k* un número entero.
Para convertir un número del binario al octal, se divide el número original en grupos de 3 bits cada uno, empezando a partir de la coma decimal, completando el grupo más significativo con ceros, si fuera necesario. Luego, cada trío de bits se convierte en forma individual al sistema octal. Para conversiones desde el binario al hexadecimal, se utilizan grupos de 4 bits.

$11010{,}01_2 = (011_2)(010_2),(010_2) = 32{,}2_8$

$1101101{,}011_2 = (0110_2)(1101_2),(0110_2) = 6D{,}6_{16}$

Representación de Binarios con signo (BCS)

Existen cuatro convenciones distintas de uso habitual en la representación con signo

1. Módulo y signo (MS)
El bit que está situado más a la izquierda representa el signo, y su valor será de 0 para el + y de 1 para el -. El resto de los bits (n-1) representan el módulo o valor absoluto del número.
El rango de representación es para n dígitos de: $[-2^{n-1}+1 \ldots 2^{n-1}-1]$
Para el caso de 8 bits el rango es [-127 ... 127]

Por ej.: disponemos de 8 bits y queremos representar los números 10 y -10.

Número 10	0	0001010	Número -10	1	0001010
	+	**módulo**		**-**	**módulo**

En esta convención existen dos modos de representar el cero, una negativa y una positiva: 0000 0000 y 1000 0000

2. Complemento a 1 (C-1)
Este sistema de representación utiliza el bit de más a la izquierda para el signo, correspondiendo el 0 para el + y el 1 para el -. Para los números positivos el resto de los bits (n-1) representan el módulo del número. El negativo de un número positivo se obtiene complementando todos sus dígitos (cambiando ceros por unos y viceversa) incluido el bit de signo.
El rango de representación es de: $[-2^{n-1}+1 \ldots 2^{n-1}-1]$
Para el caso de 8 bits el rango es [-127 ... 127]
Este sistema, a semejanza del MS, posee la desventaja de tener dos representaciones del 0, en este caso, 0000 0000 y 1111 1111.

Número 10	0	0001010	Número -10	1	1110101
	+	**módulo**		**-**	**módulo**

3. Complemento a 2 (C-2)
Este sistema de representación utiliza el bit de más a la izquierda para el signo, correspondiendo el 0 para el + y el 1 para el -. Para los números positivos el resto de los bits (n-1) representan el módulo del número. El negativo de un número positivo se obtiene complementan el número positivo en todos sus bits (cambiando ceros por unos y viceversa) incluido el bit de signo, para luego sumarle 1, despreciando el ultimo acarreo si existe.
Ejemplo: **Número 10** 0 0001010
Número -10.....1 1110101
+1 1
11110110

El rango de representación es: $[-2^{n-1} \ldots 2^{n-1}-1]$
Para el caso de 8 bits el rango es [-128 ... 127]
Este método tiene la ventaja de poseer una sola representación para el 0.
Es la convención más utilizada en las computadoras.

4. Representación excedida, o Exceso a 2 elevado a n-1

Este método no utiliza ningún bit para el signo, con lo cual todos los bits representan un modulo o valor que corresponde al número representado más el exceso, que para n bits viene dado por 2 elevado a n-1.
Por ej., para 8 bits el exceso es de $2^7 = 128$. El número 10 vendrá representado por 10 + 128 = 138. Para el caso de -10 tendremos -10 + 128 = 118.

Número 10 10001010
Número -10.....01110110

En este caso, el 0 tiene una única representación, que para 8 bits corresponde a:
Número 0 (0 + 128) → 10000000

El rango de representación es: $[-2^{n-1} ... 2^{n-1} -1]$
Para el caso de 8 bits el rango es [-128 ... 127]

En este método, el valor del exceso tiene el formato del número negativo más grande, lo que produce como efecto que los números aparezcan ordenados numéricamente si se los mira en una representación binaria no signada. Así, el número negativo más grande es $-128_{10} = 0000\ 0000_2$ y el más grande positivo es $127_{10} = 1111\ 1111_2$. Esta representación simplifica las comparaciones entre números, dado que las representaciones binarias de los números negativos tienen valores numéricamente menores que las representaciones de los números positivos. Esto se hace importante cuando se representan los exponentes de los números en coma flotante, donde se requiere comparar los exponentes de dos cantidades para igualarlos, en caso de ser necesario, para sumar o restar.

Aritmética del computador

La suma binaria se realiza siguiendo las mismas reglas que en el sistema decimal. Cuando la suma de los dígitos excede los símbolos numéricos disponibles de la notación, se acarrea un 1 a la posición de dígito inmediatamente superior. Por lo tanto, en el sistema decimal, 3+5=8, pero 9+1=0 con acarreo de un 1 (es decir 10).
En el sistema binario hay solamente dos símbolos, 0 y 1. Por lo tanto, al sumar 1+1 en la notación binaria se excede el límite de la cuenta (ya que no hay otro símbolo disponible) y, en consecuencia, el resultado es 0 con acarreo de un 1 a la posición de dígito inmediatamente superior.
En la computadora se implementa la operación de suma mediante circuitos lógicos digitales combinatorios.
Dos números binarios A y B se suman de derecha a izquierda, generando un bit de suma y uno de arrastre en cada posición binaria.

Suma y resta en complemento a dos

Se analizará la resta implícitamente en la suma, como resultado del principio aritmético según el cual:
a – b = a + (-b), por lo tanto, se puede realizar la resta por medio del número complementado de este método. Como consecuencia, se ahorra en la estructura circuital de la unidad de cálculo, dado que se evita la necesidad de un elemento restador por hardware.
Lo que sí debe hacerse al sumar en C-2 es modificar la interpretación de los resultados de la suma.
A continuación se plantean dos casos especiales de suma de dos números de 3 bits

0 1 1 (+3)	0 1 1 (+3)
+ 1 1 1 (-1)	+ 0 0 1 (+1)
(1)0 1 0 (+2)	1 0 0
Arrastre (carry)	Desborde (overflow)

En el primer ejemplo se produce un acarreo desde la posición más significativa (bit de la izquierda). En este método se descarta dicho acarreo, y el resultado de la operación es correcto.
En el segundo caso se produce un desbordamiento, y el resultado es erróneo. Si bien el resultado "parece" un 4 si se lo analiza como un número sin signo, al trabajar en forma signada, el primer dígito indica un número negativo, lo que resulta claramente erróneo.

Suma y resta en complemento a uno

Las consideraciones generales son similares a las de C-2. La diferencia está en el tratamiento del bit de arrastre que se genera a partir de la posición más significativa. Este no se descarta, sino que se vuelve a sumar con la posición menos significativa del resultado obtenido.

```
  1 0 0 1 1 (-12)
+ 0 1 1 0 1 (+13)
1 0 0 0 0 0
  └──────┘
  +      1
  0 0 0 0 1
```

Corrección del resultado

El hecho de haber dos representaciones para el cero, y la necesidad potencial de realizar otra suma para agregar el bit de arrastre son dos razones importantes para que los diseñadores prefieran la aritmética de C-2 antes que la de C-1.

Overflow y Carry

Junto con el resultado de cada operación, la UAL genera varios indicadores (flags) acerca del mismo, que se conocen por sus iniciales inglesas **S,Z,V,C**.
Indicador de Signo **S=1** indica que el resultado de la operación es negativo
Indicador Z de resultado cero **Z=1** si el resultado es cero ("zero")
Indicador C de acarreo **C=1** si existe un acarreo ("carry")
Indicador V de overflow o desborde **V=1** si el resultado de una suma entre números con bit de signo excede el mayor valor positivo o negativo que se puede representar.

Tanto en la representación en Ca1 como en Ca2 una operación puede dar como resultado un número que excede la capacidad de la palabra de memoria, produciéndose así el overflow.

Al sumar dos números el overflow se puede dar sólo si los dos tienen el mismo signo; la suma de dos números de distinto signo nunca dará como resultado un número con módulo mayor al de mayor módulo de los dados, a lo sumo será igual (al sumarle 0 a otro número), pero en general será menor, por lo tanto no puede exceder la capacidad de la palabra de memoria.

El overflow se reconoce cuando los bits de signo de los dos números que se suman son iguales entre sí pero distintos del bit de signo del resultado, o sea cuando los números son positivos y da resultado negativo o viceversa. En este caso el contenido de la palabra de memoria es incorrecta.

Representación numérica en Punto Flotante

La representación de números en formato de punto fijo ubica la coma decimal en una posición fija, por lo que tiene una cantidad fija y determinada de dígitos tanto a la izquierda como a la derecha de la coma decimal. Por lo tanto, esta notación puede requerir una gran cantidad de dígitos para representar algunos números. Por ej, si queremos representar números del orden del billón, necesitaremos al menos 40 bits, puesto que 10^{12} es aproximadamente igual a 2^{40}. Por otra parte, si además se debiera representar una fracción equivalente al billonésimo, se necesitarán otros 40 bits, lo que daría por resultado una palabra de 80 bits.
En la práctica, suelen presentarse en los cálculos números con 80 o más bits (ej: distncia al sol, trayectoria de un misil, etc). Para manejar y almacenar números con 80 o más bits se requiere una buena cantidad de hardware, y por otra parte, las operaciones de cálculo pueden llegar a resolverse más lentamente cuando se trabaja con grandes cantidades de bits.
Para números decimales, esta limitación se supera utilizando notación científica. Así, 976.000.000.000.000 puede representarse como $9{,}76 * 10^{14}$, y 0,0000000000000976 puede expresarse como $9{,}76 * 10^{-14}$. Es decir, se desplaza convenientemente la coma decimal, incrementando o decrementando adecuadamente el

exponente, para mantener la relación. Esto permite representar un rango de números muy grandes y muy pequeños con sólo unos cuantos dígitos.
Esa misma técnica puede aplicarse a números binarios, representándolos de la siguiente manera:

$$+/- S * B^{+/-E}$$

Este número puede almacenarse en una palabra binaria con tres campos

- Signo: + o –
- Parte significativa o mantisa (S → significant)
- Exponente (E)

La base B está implícita y no es necesario almacenarla.

Normalización y bit implícito

Un número representado con este formato, puede tener distintas formas. Así, son equivalentes las siguientes expresiones:

$$0{,}110 * 2^5$$
$$110 * 2^2$$
$$0{,}0110 * 2^6$$

Para simplificar los cálculos con números en coma flotante, se procede a normalizar. Un número normalizado es el que tiene la forma

$$+/- 0{,}1bbb\ldots b * 2^{+/-E}$$

donde cada b es un dígito binario (1 o 0). Se desplaza la coma decimal hasta ubicarla a la izquierda del dígito no nulo más significativo. Esto implica que el bit más a la izquierda de la mantisa fuera siempre 1, por lo que no sería necesario almacenarlo. La mayoría de los métodos de representación en coma flotante no almacenan dicho bit inicial. Lo que se hace es "recortarlo" antes de empaquetar el número para su almacenamiento, recuperándolo al desempaquetar y llevar el número a su representación de mantisa y exponente. A raíz de este artificio, se obtiene un bit adicional a la derecha de la mantisa, lo que mejora la precisión de la representación. Este bit suele denominarse **bit implícito**. Por ejemplo, si en un formato determinado la mantisa luego de la normalización se representa como 0,11010, el patrón a ser almacenado es 1010 y el bit más significativo se oculta.
Como inconveniente, este esquema no permite representar el cero, por lo que su representación debe hacerse a través de un procedimiento de excepción. Veremos que se incluye una combinación especial para representar el cero.

Ejemplo 1: Representar en coma flotante el número decimal 34, en C a 2, con 1 bit para el signo, 5 bits para el exponente y 8 bits para la mantisa, con bit implícito.

- Convertir a binario: 00100010
- Notación exponencial (normalizar): $00100010 * 2^0 \rightarrow 0{,}100010 * 2^6$
- Mantisa a almacenar (con bit implícito) 00010
- Exponente: 110
- Empaquetar los elementos, según especificaciones (rellenar con 0 la mantisa)

0 00110 00010000

Ejemplo 2: Determinar el número decimal que representa el patrón 1010110010000, codificado en coma flotante, con bit oculto, según las siguientes especificaciones: 1 bit para el signo, 4 bits para el exponente y 8 bits para la mantisa, estos últimos expresados en MS

- 1 0101 10010000
- Signo: -
- Exponente: 5
- Mantisa: **0,1** 10010000
- Componiendo: $- 0{,}110010000 * 2^5 = -11001 =$ **-25**

Rango y precisión

La representación de números en **formato de punto flotante** permite representar un amplio rango de números con poca cantidad de dígitos binarios. Para ello se separan los dígitos utilizados para determinar la precisión (mantisa), de aquellos necesarios para representar el rango (exponente). Supongamos el número decimal en formato de punto flotante

$$+6{,}023 * 10^{23}$$

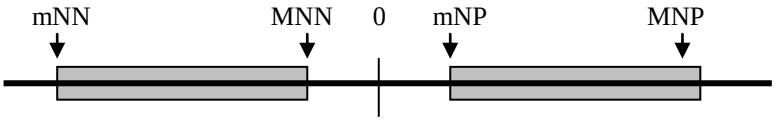
Según la convención vista (nótese que la coma decimal no se almacena)

+	2 3	6 0 2 3
Signo	Exponente	Mantisa
	Dos dígitos	Cuatro dígitos

El rango de la representación queda determinado básicamente por la cantidad de dígitos del exponente. En este caso se expresa a través de una potencia de la base 10 (10^{23}).
La precisión queda determinado por la cantidad de dígitos de la mantisa (cuatro en este ejemplo).
Si se requiere un rango mayor, y si a cambio se está dispuesto a sacrificar precisión, se pueden usar 3 dígitos para la parte fraccionaria y dejar tres dígitos para el exponente, sin necesidad de aumentar la cantidad de dígitos totales de la representación.

+	0 2 3	6 0 2

La posibilidad de plantear soluciones de compromiso entre rango y precisión es una de las ventajas principales de la representación en punto flotante.
Un inconveniente que presenta este método es que existen valores no representables. El rango de representación tiene la estructura:



- mNN es el mínimo número negativo = - máxima mantisa * base $^{\text{máximo exponente positivo}}$
- MNN es el máximo número negativo = - mínima mantisa * base $^{\text{máximo exponente negativo}}$
- mNP es el mínimo número positivo = mínima mantisa * base $^{\text{máximo exponente negativo}}$
- MNP es el máximo número positivo = máxima mantisa * base $^{\text{máximo exponente positivo}}$

Ejemplo: considérese una representación en coma flotante en la que se tiene 1 bit de signo, un exponente de 2 bits en notación exceso 2 y una mantisa normalizada binaria, con 3 bits, con el primer 1 no implícito.

Exponente mínimo (representación en exceso en 2 bits): $- 2^{n-1} = - 2^{1} = - 2$
Exponente máximo (representación en exceso en 2 bits): $2^{n-1} - 1 = 2^{1} - 1 = 1$
Mínima mantisa: 0,100
Máxima mantisa: 0,111

mNN = - máxima mantisa * base $^{\text{máximo exponente positivo}}$ = $- 0{,}111 * 2^{1}$

MNN = - mínima mantisa * base $^{\text{máximo exponente negativo}}$ = $- 0{,}100 * 2^{-2}$

mNP = mínima mantisa * base $^{\text{máximo exponente negativo}}$ = $0{,}100 * 2^{-2}$

MNP = máxima mantisa * base $^{\text{máximo exponente positivo}}$ = $0{,}111 * 2^{1}$

Estándar del IEEE[1] (norma 754)

Existen muchas maneras de representar números en formato de punto flotante. Cada uno tiene características propias en términos de rango, precisión y cantidad de elementos que pueden representarse. En un esfuerzo por mejorar la portabilidad de los programas y asegurar la uniformidad en la exactitud de las operaciones en este formato, el IEEE desarrolló su norma IEEE 754. Este estándar ha sido ampliamente adoptado y se utiliza prácticamente en todos los procesadores y coprocesadores matemáticos actuales.

Formatos

a) Simple precisión (32 bits)

Signo	Exponente	Mantisa
31	30 - 23	22 - 0

Exponente: 8 bits. Se utiliza una representación conocida como "sesgada".
El sesgo toma el valor $(2^{k-1} – 1)$, donde k es el número de bits disponibles para la representación del exponente. En este caso $2^{8-1} – 1 = 127$. Este sesgo se suma al exponente original, y el resultado es el que se almacena: E = EO + S
Mantisa: 23 bits. Bit implícito → 24 bits efectivos. Está normalizada. La normalización toma la forma 1,bb..b, donde el patrón bb..b representa los 23 bits de la mantisa que se almacenan. Nótese que la coma en esta normalización está a la derecha del primer dígito significativo.

b) Doble precisión (64 bits)

Signo	Exponente	Mantisa
63	62 – 52	51 - 0

Exponente: 11 bits. Sesgo $2^{11-1} – 1 = 1023$
Mantisa: 52 bits. Bit implícito → 53 bits efectivos. Igual consideración para la normalización.

Algunas combinaciones se emplean para representar valores especiales. Se presentan los siguientes casos:

- Un exponente cero junto a una mantisa cero representa el cero positivo o negativo, dependiendo del bit de signo. (00000000 00000000000000000000000 → 0)
- Un exponente todo unos junto con una mantisa cero representa, dependiendo del bit de signo, el infinito positivo o el negativo (11111111 00000000000000000000000 → ∞)
- Un exponente todos unos junto a una mantisa distinta de cero representa un NaN (not a number), y se emplea para señalar una excepción (11111111 xxxxxxxxxxxxxxxxxxxxxxx → NaN)

Suma y resta en formato de punto flotante

Las operaciones aritméticas en este método difieren del visto en punto fijo en el hecho de que, además de considerarse las magnitudes de los operandos, tiene que considerarse también el tratamiento que debe darse a sus exponentes. Como en el caso habitual de las operaciones decimales en notación científica, los exponentes de los operandos deben ser iguales para poder sumar o restar. Se suman o restan las mantisas según corresponda, y se completa la operación normalizando el resultado.
Los procesos de ajuste de la mantisa y de redondeo del resultado pueden llevar a una pérdida de precisión.
Hay cuatro etapas básicas en el algoritmo para sumar o restar

- Comprobar valores cero
- Ajuste de exponente y mantisa
- Sumar o restar las mantisas
- Normalizar el resultado

[1] Instituto de Ingeniería Eléctrica y Electrónica de Estados Unidos

Para la operación de suma o resta, los dos operandos deben transferirse a registros que serán utilizados por la ALU. Si el formato incluye un bit de mantisa implícito, dicho bit debe hacerse explícito para la operación.
Dado que la suma y la resta son idénticas, excepto por el cambio de signo, el proceso comienza cambiando el signo del substraendo cuando se trata de una resta. A continuación, si alguno de los operandos es cero, se da el otro como resultado.
Luego se procede al ajuste de los exponentes, en sucesivas operaciones. Si en este proceso se obtiene una mantisa cero, se da el otro número como resultado. Esto significa que cuando dos números tienen exponentes muy diferentes, se pierde el menor de los números.
Luego se suman las mantisas, teniendo en cuenta sus signos. Ya que los signos pueden diferir, el resultado puede ser cero. Existe también la posibilidad de desbordamiento de la mantisa en un dígito. Si es así, se desplaza a la derecha la mantisa del resultado (se pierde el dígito menos significativo), y se incrementa el exponente. Como resultado podría producirse un desbordamiento en el exponente. Esto se debe indicar, y la operación se detendría.
Por último, se normaliza el resultado.

Ejemplo: sumar $(0{,}101 * 2^3) + (0{,}111 * 2^4)$

- Se iguala el menor de los exponentes al mayor, modificando en forma acorde la mantisa

$$0{,}101 * 2^3 = 0{,}010 * 2^4$$

En este proceso se pierde precisión en la cifra menos significativa del valor original

- Se suma: $(0{,}010 + 0{,}111) * 2^4 = 1{,}001 * 2^4$
- Se normaliza: $0{,}1001 * 2^5$
- Se vuelve a redondear a 3 dígitos: $0{,}100 * 2^5$ perdiéndose precisión nuevamente

Codificación de la información

Para permitir la comunicación de datos existe una primera cuestión a tener en cuenta: el Hardware. Pero esta es solo una parte. Todas las personas del mundo tienen el mismo hardware para la comunicación hablada: labios, lengua, dientes y el resto del complejo aparato bucal para transmitir y los oídos para recibir. La comunicación oral, sin embargo, es posible solamente cuando dos personas conocen el mismo lenguaje, es decir la misma manera de **codificar la información.-**
Así como el habla sería imposible sin lenguajes comunes, la comunicación entre computadoras sería imposible sin coordinación de *códigos de caracteres.-*

Todas las computadoras digitales actuales usan un **lenguaje binario** para representar la información internamente. Debido a que algunos de los dispositivos con los que se deben comunicar las computadoras están diseñados para uso humano (especialmente impresoras y terminales de video), es importante que esos **periféricos** utilicen un código de comunicación compatible con la comunicación humana.
Existen varios métodos para alcanzar dicha compatibilidad y cada uno utiliza un modo diferente de codificar los números y las letras que conforman la base de la comunicación escrita entre las personas.
Actualmente es frecuente encontrar en un *sistema de cómputo* dispositivos provistos por distintos fabricantes. La posibilidad de conectarlos existe únicamente si esos dispositivos utilizan un código común para la transmisión y recepción de la información.
Son claras las ventajas de conseguir que todas las computadoras utilicen el mismo código de comunicación. Aún cuando la calidad de los códigos varía enormemente, casi cualquier estándar universal sería mejor que ninguno. Si bien hay códigos que prácticamente son aceptados por todos los fabricantes, aún no existe un estándar que optimice el aprovechamiento de los recursos del hardware.

La codificación consiste en establecer una ley de correspondencia, llamada **CÓDIGO**, entre la información por representar y las posibles configuraciones binarias, de tal manera que a cada información corresponda una y generalmente solo una, configuración binaria.
Llamamos **CODIFICACIÓN** al proceso de convertir un símbolo complejo en un grupo de símbolos más simples. Ejemplo: convertir una letra del alfabeto en un código de cinco bits.
DECODIFICACIÓN es el proceso inverso al de codificación, se convierte a un código donde la cantidad de símbolos es menor, pero cada una contiene más información.
TRANSCODIFICACIÓN: Aplicación de un cambio de código a una información ya codificada. Ejemplo: de EBCDIC a ASCII.

Codificación de la información no numérica

A diferencia de los números reales, que tiene un rango infinito, existe sólo una cantidad finita de caracteres representables.

El concepto de caracter aparece como la cantidad de BITS necesarios para representar los diferentes símbolos del alfabeto (letras, cifras, signos de puntuación, etc.)
De acuerdo al código utilizado cada caracter puede codificarse con un número varible de BITS, pero dentro de un sistema todos los caracteres se representan con el mismo número de bits.

Generalmente, los caracteres se agrupan de la siguiente manera:
Caracteres alfabéticos: letras mayúsculas y minúsculas: A-Z, a-z
Caracteres numéricos: cifras decimales: 0-9
Caracteres especiales: son caracteres no incluidos en los grupos anteriores y que se utilizan con mucha frecuencia: Ñ ñ / + = ! ? ’) (, . >< # …
Caracteres gráficos: Suelen representar iconos o figuras elementales: © ┤ ╫ ☺ ▒
Caracteres de control: representan ciertas órdenes de control, como control de carro, final de archivo, tabulación, etc.

Condiciones que se imponen para la codificación de caracteres

A) El código debe englobar al de los números, y permitir distinguir las cifras rápidamente de los otros caracteres.
B) La representación debe permitir añadir nuevos caracteres específicos para una aplicación determinada.
C) En el caso de las transmisiones, la representación debe incluir un sistema de redundancia que permita la detección de errores.

A continuación se describen tres representaciones habituales en la codificación de caracteres: los códigos ASCII, EBCDIC y Unicode.

1) Código ASCII (American Standard Code for Information Interchange)

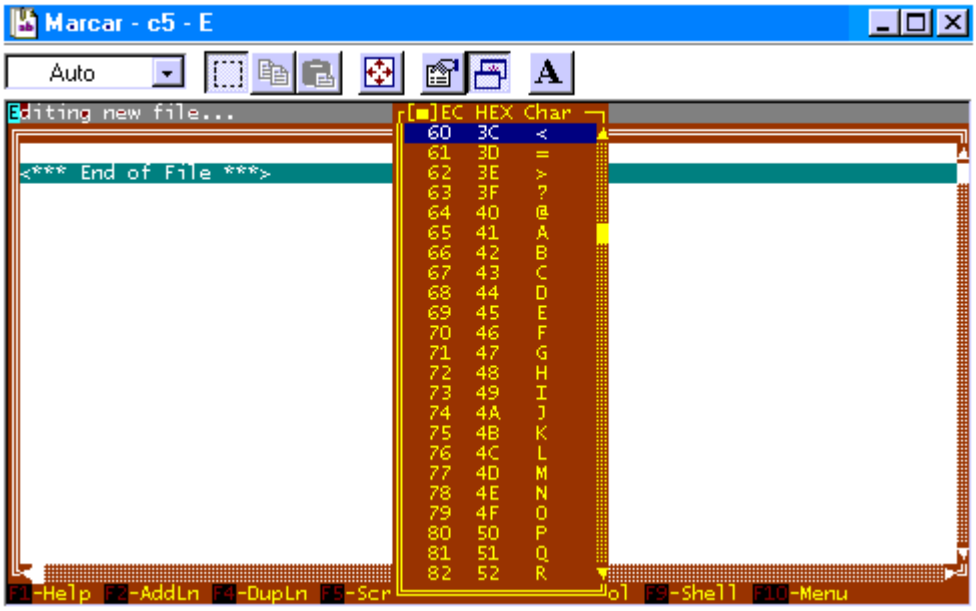
El código ASCII se definió inicialmente con 6 bits, esto permitía representar $2^6 = 64$ caracteres. Posteriormente el ANSI (Instituto Nacional Norteamericano de Normas) definió un nuevo ASCII (que se mantiene como norma) de 7 bits. Esta nueva definición permite codificar 128 caracteres; haciéndolo más apto, fundamentalmente para la transmisión, donde parte de los “caracteres” codifican funciones de control.

Existen códigos ASCII ampliados (no normalizados) que usan los 8 bits para codificar una serie de caracteres gráficos especiales, incluidas las vocales con acento.
La mayor parte de las máquinas actuales utilizan el código normalizado, pero generalmente agregan un octavo bit, que les permite extender el código para representaciones no previstas o para utilizarlo como bit de control de paridad en las transmisiones. Este código es ampliamente usado para la transmisión de información.
El éxito de este código se basa en que cumple con todas las condiciones impuestas para la codificación de caracteres:
Los valores correspondientes a las letras del alfabeto y a los restantes caracteres, siguen una secuencia binaria continua, la computadora no tiene que dejar su propio lenguaje binario para realizar operaciones secuenciales con esos caracteres. Además con el agrupamiento de las funciones de control, con solo analizar los dos primeros bits de una combinación cualquiera codificada, la computadora puede determinar si se trata de una función de control (dos ceros) o de un caracter (uno de los dos no es cero).
Las mayúsculas de la A a la Z se codifican según una sucesión ordenada de números binarios, que permite realizar ordenaciones alfabéticas:
$A = 01000001 = 41_{(16} = 65_{(10}$; $B = 01000010 = 42_{(16} = 66_{(10}$; **…** $Z = 01011010 = 5A_{(16} = 90_{(10}$

Lo mismo ocurre con los dígitos decimales:

$0 = 00110000 = 30_{(16} = 48_{(10}$; $1 = 00110001 = 31_{(16} = 49_{(10}$; **…** $9 = 00111000 = 39_{(16} = 57_{(10}$

Existen teclas de control correspondientes a órdenes mecánicas, como la barra espaciadora (SP en ingles, de SPace), la tecla de retorno de carro a un nuevo renglón (CR de Carry Return), que aparece como ↵ "Enter" en los teclados, la de retroceso ← (BS o Back Space).
Estas teclas sirven para organizar la impresión de caracteres en renglones de un papel o de una pantalla, y se conocen como caracteres no imprimibles, siendo en esencia órdenes para desplazar el carro de una máquina de escribir o su equivalente cursor en una pantalla.
Existen diversas maneras de presentación de una tabla con el código ASCII, las cuales mayormente indican, por razones de comodidad, o para operar desde el teclado, las equivalencias hexadecimales o decimales de cada combinación binaria del código, sin indicar ésta. Esto puede observarse en la siguiente figura, que muestra la tabla de equivalencias a la que se accede desde un editor de texto sin formato.

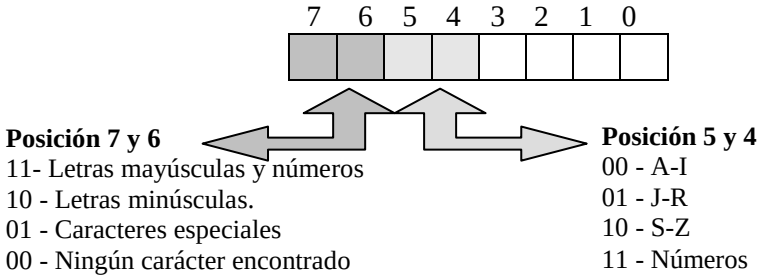


Representación de los Números

Cada dígito decimal es representado internamente con 8 bits, distribuidos de la siguiente forma:
ZONA: Es siempre 0011
DIGITO: En este espacio (4 bits) se representa el número decimal codificado en BCD
Los datos numéricos así codificados no son técnicamente aptos para ser procesados aritméticamente. Si es necesario realizar operaciones aritméticas con ellos, se debe eliminar la parte correspondiente a la ZONA de cada byte. Esta operación se llama empaque y la información resultante, información empacada (empaquetada) o decimal sin zona. Los datos numéricos con zona se denominan información desempacada o zoneada.

2) Código EBCDIC(Expanded Binary Code Decimal Interchange Code)

Este código fue diseñado y utilizado exclusivamente por IBM. Su importancia radica en que sirvió como base para los códigos posteriores normalizados.
Utiliza 8 dígitos binarios para representar cada carácter, por lo tanto hay un total de 2^8=256 caracteres posibles. La correspondencia entre la información por representar y la correspondiente secuencia binaria se encuentra en tablas con distintos formatos.
Con respecto a los caracteres alfabéticos y signos de puntuación no tiene características que lo destaquen, salvo que podemos encontrar pequeñas alteraciones al ser utilizados en países con distintos alfabetos.
La estructura del Byte según la representación es la siguiente:



Representación de los Números

En este código, al igual que en el ASCII, cada dígito decimal es representado internamente con 8 bits, distribuidos de la siguiente forma:

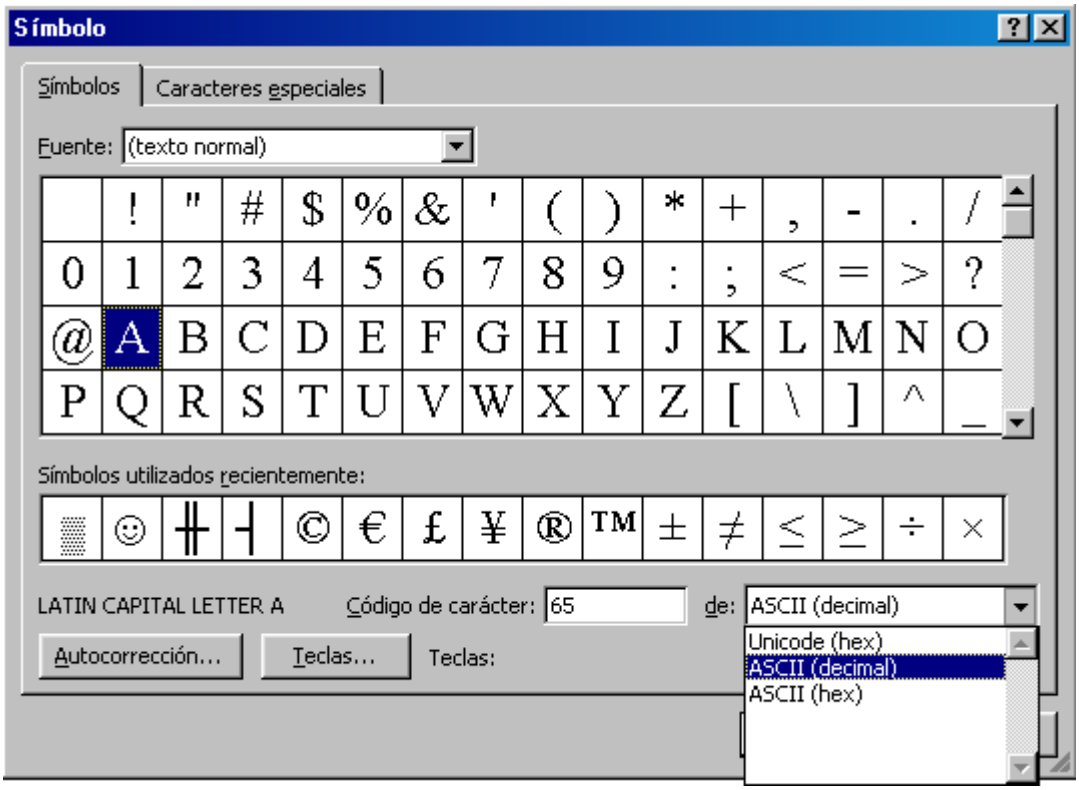
ZONA: Ocupa los 4 bits de orden superior del Byte. Tiene una secuencia binaria fija para cualquier número: $\mathbf{1111_{(2} = F_{(16}}$

DIGITO: En este espacio se representa el número decimal codificado en **BCD**

Aquí surge el mismo inconveniente visto en el código ASCII: la dificultad para realizar operaciones matemáticas con los números representados con la ZONA. La solución es la misma que la indicada para el código anterior. Como la mayor parte de las máquinas utilizan 8 bit para cada dígito no surgen conflictos al producirse el EMPAQUE.

3) El código UNICODE

Los códigos ASCII y EBCDIC sirven para soportar los conjuntos de caracteres (latinos) históricamente dominantes en las representaciones de la computación. Existen muchos otros conjuntos de caracteres en uso en el mundo, y no siempre es posible la conversión de código ASCII al código correspondiente al idioma X. A raíz de esto surgió un nuevo conjunto de caracteres, universal y normalizado, al que se conoce como Unicode, y que sirve para soportar una buena cantidad de alfabetos que se usan en el mundo.
Este código es una norma en desarrollo, que se modifica a medida que se le incorporan los símbolos correspondientes a alfabetos nuevos, y a medida que los conjuntos de caracteres incorporados se van modificando y se refinan las correspondientes representaciones. En la versión 2.0 de Unicode se incluyen 38.885 caracteres diferentes, los que cubren los principales lenguajes escritos de uso en América, Europa, Medio Oriente, Asia, India y las islas del Pacífico.
Utiliza un conjunto de caracteres de 16 bits. Si bien este código soporta muchos más caracteres que ASCII y EBCDIC, no es la norma de mayor importancia.
Los primeros 128 caracteres del Unicode coinciden con los del ASCII.
La siguiente figura muestra los símbolos disponibles para inserción desde un documento de Word. En la ventana puede observarse la disponibilidad de caracteres Unicode y ASCII, y el código decimal o hexadecimal correspondiente.



Códigos redundantes

Es difícil pensar en un equipo que funcione sin fallas durante un tiempo indefinido. Para cualquier máquina se define un Tiempo Medio Entre Fallas (MTBF). El objetivo de los desarrollos tecnológicos es incrementar ese tiempo.

Teniendo en cuenta esa premisa, es de esperar que la información pueda verse alterada en el transcurso de la transmisión o almacenamiento. Si el equipo tiene la posibilidad de detectar o, mejor aún, corregir esas modificaciones, es evidente que aumentará la confiabilidad del mismo. Para este fin, existen dos tipos de códigos:

1. Códigos autodetectores

Código en el que mediante un determinado número de bits de redundancia se puede detectar si la información recibida es correcta o no. El ejemplo mas clásico de este tipo de códigos es el de *control de paridad*.

1.1. Control de paridad

Este código, si bien no permite detectar errores dobles, es el más utilizado debido a su simplicidad y a que en los ordenadores la probabilidad de que ocurra un error es muy pequeña, por lo tanto que ocurran 2 es mucho menos probable.
Consiste en agregar a los bit de información transmitidos un bit mas (generalmente el primero de la izquierda), que hace que la cantidad de unos transmitidos sea PAR (PARIDAD PAR – pone un 0) o IMPAR (PARIDAD IMPAR – pone un 1).

Ejemplo: Se desea enviar los caracteres ASCII ‘A’ (código decimal 65) y ‘C’ (código decimal 67) desde un equipo emisor a otro receptor, utilizando una comprobación de paridad par:

Emisor (Codificación del Byte del carácter para comprobación de paridad PAR)

Carácter ASCII	Código binario del dato	Bit de paridad	Byte completo
'A' (65)	1000001	0 (el número de unos es par)	**0** 1000001
'C' (67)	1000011	1 (el número de unos es impar)	**1** 1000011

Cuando el receptor recibe la información lo que hará será volver a contar el número de unos que tiene el dato y comprobar que concuerda con lo especificado en el bit de paridad. Si llega:

Receptor (Comprobación de paridad PAR)

Byte recibido	Dato	Nº de unos recibido	Bit de paridad que tiene	Error
0 1000001	1000001 'A'	2 (par)	0	NO
1 1000001	1000001 'A'	2 (par)	1	SI

Se puede detectar que en el segundo carácter transmitido hay un error porque el bit de paridad no concuerda con lo que se expresa en los bits de datos: el número de unos es impar pero el bit de paridad indica que debería ser par.

2. Códigos correctores

Mediante el uso de estos códigos, el receptor puede determinar si la información recibida es correcta o no y en este caso corregir el error producido durante la transmisión.

2.1. Códigos de Hamming

Estos códigos permiten detectar y corregir uno o más errores producidos durante la transmisión para palabras de cualquier número de bits.
Consiste en determinar en cuántos bits difiere la palabra recibida de la enviada, a lo que se llama distancias de Hamming. Para ello, la información *n* enviada consiste de *m* bits de datos y *r* bits redundantes, de control de paridad, siendo $n = m + r$

2.2. Control 2 en 3

Para transmitir una información cualquiera de **"n"** bits, se envían 3 veces esos **"n"** bits, en forma sucesiva. Al receptor de la información, al efectuar el análisis de la misma, pueden presentársele tres situaciones distintas:

- Las tres son idénticas. La información se toma como correcta.
- Dos son iguales y una distinta. El código se comporta como AUTOCORRECTOR, selecciona una de las dos iguales y la toma como correcta.
- Las tres son distintas. El código se comporta como AUTODETECTOR, la máquina detecta que hay error pero no puede determinar cual es la información correcta.