# 3. Representación de los datos

> "El concepto 'perro' no muerde".
>
> *William James*
>
> *"Some problems of philosophy"*

Se suele decir que los computadores trabajan en binario. Esto quiere decir que los componentes un computador pueden "recordar" únicamente dos estados, a estos estados los llamamos 0 y 1, y los identificamos con dichos números.

¿Por qué no tres, cuatro o diez estados? porque hay dificultades tecnológicas para el desarrollo de tales elementos; pero, en principio, se podría pensar en desarrollar un computador que trabaje base $n$, si se dispone de elementos con $n$ estados estables.[1]

Tenemos entonces que el computador sólo puede "recordar" los dígitos 0 y 1, y, por ende, series de los mismos. Esto no es suficiente para nuestros fines: necesitamos números positivos y negativos, letras y otras entidades más complejas —imágenes, vídeo, sonidos, etc.—.

En conclusión, el objetivo de las secciones siguientes es definir convenciones para representar diferentes entidades —números, caracteres y valores lógicos— usando unos y ceros. Estas nos permitirán:

- A partir de una cadena de dígitos binarios, y sabiendo cuál es el tipo de la entidad codificada, determinar qué representa.
- Codificar una entidad determinada usando dígitos binarios.

## De significados y significantes

Debemos tener presente lo que quiere decir "representar"; todo símbolo está compuesto por dos partes: la entidad representada (significado) y lo que la representa (significante); una cosa es un árbol y otra la palabra "árbol". En

[1] De hecho, aunque no llegaron a nada concreto, los rusos estudiaron la posibilidad de construir computadores base 3. Por otro lado, hay memorias *flash* que trabajan en base 4, y algunos sistemas de comunicación también transmiten la información en bases superiores a 2.

principio, el significante es arbitrario y no tiene por qué tener una relación objetiva con el significado (“perro”, en otros idiomas, se dice “dog” o “chien”; ninguna de estas palabras tiene algo especialmente canino, y todas designan el mismo cuadrúpedo).

El significante está ligado a la sintaxis, a la forma; es alguna entidad física que puede ser captada. El significado está más ligado a la semántica, al fondo. Para pasar de uno al otro es necesario algún sistema, algunas reglas. Veamos un caso: ¿qué quiere decir "PIE"? Depende de si lo interpretamos en inglés o en español; sin embargo, las letras son las mismas; nosotros decidimos cómo las interpretamos.

Igualmente, las series de dígitos binarios no quieren decir nada por sí mismas; nosotros las interpretamos como una cosa u otra. Es responsabilidad del programa —y, por ende, del programador— manejar los datos correctamente; el computador ejecutará ciegamente lo que se le ordene. Recuerde la frase: "el computador no hace lo que usted quiere que haga, sino lo que usted le dice que haga".

### Espacio de codificación

Cuando nos comunicamos, no podemos enviarnos los significados directamente; lo más que podemos hacer es enviarnos formas sensibles, es decir, formas que pueden ser captadas por los sentidos —como un sonido o una imagen—, o que, en todo caso, tienen existencia física —como las ondas electromagnéticas—. El receptor del mensaje extrae el significado del mensaje a partir de su representación física. Esto implica que el receptor debe interpretar, en su interior, al significante; la semántica de un símbolo no está en el significante sino en el interior de quien lo interpreta.

Puesto que un símbolo solo se puede distinguir unívocamente por la forma de su significante, para representar un conjunto de entidades necesitamos tantos significantes como entidades haya. Por supuesto, esto implica que cuantas más entidades haya, tantos más símbolos diferentes se requerirán, lo cual puede tornar inviable al sistema de representación. Una forma de evitar este problema consiste en tener un conjunto base de símbolos, llamado *alfabeto*, el cual es finito. Los símbolos del alfabeto se combinan entre ellos formando *palabras*, o *codificaciones*, que son los verdaderos significantes del sistema.

Si tenemos un alfabeto de tamaño $n$, y palabras de $k$ letras, el espacio de codificación es de tamaño $n^k$. Note que, en este caso, no podemos representar más de $n^k$ entidades (aunque sí podemos representar menos, desperdiciando algunos códigos o usándolos para redundancia, como veremos más adelante). Suponiendo que $n$ es fijo, de todas maneras se puede representar la cantidad de entidades que se desee seleccionando un $k$ apropiado.

### Sinónimos y homónimos

En la presentación anterior se supuso una relación uno a uno entre significantes y significados: a cada significante le corresponde uno y solo un significado, y viceversa. Este supuesto excluye el caso de los sinónimos —varios significantes asociados a un mismo significado—, y el de los homónimos —un significante asociado a varios significados.

Cuando se tienen sinónimos, se necesita un espacio de codificación más grande, puesto que, de cierta manera, se están "desperdiciando" códigos. Aunque esto puede parecer así, este "desperdicio" puede ser productivo; en el caso de la informática, como veremos más adelante, este mecanismo se puede utilizar para protección de la información: detección y corrección de errores.

En el caso de los homónimos, aunque se requieren menos significantes, se dificulta la interpretación puesto que se genera el problema de distinguir entre los dos o más significados que puede tener un significante; normalmente esto se resuelve por el contexto: el contexto, de alguna manera, da pistas sobre cuál es la interpretación que se le debe dar al significante. En el caso de la informática, este tipo de codificaciones puede servir, por ejemplo, para generar codificaciones de tamaño variable.

## 3.1 LOS NÚMEROS NATURALES

En informática, cada dígito binario, 1 ó 0, se llama *bit* —del inglés *BInary digiT*—. Un procesador maneja los bits en grupos de tamaño limitado; es decir, no puede efectuar las operaciones primitivas sobre series de bits tan largas como se quiera. Las operaciones se realizan sobre secuencias de dígitos de un tamaño determinado; estas secuencias son llamadas *palabras*. El tamaño de palabra depende de la máquina; no hay un estándar; se encuentran de 8, 16, 32 y más bits.

**En sus propios términos**

"[La guerra] empezó con el siguiente acontecimiento. Todos reconocen que la forma original de cascar huevos, antes de comerlos, era por el extremo más grande. Pero el abuelo de su Majestad actual, cuando era niño, iba a comer un huevo, y lo rompió siguiendo la antigua tradición, ocurriéndole que se cortó un dedo. Tras lo cual el emperador, su padre, publicó un edicto ordenándole a todos sus súbditos, bajo graves penas, romper los huevos por el extremo pequeño [...]. Muchos cientos de volúmenes se han publicado acerca de esta controversia; pero los libros de los ***Big-Endians*** han estado prohibidos por mucho tiempo".

*Jonathan Swift*

*"Gulliver's Travels"*

En una palabra, los bits se numeran a partir de cero. En algunos computadores, se numeran de derecha a izquierda; en otros, de izquierda a derecha. Es decir, en unos casos, el bit menos significativo tiene el subíndice cero; en otros, es el más significativo quien lo tiene. Los computadores con numeración de izquierda a derecha son llamados *"Big endian" a nivel bit*. Esto porque empieza a numerar por el bit más significativo, o sea, el más "grande". Si la numeración es de derecha a izquierda, son llamados "*Little endian*" (fig. 3.1). La historia de estos nombres es más larga y tiene que ver con los huevos cocidos.

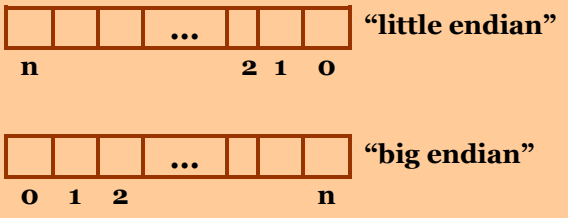


**Fig. 3.1. Numeración de los bits dentro de una palabra**

En la IA32 se usa la numeración de derecha a izquierda (es *little endian* a nivel bit).

Aparentemente el tamaño de palabra es una limitante, sin embargo, esta restricción puede ser corregida por software: una entidad muy extensa —como un número muy grande o una cadena de caracteres— se puede representar usando varias palabras.

Dado que sólo disponemos de 1 y 0, lo más natural para representarlos los números es el sistema binario. Puesto que estamos limitados por el tamaño de palabra, es necesario expresarlos usando un cierto número fijo de bits.

Esto implica que vamos restringirnos a un cierto rango de valores. Por ejemplo, si las palabras son de ocho bits, solamente podemos representar números en el rango `00000000…11111111`, es decir, 0 a 255. En general, si se tiene palabras de $n$ bits, se pueden representar $2^n$ números, que cubren el rango de valores 0 a $2^n$-1.

Por supuesto, el valor máximo puede incrementarse usando varias palabras. Por ejemplo, usando dos palabras de 8 bits, obtendríamos un valor máximo de: `11111111 11111111`. Ahora, dado que el procesador no puede manipular más de una palabra, el software debe ocuparse de sumar las dos palabras.

**Números con signo: signo y magnitud**

La notación anterior sólo permite representar naturales —números sin signo—, y la convención más relevante es el número de bits utilizados en la codificación. El caso de los enteros —números con signo— es diferente, ya que no existe una notación "natural" para el signo. Hay que desarrollar algún método para simbolizarlo.

Una representación, poco usual, es la llamada *signo y magnitud;* el principio del método es utilizar un bit para indicar el signo y el resto para la magnitud en valor absoluto. Podemos tomar el primer bit de la izquierda, el más significativo, como el signo: si vale uno, el número será considerado negativo, de lo contrario será positivo. Así, el número 6, en ocho bits, se escribe:

```
00000110
```

Mientras que -6 es:

```
10000110
```

Este método es conceptualmente simple, pero su realización es complicada, ya que al realizar una adición hay que revisar los signos de los dos números y realizar acciones diferentes dependiendo de si son positivos o negativos. Debido a esto, aunque ha habido máquinas que usan signo y magnitud, es, en general, poco utilizado.

En esta representación, las convenciones más importantes son el número de bits utilizados, la posición del signo —bit más significativo— y la forma de representar el signo —uno es negativo; cero, positivo—.

**Números con signo: complemento a dos**

El método más popular para representar números enteros es el *complemento a dos*. El objetivo de este método es simplificar los circuitos de suma. Para esto, la

restricción que se impone es que los circuitos deben funcionar igual sin importar que los sumandos sean positivos o negativos; se tiene que buscar una convención para los números negativos que funcione con un circuito sumador común y corriente. A continuación mostramos cómo se puede lograr ese objetivo, pero, por claridad, debemos empezar por explicar una particularidad de los circuitos sumadores.

Supongamos que se está trabajando con palabras de $n$ bits. En palabras de este tamaño, se pueden representar números entre 0 y $2^n$-1. La aritmética se encuentra restringida por dichos valores, debido a lo cual, es necesario usar una aritmética especial: la *aritmética modulo* $2^n$. Lo que tiene de especial es que el resultado de toda operación se calcula módulo $2^n$, de otra manera los resultados no cabrían en una palabra. La suma viene definida por:

$$x +_{\text{mod}2n} y = (x + y) \text{ MOD } 2^n$$

Lo cual equivale a decir que únicamente se toman los $n$ bits menos significativos del resultado. Veamos un ejemplo con palabras de 4 bits: al sumar 1 a 0000 obtenemos 0001, y así sucesivamente hasta llegar a 1111, pero al sumar 1 a este último, obtenemos 0000 de nuevo. Se puede visualizar como un sumador circular (ver fig. 3.2).

Note que con el sumador circular 15 + 1 = 0, 15 + 2 = 1, etc. Es decir, 15 se comporta de la misma manera que lo hace -1 en la aritmética tradicional; igualmente, 14 se comporta como -2; 13, como -3, etc. De cierta forma se puede decir que 15 y 14 están "a la izquierda del 0", tal como ocurre con los negativos en la recta numérica.

Otra forma de visualizar lo que ocurre es imaginar que se suman segmentos de arco. El número 15, por ejemplo, es un segmento circular que casi da la vuelta al sumador, pero le falta uno para lograrlo. Si ponemos este segmento a continuación de otro cualquiera, el punto final estará una posición atrás del final del segmento del cual partimos. Es decir, se habrá restado 1 al segmento original. Si agregamos 14, faltarán dos unidades para llegar; luego, adicionar 14 a otro número equivale a restar 2. Por lo tanto, podemos representar los números negativos usando números positivos que se comportan como los negativos al usar la "suma circular".

En general, un número negativo $-y$ ($y > 0$), se comporta igual que el número $2^n$-$y$. Así que podemos definir la representación en complemento a dos por:

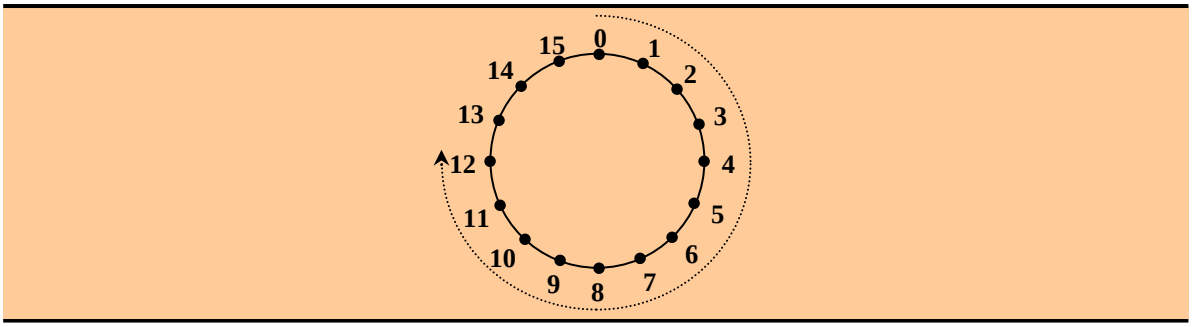


**Fig. 3.2. Sumador circular**

$$\text{Complemento2}(x) = x \quad \text{, si } x \geq 0$$
$$= 2^n + x \quad \text{, si } x < 0$$

Se puede expresar en una sola formula:
$\text{Complemento2}(x) = (2^n + x) \text{ MOD } 2^n$

Note que la representación de los números positivos en complemento a 2 es el mismo binario de siempre; sólo los negativos se representan de manera especial.

Veamos unos ejemplos usando 8 bits. El número 3 es 00000011; -3 se representa por $2^8+(-3) = 253$, en binario esto es 11111101. La suma de estos dos números da como resultado 1 00000000, pero, dado que usamos aritmética módulo $2^8$, tomamos como resultado los 8 bits menos significativos obteniendo 00000000, lo cual es el resultado esperado de (3 + (-3)).

De los $2^n$ códigos disponibles, se destinan unos para los números positivos y otros para los negativos. La convención adoptada es que los números entre $2^n$-1 y $2^{n-1}$ representan negativos. Se eligieron estos dos valores por dos razones: primero, porque de esta manera la mitad de los números son positivos y la otra mitad negativos; segundo, porque así es fácil identificar los números negativos. En efecto, los números entre $2^n$-1 y $2^{n-1}$ tienen la particularidad de empezar por 1, luego, si el primer bit por la izquierda de un número es 1, el número es negativo. Note que el primer bit *denota* el signo, pero no *es* el signo.

En el caso de las palabras de 8 bits, los números positivos están entre `00000000` y `01111111`, es decir entre 0 y 127. Los negativos se representan por los números entre `10000000` y `11111111`, es decir entre 128 y 255, y los valores representados son -1 a -128. En general, los positivos van de 0 a $2^{n-1}$-1, y los negativos, de -1 a $-2^{n-1}$.

Un aspecto que no hemos considerado es cómo obtener el negativo de un número; es decir, teniendo $x$, cómo se calcula $-x$. En principio, tenemos una fórmula para hacerlo, pero esta fórmula implica restarle a $2^n$ y no tenemos capacidad para realizarla (tenemos aritmética de $n$ bits y $2^n$ ocupa $n+1$ bits). Por suerte, los números binarios tienen una característica que simplifica esta operación:

$2^n = x + \bar{x} + 1$, para $0 \leq x < 2^n$, y representando $x$ con $n$ bits (verifique que es cierto)
Luego: $2^n - x = \bar{x} + 1$

Por lo tanto, para encontrar el negativo de un número, basta con complementar todos los bits —negarlos, en el sentido lógico del término— y sumarle uno al resultado. A manera de ejemplo, calculemos la representación de -5:

Cinco se escribe 00000101, luego -5 será:
$\overline{00000101} + 1 = 11111010 + 1 = 11111011$

El método es reversible, es decir, si se hacen las mismas operaciones sobre el número negativo, obtenemos el positivo correspondiente:

$\overline{11111011} + 1 = 00000100 + 1 = 00000101$

Ahora, supongamos que tenemos los bits 11110110; sabemos que es un número negativo, puesto que empieza por 1, pero no sabemos cuál. Para encontrar el positivo correspondiente hacemos:

$$\overline{11110110}+1 = 00001010$$

El resultado es diez, luego el número representado es -10.

Es importante notar que, desde el punto de vista de los circuitos, todos los números son positivos, pero nosotros decidimos ver algunos de estos números como negativos, aprovechando que el comportamiento es similar.

Como siempre, es responsabilidad del programador saber si un número tiene signo o no, y, si lo tiene, cuál es la representación.

Miremos cómo se comporta el complemento a dos matemáticamente:

Supongamos que tenemos dos números negativos $-x$, $-y$, y queremos calcular su suma ( $-x+(-y) = -x-y = -(x+y)$ ). Sus representaciones son $2^n-x$ y $2^n-y$, respectivamente, luego la suma módulo $2^n$ dará:

$$(2^n-x + 2^n-y) \text{ MOD } 2^n = (2^n + 2^n - (x+y)) \text{ MOD } 2^n = 2^n - (x+y)$$

Pero $2^n - (x+y)$ es la representación de $-(x+y)$, que es justamente el resultado de la suma.

Supongamos ahora que tenemos $x$, $-y$ ($x > y$). La suma módulo $2^n$ será:

$$(x + 2^n-y) \text{ MOD } 2^n = (2^n + (x-y)) \text{ MOD } 2^n = x - y$$

El resultado es positivo, puesto que $x > y$; la representación es el número mismo en binario.

Ahora bien, si tenemos $x$, $-y$, pero $x < y$ entonces:

$$(x + 2^n-y) \text{ MOD } 2^n = (2^n - (y-x)) \text{ MOD } 2^n = 2^n - (y-x)$$

Lo cual es la representación de $-(y-x)$.

Conviene anotar que el método de complemento a dos puede generalizarse a otras bases; basta con tomar la formula $b^n-x$, donde $b$ es la base que se está utilizando y $n$ es el número de dígitos de los que se dispone.

### Operaciones sobre n bits

En esta sección vamos a estudiar cómo efectuar operaciones aritméticas sobre las representaciones vistas. También se mostrarán algunos inconvenientes que surgen de representar los números con una cantidad fija de bits. Además, veremos unas operaciones nuevas que nacen de esta característica. Estas operaciones no son particularmente trascendentales, pero, son bastante utilizadas en la práctica.

#### *Operaciones aritméticas*

La suma, resta y negación son operaciones que se realizan por hardware.

En el caso de la suma, existen diversos circuitos. El más simple se construye por medio de módulos que reciben un dígito de cada operando y el acarreo del módulo anterior. Tienen dos salidas: el resultado de sumar las tres entradas y el acarreo correspondiente. Estos módulos se conectan en cascada: el acarreo de salida se conecta a una de las entradas del siguiente sumador.

Otra forma de verlos es como un módulo que recibe tres entradas de un bit, las suma, y bota el resultado como un número de 2 bits.

El primer módulo de la cascada debe tener una de sus entradas en 0, ya que no hay un acarreo anterior. Sin embargo, esta entrada también se puede poner en 1, con lo cual se realiza la operación $x + y + 1$, que, como se verá después, es útil en la resta.

Si los números negativos se representan usando complemento a dos, se pueden utilizar los circuitos sumadores antes descritos sin ningún problema. Si la representación es signo y magnitud, los circuitos se tornan más complejos, lo cual es una de las razones para preferir el complemento a 2.

La negación, usando complemento a dos, se hace complementando el número y sumándole 1, tal como se describió anteriormente. Es sencillo construir circuitos lógicos que realicen esta operación. En el caso de signo y magnitud, la negación se puede hacer complementando el primer bit —el bit de signo—.

Basados en las dos operaciones anteriores, es posible realizar la resta. Es más, sólo con el sumador se puede construir; en efecto, para calcular $x - y$, basta con activar el sumador con los operandos $x$ y $\overline{y}$, poniendo una de las entradas del primer módulo en 1. El resultado será $x + \overline{y} + 1$, que equivale a $x - y$, puesto que $-y = \overline{y} + 1$.

La multiplicación y división pueden implantarse por hardware o por software. En los primeros microprocesadores, la multiplicación y la división se hacían por software; había que escribir un programa que las calculara. Hoy en día, muchos microprocesadores poseen hardware para efectuarlas.

El método usado para la multiplicación es, *grosso modo*, el que se explicó en la sección de aritmética en otras bases, sólo que se debe adecuar para que sea fácil de hacer en hardware. A continuación, se presenta una posible realización usando números de $n$ bits:

- Se reservan $2*n$ bits para el resultado, y se ponen en cero.
- Se mira si el bit menos significativo del multiplicador es 1, en cuyo caso se suma el multiplicando a los $n$ bits más significativos del resultado.
- Se desplazan una posición a la derecha los $2*n$ bits del resultado. El bit menos significativo se pierde, y el acarreo de la suma anterior queda como dígito más significativo.
- Se desplaza el multiplicador a la derecha una posición, el bit más significativo queda en cero.
- Se regresa al paso 2 —Hay que iterar $n$ veces—.

A continuación, un ejemplo con 4 bits (multiplicador = 0101, multiplicando = 1111):

| Comentario | Multiplicador | Resultado |
|---|---|---|
| **1-** Bit menos significativo del multiplicador es **1**; luego | 0101 | 0000 0000 |
| sumar el multiplicando al resultado: | | + 1111 |
| | | 0 1111 0000 |
| Correr a la derecha Multiplicador y Resultado: | 0010 | 0111 1000 |
| **2-** Bit menos significativo del multiplicador es **0**; luego no hay que sumar el multiplicando. | | |
| Correr a la derecha Multiplicador y Resultado: | 0001 | 0011 1100 |
| **3-** Bit menos significativo del multiplicador es **1**; luego | 0001 | |
| sumar el multiplicando al resultado: | | + 1111 |
| | | 1 0010 1100 |
| Correr a la derecha Multiplicador y Resultado: | 0000 | 1001 0110 |
| **4-** Bit menos significativo del multiplicador es **0**; no hay que sumar el multiplicando. | | |
| Correr a la derecha Multiplicador y Resultado: | 0000 | 0100 1011 |

Note que el resultado de la multiplicación puede ocupar $2*n$ bits (01001011B, en este caso).

En el caso de la división, también se usan los métodos de aritmética en otras bases:

- Se reservan $2*n$ bits. El dividendo va en los $n$ menos significativos, el resto se ponen en cero.
- Se corren a la izquierda los $2*n$ bits del dividendo. El bit más significativo se pierde, y un cero queda como el menos significativo.
- Se verifica si el divisor es menor o igual que los $n$ bits más significativos del dividendo, en cuyo caso se resta de los mismos.
- El resultado se corre a la izquierda; el bit menos significativo queda en 1, si se pudo restar el divisor, y en cero, si no.
- Se regresa al paso 2 —Hay que iterar $n$ veces—.

A continuación, un ejemplo con 4 bits (divisor = 0101 y dividendo = 1111):

| Comentario | Resultado | Dividendo |
|---|---|---|
| | 0000 | 0000 1111 |
| Correr a la izquierda el dividendo | | 0001 1110 |
| ¿Divisor ≤ dividendo? No, luego no se resta el divisor | | |
| Correr a la izquierda el resultado, | | |
| poner un cero como bit menos significativo | 0000 | |
| Correr a la izquierda el dividendo: | | 0011 1100 |
| ¿Divisor ≤ dividendo? No, luego no se resta el divisor | | |
| Correr a la izquierda el resultado, | | |
| poner un cero como bit menos significativo | 0000 | |

Correr a la izquierda el dividendo:
¿Divisor ≤ dividendo? Sí, luego se resta el divisor

| | | | |
|---|---|---|---|
| | | 0111 | 1000 |
| | - | 0101 | |
| | | 0010 | 1000 |

Correr a la izquierda el resultado,
poner **1** como bit menos significativo

| |
|---|
| 0000 |
| 0001 |

Correr a la izquierda el dividendo:
El divisor es menor o igual que el dividendo,
se resta el divisor del dividendo:

| | | | |
|---|---|---|---|
| | | 0101 | 0000 |
| | - | 0101 | |
| | | 0000 | 0000 |

Correr a la izquierda el resultado,
poner **1** como bit menos significativo

| |
|---|
| 0010 |
| 0011 |

Incidentalmente, en los $n$ bits más significativos del dividendo queda el residuo de la división —en este caso es cero—.

***Inconvenientes***

El hecho de tener una cantidad limitada de bits introduce algunas sutilezas en la aritmética, veamos cuáles.

Primero que todo, cuando se suman dos números un bit puede "salir" por la izquierda. Por ejemplo, trabajando con 6 bits:

| | | |
|---|---|---|
| | | 111111 |
| | + | 111111 |
| = | 1 | 111110 |

Hay dos posibilidades: si se interpretan los números como representados en complemento a dos, la operación es correcta sin importar el dígito que salió (-1 + -1 = -2); pero, si se interpretan como número sin signo, el resultado es erróneo ya que se pierde el dígito más significativo (¿63 + 63 = 62?).

Un caso particular es cuando se han representado los números usando varias palabras; entonces se debe sumar el bit que sale a la segunda parte del número:

| | | | |
|---|---|---|---|
| | | 1 | |
| | | 000001 | 111111 |
| | + | 000000 | 111111 |
| = | | 000010 | 111110 |

El bit que "sale" se llama el bit de acarreo (*carry*). Cuando se hace una operación aritmética, el procesador recuerda este bit para permitir al programador mirarlo y tomar decisiones según el valor que tenga.

Veamos otros dos casos:

| | | |
|---|---|---|
| | | 000001 |
| | + | 011111 |
| = | 0 | 100000 |

| | | |
|---|---|---|
| | | 000001 |
| | + | 011111 |
| = | 0 | 100000 |

En ambos casos, si interpretamos los números como complemento a dos, el resultado es incorrecto; en el primer caso, la suma de dos negativos da un positivo;

en el segundo caso, la suma de dos positivos da un negativo. Esto ocurre porque hay un máximo para los valores representables. Aunque cada sumando es menor que el máximo, la suma lo sobrepasa. En ese caso, decimos que ha ocurrido un desbordamiento (*overflow*); se ha rebasado la capacidad máxima de la suma.

Cuando ocurre un desbordamiento, el acarreo que sale de la penúltima columna y el que sale de la última columna (*carry*) son diferentes. Podemos definir un *bit de desborde* como:

Bit Desborde = último acarreo O-excluyente penúltimo acarreo

El bit vale uno si ha ocurrido un desborde, cero de lo contrario. Puede verificarlo en los dos ejemplos anteriores. Este bit también es memorizado por el procesador.

Podemos concluir que, en el caso de los números naturales, el acarreo indica que hay un desbordamiento, el resultado es incorrecto. En el caso de los enteros, usando complemento a dos, el o-excluyente de los dos últimos acarreos indica si hay error o no.

### *Otras operaciones*

El hecho de trabajar con un número limitado de bits, nos permite definir otras operaciones:

Corrimiento (*shift*): consiste en desplazar los bits dentro de una palabra hacia la izquierda o hacia la derecha. Ejemplo:

| | |
|---|---|
| Tenemos | 10110010 |
| Un corrimiento de 2 posiciones a la izquierda produce: | 11001000 |
| Un corrimiento de 2 posiciones a la derecha produce: | 00101100 |

Rotación: consiste de desplazar circularmente los bits dentro de la palabra. Es decir, los bits que salen por la izquierda vuelven a entrar por la derecha, y viceversa cuando se hace la rotación contraria. Ejemplo:

| | |
|---|---|
| Tenemos | 10110010 |
| Una rotación de 2 posiciones a la izquierda produce: | 11001010 |
| Una rotación de 2 posiciones a la derecha produce: | 10101100 |

Ocasionalmente, las rotaciones y los corrimientos se pueden hacer considerando el bit de acarreo como si hiciera parte de la palabra:

| | | |
|---|---|---|
| Tenemos | 1 | 10110010 |
| Una rotación de 2 posiciones a la izquierda produce: | 0 | 11001011 |
| Una rotación de 2 posiciones a la derecha produce: | 1 | 01101100 |

Note que estas operaciones son necesarias para implantar los métodos antes descritos de multiplicación y división.

## 3.2 LOS CARACTERES

No hay ningún método "natural" para representar los caracteres, lo más que se puede hacer es dejarlos ordenados... de alguna manera. La solución que se ha tomado es asignarle un valor, más o menos arbitrario, a cada carácter.

Ha habido diversos códigos de caracteres. Entre los más antiguos se cuentan el código Morse y el código de Baudot; el primero de longitud variable y el segundo de 5 bits. Otros códigos, más recientes, son: EBCDIC, ASCII y Unicode.

### ¿Qué es un carácter?

En una primera aproximación, puede parecer sencillo decir qué es un carácter, pero, al analizar atentamente el concepto —especialmente en un contexto multicultural—, se torna menos evidente.

Empecemos diciendo que es un concepto que va más allá del concepto de “sonido”, de “letra” y de “fuente tipográfica”. Los caracteres son entidades abstractas designadas por representaciones gráficas.

En primer lugar, son signos, y, como dice el lingüista suizo Ferdinand de Saussure —fundador de la semiología—, el signo está compuesto de dos partes: el significado (lo denotado) y el significante (lo que lo denota).

Ahora bien, el significado puede ser más o menos abstracto: si decimos “Napoleón Bonaparte” o “Roma”, nos referimos a entidades concretas con existencia en el mundo real. En cambio, si decimos “persona” nos referimos a una abstracción: no existe una cierta entidad “persona”; es un concepto generado por cada cual después de conocer una variedad de personas concretas. Los caracteres son todavía más abstractos, porque no son entidades con existencia física en el mundo, ni abstracciones creadas sobre las mismas, sino construcciones humanas arbitrarias que no corresponden a una realidad concreta: son los que queremos que sean.[2]

En la representación informática de los caracteres, importa el significado; en los despliegues gráficos, el significante. Así, en una codificación de caracteres, tenemos un solo código para la a, pero, disponemos de varias representaciones gráficas alternativas, dadas por diversas fuentes tipográficas ( *a*, a, a ) y formatos ( **a**, *a* , ***a*** ), los cuales no se incluyen en la codificación.

Sin embargo, ciertas variaciones en la forma del significante son importantes: considere la a y la A ¿son diferentes las minúsculas y las mayúsculas? ¿por qué son caracteres diferentes? ¿en qué difiere su significado? La respuesta es decepcionantemente simple: son diferentes porque queremos que sean diferentes.[3]

Este tipo de variaciones también se presentan en otros idiomas, aunque por motivos diferentes; en el alfabeto griego, por ejemplo, existen dos sigmas: una se usa en medio de una palabra ( σ ) y la otra, al final de palabra ( ς ). En el alfabeto

---

[2] ¿Acaso son imprescindibles símbolos como: &, #, @? Es más, ¿qué significan? ¿Podríamos prescindir de la c? En español, se podría reemplazar su uso con k y z.

[3] Si quisiéramos, y nos pusiéramos de acuerdo, ¡podríamos inventarnos la “a mediúscula” y asignarle alguna función en la escritura!

árabe hay algo similar: hay varias formas para un carácter dependiendo de su contexto. La pregunta es: ¿son caracteres distintos o son variaciones del mismo carácter de cuya representación visual se deben encargar niveles superiores de interpretación?

Hay otro tipo de variaciones que vienen dadas por los signos diacríticos: ¿es á diferente de a? De cierta forma sí, de cierta forma no. Quizás la mejor forma de describirlo es que se trata de la unión de dos símbolos, pero sin que pierdan su identidad: la a denota una cosa y la tilde otra. Pero en otros casos es diferente: para los suecos å es una letra, no una a con un signo diacrítico; en español, ñ es un solo símbolo, no una n con tilde.

Por supuesto, existe el problema contrario: si dos caracteres tienen igual significante, ¿son el mismo? ¿Es la letra kappa ( κ ) una k? ¿Es épsilon ( ε ) una e? ¿Es la pe del alfabeto cirílico ( п ) igual a pi ( π )? ¿Es el *er* del cirílico ( р ) igual a la p latina? Estas preguntas no se pueden responder automáticamente: dependen de los leguajes en cuestión, de su origen y de sus usos. En Unicode, por ejemplo, la respuesta a las anteriores preguntas es negativa: son letras que pertenecen a alfabetos diferentes, así sean parecidas en algunos aspectos, luego son caracteres diferentes. Pero esta argumentación no se puede llevar al extremo: ¿es diferente la e del francés de la e del inglés, o la del español o la del alemán? Los respectivos alfabetos tienen pequeñas diferencias ( ç , ñ , ß ) pero pueden ser considerados, en esencia, el mismo: alfabeto latino; en consecuencia, probablemente no vale la pena replicar símbolos como la e.

La problemática de la representación de caracteres no termina aquí, y, de hecho, se vuelve más compleja al incluir concepciones diferentes de la escritura —como la escritura ideográfica—. Para concluir, digamos que la representación de caracteres plantea muchas preguntas, y que no hay respuestas absolutas ni universales. Muchas decisiones se deben tomar por análisis casuístico, y dependen de características gramaticales, fonéticas y culturales de los idiomas en cuestión.

### Código ASCII

Una representación muy usada es el código ASCII (*American Standard Code for Information Interchange*). En este código, cada carácter se almacena en un byte;[4] con los 7 bits menos significativos se asignan códigos —de 0 a 127— a los diferentes caracteres, el bit más significativo siempre es cero (ver el código en la figura 3.3).

Para encontrar el código de un carácter en la tabla, basta con buscar el carácter, tomar el dígito de la columna y adjuntarle el dígito de la fila. De esta manera podemos ver que el código de "A" es `41H`, ó `01000001B`.

Las siglas a la izquierda de la tabla (NUL, SOH etc.) son los llamados *caracteres de control*. Son utilizados para enviar mensajes de control —por ejemplo, cuando el

[4] Dado que varios códigos representan los caracteres en 8 bits, se decidió darle un nombre a los grupos de 8 bits: *Byte* (*BinarY TErm*). Ha habido computadores que tenían "bytes" con un número de bits diferente de 8, pero el estándar actual es 8 bits. Esto explica la existencia y uso de la palabra “octeto” —incluso en inglés— como una forma más precisa de referirse a un grupo de 8 bits.

procesador se comunica con otros dispositivos—. Estos caracteres no son imprimibles, es decir que no dibujan nada en una pantalla o en una impresora.

El ASCII no es tan arbitrario, fue diseñado con ciertas características en mente:

- Si se hace el O lógico del código de una letra mayúscula con `20H`, el resultado es el código de la letra en minúscula.
- Si se efectúa un Y lógico entre una minúscula y `DFH`, se obtiene la mayúscula correspondiente.
- Al hacer el Y lógico de una letra cualquiera con `1FH`, se obtiene el carácter de control correspondiente. Eso es lo que ocurre cuando uno teclea CONTROL y una letra en un computador. En consecuencia, cuando se teclea CONTROL-C en realidad se está mandando un carácter de control al computador: ETX (*End of TeXt*).
- El Y de un carácter-dígito ("0" a "9") con 0FH, da como resultado el valor numérico del carácter. Por ejemplo, el código del carácter "4" es `34H`, después del Y el resultado es `04H`. Lo contrario, pasar del valor al carácter, se logra haciendo un O lógico con `30H`.

### Código EBCDIC

Otra codificación para caracteres, un poco más patológica que la ASCII, es el código EBCDIC (*Extended Binary Coded Decimal Interchange Code*). Este código de 8 bits,

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [ | k | { |
| C | FF | FS | , | < | L | \ | l | \| |
| D | CR | GS | - | = | M | ] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | _ | o | DEL |

**Fig. 3.3. Código ASCII**

desarrollado por IBM, se desprendió de uno anterior de 6 bits BCDIC (*Binary Coded Decimal Interchange Code*).

Hoy en día, EBCDIC no es muy popular. Le faltan algunas propiedades del ASCII, y no tiene características diferentes, exceptuando una cierta compatibilidad con las tarjetas perforadas (¿Las conoce? Unas tarjetas de cartulina que se usaban, hace tiempo, para programar).

### Unicode

Tanto ASCII como EBCDIC tienen el inconveniente de no haber sido diseñados para soportar múltiples lenguas; esto se convierte en un problema serio cuando se desea desarrollar software que pueda operar en diversos idiomas.

Como consecuencia de lo anterior, se desarrollaron formas diferentes, e incompatibles, para representar los sistemas de escritura no cubiertos por los códigos tradicionales. Esta confusa situación condujo a la idea de definir un código estándar con soporte para los diversos idiomas del mundo: Unicode. Unicode incluye sistemas de escritura alfabéticos (como el latino y el cirílico), silábicos (como el japonés y el coreano) e ideográficos (como el japonés y el chino); también tiene soporte para conjuntos de caracteres históricos.

Unicode es administrado por el Consorcio Unicode, pero este mantiene relaciones con la ISO (*International Organization for Standardization*), por lo cual Unicode es compatible con el código ISO/IEC 10646: "*Information technology — Universal Multiple-Octet Coded Character Set* (UCS)".

Unicode es un sistema extenso. La descripción que se da a continuación está basada en la versión 4.0 del estándar, y solo puede ser considerada introductoria. Aun así, es necesario tener claros algunos conceptos básicos antes de entrar en el detalle.

#### *Conceptos básicos*

En primer lugar, Unicode diferencia entre carácter (el significado) y glifo (el significante; la representación gráfica del carácter). Unicode no se pronuncia sobre los glifos, solo sobre los caracteres.[5]

Unicode tiene composición estática y dinámica de caracteres. En la estática, hay un solo carácter que representa completamente el símbolo; en la dinámica, se usan varios caracteres que, en conjunto, definen el símbolo (esto se usa, por ejemplo, para las formas acentuadas de las vocales).[6]

En Unicode se diferencia entre los siguientes conceptos:

- Repertorio de caracteres abstractos: es el conjunto de caracteres que se desea representar.

---

[5] Aunque el estándar incluye el glifo como parte de la descripción, esto solo se hace a título informativo, no normativo.

[6] En ocasiones, un mismo símbolo se puede representar por dos secuencias distintas. Esto se hizo para tener compatibilidad con otros códigos de caracteres, pero introduce el problema de tener varias representaciones para una misma entidad, y la necesidad de definir equivalencias.

- Conjunto de caracteres codificados: consiste en la asignación de números naturales a los caracteres abstractos. Estos números se llaman "puntos de codificación". Como se dijo anteriormente, un carácter abstracto puede tener asignados varios códigos; en otros casos, un carácter abstracto puede ser representado por varios caracteres codificados (como es el caso de las vocales con acento: [á] = [a] + [´] ).
- Forma de codificación de los caracteres: es la forma como se codifica el punto de codificación como un número entero en el computador.

En resumen: Unicode diferencia entre el carácter en sí, su código y la forma de representar dicho código en el computador. Esto último implica que Unicode puede tener diversas representaciones donde el número de bits varía.[7]

### *Codificación*

El estándar Unicode, en su versión 4.0, codifica los caracteres en el rango de 0 a `10FFFFH`. Por comodidad, se acostumbra dividir este rango en planos; cada plano es un espacio de 64K puntos de codificación.

En general, se intenta que los caracteres relacionados por razones lingüísticas queden adyacentes.

Los códigos se expresan usando la notación U+punto de codificación, donde el punto de codificación se escribe en hexadecimal. Por ejemplo `U+0061` es el código de la a minúscula (`61H`, como en ASCII). Cada carácter también tiene asociado un nombre oficial, así que la identificación completa es del estilo: `U+0061` LATIN SMALL LETTER A.

Los primeros 128 puntos de codificación corresponden a los caracteres del ASCII.

Note que, hasta este punto, solo tenemos que a cada carácter abstracto se le asigna un número; falta tratar el aspecto de cómo se representan estos números en el computador. Unicode tiene tres formas de codificación: UTF-8, UTF-16 y UTF-32 (UTF significa *Unicode Transformation Format*).

- UTF-32: es el formato más sencillo; cada punto de codificación se representa usando 32 bits. Puesto que el espacio de codificación de Unicode va hasta 10FFFFH, 21 bits, no hay problema en representarlos usando 32 bits —de hecho, sobran 11 bits—.[8]
- UTF-16: la unidad de codificación es de 16 bits, lo cual es suficiente para los puntos de codificación en el rango de 0 hasta FFFFH; en cuanto a los puntos de codificación en el rango de 10000H a 10FFFFH, se usan parejas de unidades de codificación. Para poder hacer esto, Unicode reservó las unidades de codificación entre D800H y DFFFH: ningún carácter puede tener

[7] Debido a versiones anteriores, existe la idea, errónea, de que Unicode es un código de 16 bits.

[8] Unicode entra en más detalle, en particular, entra en consideraciones de representaciones *little endian* y *big endian*; nosotros dejaremos de lado esos aspectos en esta discusión.

esos códigos. Los puntos de codificación superiores a FFFFH, se representan de la siguiente manera:

| | | |
|---|---|---|
| UTF-32 | 00000000000ccccc | dddddddddddddddd |
| UTF-16 | 110110aaaaddddddd | 110111dddddddddd |

Donde aaaa = ccccc – 1. Note que el código **110110** es el indicador de que se trata de la primera mitad y **110111**, el de la segunda mitad. También note que D800H = **110110**0000000000B y que DFFFH = **110111**1111111111B, por ende, los códigos UTF-16 no se pueden confundir con ningún otro carácter.

- UTF-**8**: la unidad de codificación es de 8 bits. Todos los caracteres ASCII se pueden representar usando un byte; en cuanto a los otros, se usan entre 2 y 4 unidades de codificación, según se muestra en la siguiente tabla:

| | UTF-8 | | | |
|---|---|---|---|---|
| UTF-32 | 1er byte | 2º byte | 3er byte | 4º byte |
| 0000000000000000000000dddddddd | 0ddddddd | | | |
| 00000000000000000000ccccddddddd | 110ccccc | 10dddddd | | |
| 0000000000000000bbbbccccccddddddd | 1110bbbb | 10cccccc | 10dddddd | |
| 00000000000aaabbbbbbccccccddddddd | 11110aaa | 10bbbbbb | 10cccccc | 10dddddd |

Note que el comienzo del primer byte indica cuántos bytes componen el código; para ser precisos, el número de unos antes del primer cero indica la cantidad de bytes usados para codificar.

Recuerde: una serie de dígitos binarios no quiere decir nada sin una interpretación del programador: 56H puede ser la letra "V" o el número 86 decimal. El computador no diferencia, es el programador quien lo hace.

Las convenciones importantes en la representación de caracteres son: el número de bits utilizados en la representación y las tablas de codificación. Estas últimas son de importancia capital; dado que la codificación es arbitraria, si no se conocen las tablas, es imposible interpretar.

## 3.3 Los números decimales

Tenemos un problema de comunicación con los computadores, ya que ellos trabajan en binario y nosotros en decimal. Esta diferencia implica un proceso de traducción: les damos los números en decimal, ellos los convierten a binario y realizan las operaciones correspondientes; al terminar sus cálculos, traducen el resultado a decimal. Es decir, cada entrada o salida implica una traducción, por lo tanto, implica más trabajo para el procesador.

Por esta razón, algunos procesadores soportan la aritmética decimal. La ventaja es que simplifica el proceso de traducción; la desventaja, que las operaciones son más demoradas.

A continuación, vamos a mostrar cómo se hace la traducción binario-decimal y, en seguida, vamos a estudiar la aritmética decimal en el procesador.

Cuando se digita un número en el teclado, en realidad se están enviando los códigos ASCII de los dígitos al procesador. Supongamos que se teclea el número 301, es decir, se envían los caracteres "3","0" y "1"; el procesador recibe los códigos ASCII 33H, 30H y 31H. Lo primero que debe hacer es obtener sus valores numéricos; para esto, como vimos antes, se hace el Y de cada ASCII con 0FH. Una vez hecha esto, se obtiene 03H, 00H y 01H. Ya se tiene el valor de los dígitos decimales que componen el número; para convertirlos a binario debemos utilizar la función de interpretación que vimos en la sección de cambio de base:

$\text{Entero}_{10}(301) = (\mathbf{3}*10+\mathbf{0})*10 + \mathbf{1}$
o en hexadecimal: $(\mathbf{03H}*0AH+\mathbf{00H})*0AH + \mathbf{01H}$

Dado que el procesador realiza los cálculos en binario, el resultado de evaluar la expresión anterior es 301 en binario. La sumatoria que define el valor de una representación sirve como método general de cambio de base, siempre y cuando se domine la aritmética de la base en cuestión, y los computadores saben hacer aritmética binaria —es su base "natural", como para nosotros lo es la base 10—.

Cuando el computador va a presentarle los resultados al usuario, debe traducirlos de binario a decimal. Para esto utiliza el algoritmo de divisiones sucesivas: divide el número por 10 —el procesador hace la operación en binario; no importa: la aritmética siempre es la misma—. El módulo de la división es un valor entre 0 y 9, le hace un O lógico con 30H, con lo cual queda convertido en carácter. Se repite la operación con el cociente de la división hasta que este sea 0. Los caracteres obtenidos representan el número en la base 10.

### La adición en BCD

Para evitar el proceso de traducción descrito en la sección anterior, se utiliza la aritmética decimal. Los números se codifican usando una representación llamada BCD ("Binary Coded Decimal"). La idea es representar cada dígito decimal en binario, usando 4 bits, como se puede ver en el ejemplo siguiente:

1823 se representaría por: `0001 1000 0010 0011`

El valor de cada grupo de cuatro bits representa el dígito decimal que corresponde. Observe que, usando esta representación, la traducción de los dígitos ASCII se reduce a hacer el Y del código ASCII con 0FH.

El problema del BCD es que el procesador no sabe realizar cálculos decimales; los circuitos están hechos para operar con números binarios. Por ejemplo, si usamos la suma binaria, el resultado no siempre es correcto. Veamos dos casos de números BCD adicionados usando suma binaria:

```
   0001  0010  (12)        0010  0111  (27)
+  0011  0101  (35)    +   0100  0100  (44)
-------------------     -------------------
   0100  0111  (47)        0110  1011  (6?)
```

En el primer caso, sumamos 12 y 35, y obtenemos 47, lo cual es correcto. En el segundo caso, sumamos 27 y 44 y obtenemos 6? —escribimos '?' porque 1011 no representa nada en BCD—, es decir, es un resultado incorrecto.

Es necesario desarrollar un método para evaluar las sumas BCD usando la aritmética binaria disponible en el computador. Por supuesto, podríamos usar las técnicas descritas en la sección de aritmética en otras bases; es decir, sumar cada pareja de dígitos BCD, y sacar el cociente y el residuo para obtener el resultado y el acarreo. Esta técnica no es conveniente porque implica muchos cálculos y vuelve muy lenta la aritmética decimal. Es mejor tratar de aprovechar al máximo las operaciones binarias del computador ajustando los resultados para que coincidan con el decimal correspondiente.

El problema reside en que cada grupo de 4 bits se puede ver como un sumador módulo 16, y nosotros pretendemos hacerlo funcionar como un sumador módulo 10. Lógicamente no funciona puesto que el computador trabaja en binario y no en decimal.

Vamos a intentar el siguiente método: sumar los números en binario y después corregir los dígitos que hayan quedado mal. De esta manera aprovechamos los circuitos sumadores del computador y obtenemos una aritmética más rápida que si la hubiéramos simulado.

La idea es la siguiente, si la suma de dos dígitos BCD es menor que 10, el resultado es correcto. Pero si la suma es mayor que 10, es un resultado incorrecto, ya que no hay dígitos mayores que 10 en la base 10, y debe ser corregido de alguna manera.

Para corregir los resultados erróneos, tenemos que hacer funcionar el sumador base 16 como un sumador base 10. Para esto, hay que saltarse 6 posiciones cuando el resultado sea mayor que 9. Es decir, hay que obligar al sumador a pasar del 9 al 0. En efecto, si el resultado de la suma es AH (10), al adicionar 6H nos da 10H el cual, interpretado como BCD, es justamente el resultado. De la misma manera, si el resultado es BH (11), al sumarle 6H el resultado es 11H, de nuevo obtenemos la respuesta correcta en BCD.

Tenemos entonces que el número 6 funciona como un factor de corrección; cuando la suma de dos dígitos da un resultado superior a 10, basta con sumar 6 para obtener el resultado debido.

Ahora bien, una suma mayor que 10 se puede manifestar de dos maneras: la primera es cuando el número que queda en los 4 bits es mayor que 10, caso 6+7; la segunda es cuando se produce un acarreo fuera de los 4 bits, caso 9+9.

A partir de las dos observaciones anteriores, podemos desarrollar un algoritmo para sumar BCD: se suman los dígitos BCD en binario. Después de mira cada grupo de 4 bits, si el resultado es mayor que 10 o se produce un acarreo, se le suma 6 al dígito en cuestión.

Para simplificar el proceso, se pueden sumar los dígitos BCD de a dos en dos (8 bits). Para esto, se debe recordar tanto el acarreo final —producido por los últimos 4 bits— como el acarreo de la mitad —producido por los primeros 4 bits—. Este último es el *bit de acarreo intermedio*. Sólo resta aplicar el método anterior a cada uno de los dos dígitos. Veamos un ejemplo:

```
(0)   (1)                        (0)   (0)
      0001  1001  (19)                 0010  0111  (27)
  +   0011  1000  (38)             +   0100  0100  (44)
      0101  0001  (51)                 0110  1011  (6?)
  +   0000  0110  (06)             +   0000  0110  (06)
      0101  0111  (57)                 0111  0001  (71)
```

En el primer caso, al sumar 8H y 9H da 1, y hay un acarreo de 1. Por lo tanto, hay que sumarle 6 para corregir el dígito; el segundo dígito no hay que corregirlo puesto que no da mayor que 10 ni hay acarreo. En el segundo caso, al sumar 4H y 7H da BH, es mayor que 10, luego hay que corregirlo.

En resumen, el proceso de ajuste es el siguiente:

- Se suman los dos operandos BCD.
- Si el primer dígito está mal —suma mayor que 10—, se suma 06H al resultado anterior.
- Si cualquiera de las dos primeras sumas causó un error en el segundo dígito, se suma 60H al resultado anterior.
- Si cualquiera de las tres sumas produce un acarreo —en el bit de "carry", no en el de "carry" intermedio—, se debe anotar un acarreo final.

Cuando se tienen más de 2 dígitos BCD, se aplica el método anterior a cada pareja. Lo único que se debe modificar es adicionar el acarreo de una suma a la siguiente.

**La resta en BCD**

Los números negativos en BCD se representan con técnicas similares a las binarias. Suponiendo que se dispone de $n$ dígitos BCD, se puede utilizar signo y magnitud: el dígito más significativo guarda el signo, 0H si es positivo y 9H si es negativo o cualquier otra convención. O, si se prefiere, se pueden representar en complemento a 10, es decir $-x$ se representa por $10^n-x$.

En primer lugar, vamos a estudiar cómo calcular el complemento a 10 de un número BCD. Si el número BCD es $(d_n \ldots d_0)$ la operación decimal que se debe efectuar es:

$$\begin{array}{llll} 1 & 0 & \ldots & 0 \\ - & d_n & \ldots & d_0 \\ \hline & r_n & \ldots & r_0 \end{array}$$

Donde $r_n \ldots r_0$ es el complemento a 10 del número BCD. Lo anterior se puede expresar dígito a dígito por medio de las siguientes ecuaciones:

$10 - d_0 = r_0$, para el dígito menos significativo.
$9 - d_i = r_i$, para los otros dígitos.

Sin embargo, se tiene el inconveniente de siempre; el procesador calcula en binario y no en decimal. Se puede hacer lo mismo que en el caso de la suma: hacer la operación en binario, y calcular el complemento a 2; lo cual, mirado con los "lentes hexadecimales", quiere decir que la operación realizada por el procesador es:

10 ... 0H
$-d_n \ldots d_0$H

Lo mismo que en el caso de la suma, el resultado de esta operación no es el buscado. Efectivamente, si expresamos la operación dígito a dígito las operaciones son:

16 - $d_0$, para el dígito menos significativo (recuerde que 10H = 16).
15 - $d_i$, para los otros dígitos.

Lo cual, claramente, es diferente de las ecuaciones que se utilizan en el caso del complemento a 10. Pero podemos expresar las ecuaciones anteriores de una manera alterna:

10 - $d_0$ + 6, para el dígito menos significativo.
9 - $d_i$ + 6, para los otros dígitos.

Hemos obtenido las mismas expresiones del complemento a 10, sólo que con un 6 adicionado. Ahora la solución es clara: se calcula el complemento a 2 y se le resta 6 a cada dígito (hexadecimal) del resultado. A continuación, mostramos un ejemplo de este proceso; aunque las operaciones son binarias las vamos a expresar en hexadecimal por comodidad. Recuerde: para nosotros la base 16 es únicamente una forma de facilitar la escritura de números binarios.

Se tiene el número BCD `0295` y se quiere hallar su complemento a 10.
En primer lugar se calcula el complemento a dos:
`10000H - 0295H = FD6BH`
Al resultado se le resta 6 a cada dígito:
`FD6BH - 6666H = 9705H`
Y, en decimal, se cumple que `10000 - 0295 = 9705`. ¡Funciona!

Pero —siempre hay un pero— el método anterior falla cuando el número BCD acaba en uno o más ceros. En ese caso, 0 - 0 = 0, el dígito no tiene un 6 sumado. Por ejemplo:

Se tiene el número BCD 2090 y se quiere hallar su complemento a 10.
Complemento a dos: `10000H - 2090H = DF70H`
Restarle 6 a cada dígito: `DF70H - 6666H = 790AH`
Debería dar: `10000 - 2090 = 7910`. Casi, pero no.

Debemos generalizar el método para que incluya este caso; para esto, notamos lo siguiente: cuando se calcula el complemento a dos, se niega el número y se le suma 1. La negación convierte los `0H` en `FH`, y, al sumar 1, los `FH` iniciales se convierten otra vez en cero pero con un acarreo de 1. Note que sólo a los ceros menos significativos les ocurre esto.

El algoritmo se convierte en:

- Hallar el complemento a 2 del número BCD.

- A los dígitos que NO produjeron acarreo se les resta 6; los que produjeron acarreo no se modifican

Veamos ahora la manera de calcular las restas en BCD. Una posibilidad es calcular el complemento a diez del substraendo y adicionarlo al minuendo. Alternativamente, la operación se puede efectuar de manera directa, con lo cual se puede aprovechar la operación de resta del procesador. Dicha operación, en binario, es realizada por el procesador de la siguiente manera:

$$a - b = a + \bar{b} + 1$$

Como vimos anteriormente, al realizar una suma o al calcular el complemento hay que corregir el resultado. Al hacer la resta, se está haciendo implícitamente una suma y un complemento, cada una de estas operaciones introduce un error: el primero, que el complemento a dos da como resultado el complemento a 10 pero sumándole 6 a los dígitos; el segundo, que, al realizar la suma, algunos dígitos son mayores que 9.

Sin embargo, los dos errores se compensan de una manera interesante; hay que sumar 6 a los dígitos erróneos de la suma, pero como también se ha calculado el complemento a dos —no a 10— ya se ha sumado un 6. En consecuencia, los dígitos erróneos se han corregido automáticamente. Un problema que persiste es que se han alterado los dígitos correctos de la suma, puesto que a ellos también se les ha sumado 6 al calcular el complemento a 2.

En resumen, al efectuar la suma entre $a$ y el complemento de $b$, los números que deberían aparecer mal en la suma BCD han sido corregidos, pero, por otro lado, los que deberían aparecer bien han sido alterados sumándoles 6. El resultado de todo esto es que, en el caso de la resta, tenemos que invertir el algoritmo de corrección usado en el caso de la suma: si se produce un acarreo fuera de los cuatro bits que representan al dígito BCD, el resultado es correcto; si no se produce el acarreo, hay que restar 6 al dígito. Veamos un ejemplo:

Se quiere restar 27 de 43 usando BCD.

```
(1)  (10)
     0100  0011  (43)
     1101  1000  (negación de 27)
   + 0000  0001  (+1)
     ----  ----
     0001  1100  (1?)
   - 0000  0110  (06)
     ----  ----
     0001  0110  (16)
```

Algunos computadores tienen instrucciones para realizar las diferentes operaciones BCD haciendo las correcciones de manera automática. En otros, el programador debe realizar las operaciones en binario, después de lo cual utiliza una instrucción especial que se encarga de corregir el resultado. Otros computadores no tienen aritmética decimal y hay que desarrollarla usando métodos similares a los que se mostraron en esta sección.

## 3.4 LOS NÚMEROS DE PUNTO FLOTANTE

En ocasiones, es necesario trabajar con números muy grandes, muy pequeños o con parte fraccionaria. Los enteros no permiten manejar estos casos; por lo cual se desarrolló otro formato de número: el punto flotante. A veces, estos números son llamados "reales"; estrictamente hablando, no son reales, pero se les suele llamar así, especialmente en los lenguajes de alto nivel.

Esta sección se divide en cuatro partes: la primera describe nuestro, ahora típico, problema de conversión: humanos – procesador; base 10 - base 2. La segunda parte está dedicada a la descripción del formato utilizado para la representación de números de punto flotante. La tercera presenta la forma de efectuar las operaciones aritméticas usando esta representación. La última parte presenta, en detalle, una representación de números de punto flotante.

### Conversión a binario

Nosotros utilizamos números de punto flotante de la forma: $a*10^b$. Los computadores, por su lado, utilizan $a'*2^{b'}$. Nuestro problema inmediato es traducir de una forma a la otra. Partiendo de la representación decimal, $a*10^b$, vamos a construir la representación binaria.

Empecemos por resolver la ecuación: $10^b = 2^c$

$$\ln(10^b) = \ln(2^c) \quad \Rightarrow$$
$$b*\ln(10) = c*\ln(2) \quad \Rightarrow$$
$$c = (\ln(10) / \ln(2))*b = 3.32193*b$$

Por lo tanto: $a*10^b = a*2^{3.32193*b}$. El exponente se puede separar en dos partes: una parte entera ($x$) y una parte fraccionaria ($y$), es decir:

$$3.32193*b = x + y,\ x \in \mathbb{N} \text{ y } |y| < 1$$

Podemos reescribir la expresión como:

$$a*10^b = a*2^{x+y} = (a*2^y)*2^x$$

Todo lo que resta por hacer, es pasar x y la expresión entre paréntesis a binario.

Ejemplo, pasar $0.153*10^{15}$ a binario:

$$0.153*10^{15} = 0.153*2^{15*3.32193} = 0.153*2^{49.82895} = (0.153*2^{0.82895})*2^{49} = 0.27179*2^{49}$$

Convirtiendo a binario obtenemos, aproximadamente, $0.010001011*10^{110001}$B.

El paso de binario a decimal es parecido, solo cambia el factor de multiplicación; en vez de 3.32193 se usa 0.30103 (el inverso). Por ejemplo, tomemos el binario del ejemplo pasado reconvirtiéndolo a decimal. Habrá una diferencia con el original, introducida por las aproximaciones que se hacen durante los cálculos:

$$0.27148*2^{49} = 0.27148*10^{49*0.30103} = 0.27148*10^{14.75047} = (0.27148*10^{0.75047})*10^{14} = 1.5283*10^{14}$$

### Representación de números de punto flotante

Los números de punto flotante nos dan la posibilidad de trabajar con valores más grandes que los permitidos por los enteros, pero se pierde precisión. Lo anterior puesto que los exponentes son tan grandes que no se pueden guardar todos los dígitos del número. Dependiendo del caso, la precisión puede ser, o no, suficiente.

En los lenguajes de alto nivel, los números de punto flotante se escriben con notaciones del tipo: 3.726 E -4, lo cual quiere decir $3.726*10^{-4}$. Podemos observar que están compuestos por tres partes: la mantisa (en el ejemplo es 3.726), el exponente (-4) y la base de exponenciación. Esta última es implícita y en general es 10. Pasemos a ver el formato usado en binario.

Se encuentran diversos formatos para representar los números de punto flotante, pero todos siguen el mismo esquema. Por esto, describiremos la representación en términos generales. Más adelante se encuentra la descripción de un método en concreto: el estándar 754 de la IEEE para la aritmética de punto flotante.

En primer lugar, hay que decidir cómo se van a representar los tres componentes del número de punto flotante. La mantisa y el exponente se representan explícitamente por medio de dos números binarios. La base de exponenciación no se representa explícitamente, el procesador trabaja con una base definida al diseñarlo. Esta base puede ser 2 —en lo que sigue, supondremos que ese es el caso—, pero no es obligatorio; algunos computadores han usado 8 ó 16.

Para representarlos se destina una cierta cantidad $m$ de bits para codificar el número de punto flotante; de estos, $l$ bits se destinan para el exponente, $n$ para representar la mantisa y uno para el signo de la mantisa (fig. 3.4)

El exponente no presenta ningún problema. Es simplemente un entero binario sobre $l$ bits; puede ser representado en signo y magnitud, complemento a 2 o algún otro método. Lo importante es que la representación permita los números negativos, ya que el exponente puede ser negativo o positivo.

La mantisa es un poco más delicada, puesto que puede tener una parte fraccionaria. Su representación se puede hacer de diversas maneras. Por ejemplo, se puede suponer que la mantisa siempre es fraccionaria, es decir, que el número binario almacenado tiene un punto a la izquierda. Además, se representa en valor absoluto; el signo se almacena independientemente: 1 si es negativo y 0 si es positivo. Veamos un ejemplo con las convenciones $l = 6$ y $n = 8$:

| 011010 | 0 | 11011101 |
|---|---|---|

representa a: $+0.11011101*10^{011010B}$

Aún falta una cosa por precisar. Un número de punto flotante se puede escribir de diferentes formas, como lo muestra la siguiente igualdad:

$$a*2^b = a*2^b*(2^c/2^c) = (a/2^c)*2^{b+c}$$

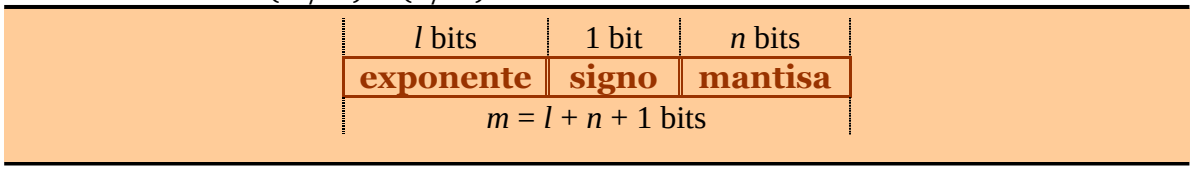


**Fig. 3.4. Formato de números de punto flotante**

Entre estas diferentes formas ¿cuál vamos a aceptar? Dado que tenemos una cantidad limitada de dígitos en la mantisa, lo mejor es aprovecharlos al máximo; por ejemplo, se puede exigir que el número empiece lo más a la izquierda posible. Esto es razonable, ya que los ceros que están a la izquierda del primer uno ocupan espacio inútilmente y en cambio hacen perder dígitos que se encuentren a la derecha, lo cual disminuye la precisión de la mantisa. De esta manera no se escribiría $0.001011*10^{00110}$, sino $0.101100*10^{00100}$.

Los números de punto flotante cuya mantisa no tiene ceros a la izquierda del primer uno se llaman *números normalizados*.

### Aritmética de punto flotante

Tenemos una representación para los números de punto flotante; falta desarrollar la aritmética usando dicha representación. Para esto solo disponemos de la aritmética entera. Debemos desarrollar un método para efectuar la aritmética de punto flotante a partir de la entera. Este método se basa en estas igualdades:

$$a*2^b * a'*2^{b'} = (a*a')*2^{(b+b')}$$
$$a*2^b / a'*2^{b'} = (a/a')*2^{(b-b')}$$
$$a*2^b + a'*2^b = (a+a')*2^b$$
$$a*2^b - a'*2^b = (a-a')*2^b$$

Para multiplicar dos números de punto flotante, basta con multiplicar las mantisas y sumar los exponentes; los resultados de estas dos operaciones constituirán la mantisa y el exponente, respectivamente, del resultado.

La suma de los exponentes no presenta ningún problema: son dos enteros binarios. La multiplicación de las mantisas puede parecer un poco rara puesto que son fracciones; pero, eso no tiene importancia, recuerde que los bits de un computador los interpreta el programador. Podemos interpretar, temporalmente, las mantisas como dos naturales de $n$ bits, multiplicarlos obteniendo un natural de $2*n$ bits y tomar los $n$ bits más significativos como la nueva mantisa. Por ejemplo, si tenemos el número 2.0 y lo queremos multiplicar por sí mismo, procedemos así:

- El número 2.0, escrito en binario, es 10.0, su representación normalizada usando $l = 5$ y $n = 8$ es: `00010 | 0 | 10000000`
- El primer paso para hacer la multiplicación es sumar el exponente con él mismo: `00010 + 00010 = 00100`.
- Tomamos la mantisa como un natural y la elevamos al cuadrado; eso en base 16 (para no escribir todos los ceros binarios) es: `80H * 80H = 4000H`.
- Tomamos los 8 bits más significativos como la mantisa: `40H = 01000000B`.
- En total tenemos: `00100 | 0 | 01000000`

La mantisa que se obtiene no está normalizada —el primer dígito de la mantisa no es 1—. La normalización se hace fácilmente; basta con correr la mantisa hacia la izquierda hasta que el primer 1 quede en la primera posición; después se

decrementa el exponente tantas veces como corrimientos se hicieron. En este caso se debe hacer 1 corrimiento, luego el número normalizado es:

| 00011 | 0 | 10000000 |
|---|---|---|

Es decir, $0.1*10^{00011B} = 0.5*2^3 = 4.0$

Cuando se normaliza, se está multiplicando por 2 —correr la mantisa— y dividiendo por 2 —restar uno al exponente— repetidas veces, luego el valor no cambia. En la práctica, la normalización se debe hacer sobre los $2*n$ bits resultantes de la multiplicación de las mantisas; si no, se pueden perder dígitos significativos.

Falta considerar el signo del resultado. Se determina fácilmente: si los signos de los operandos son iguales, el resultado es positivo; si son diferentes, es negativo. Es decir, el signo del resultado es el O-excluyente de los signos de los operandos.

Este proceso es igual al que usted sigue cuando multiplica números de la forma $a.b*10^c$: suma los exponentes, multiplica las mantisas como si fueran números enteros, corre el punto decimal tantas posiciones como dígitos fraccionarios tienen los operandos y desprecia una parte de la fracción según la precisión deseada.

Para la suma se requiere una operación previa. Como se ve en la ecuación antes dada para la suma, se exige que los dos números tengan el mismo exponente. Por lo tanto, antes de realizar la operación, los exponentes deben ser igualados. Esto se hace de manera similar a la normalización: se resta el exponente menor del mayor, llamemos a este resultado *r;* la mantisa del número con exponente menor se corre a la derecha $r$ posiciones y su exponente se iguala con el exponente del mayor:

Sean $a*2^b$ y $a'*2^{b'}$, $b > b'$
$a'*2^{b'} = a'*2^{b'}*(2^{b-b'}/2^{b-b'}) = (a'/2^{b-b'})*2^{b'}*2^{b-b'} = (a'/2^{b-b'})*2^b$

Observe que al correr la mantisa hacia la derecha, pueden desaparecer algunos dígitos (incluso todos) produciendo perdidas en la precisión de la operación. Por esto al sumar un número grande con uno pequeño el resultado puede ser exactamente igual al número mayor.

Después de igualar exponentes, se suman las mantisas y se normaliza el resultado. En el caso de la suma, el resultado puede ser mayor o igual a 1 ó tener ceros a la izquierda; la normalización se hace hacia la izquierda o hacia la derecha según sea el caso. Por ejemplo:

$0.11*10^{00010}+0.1*10^{00001}=0.11*10^{00010}+0.01*10^{00010}=1.0*10^{00010}$
La normalización se hace hacia la derecha:
$0.1*10^{00011}$

$0.101*10^{00010}+(-0.1*10^{00001})=0.101*10^{00010}+(-0.01*10^{00010})=$
$0.011*10^{00010}$
La normalización se hace hacia la izquierda:
$0.11*10^{00001}$

La resta y la división siguen líneas similares a las ya expuestas para la suma y la multiplicación, y no las presentaremos aquí.

**Estándar 754 de la IEEE**

Los fabricantes de computadores tenían la tendencia a definir cada uno su formato de punto flotante. Como consecuencia, hubo una proliferación de representaciones. Para estandarizar, la IEEE propuso un formato que presentamos a continuación.

En este estándar, hay tres tamaños para los números: 4 bytes (simple), 8 bytes (doble) y 10 bytes (extendido).

En primer lugar, vamos a estudiar los números de punto flotante de *precisión simple*. Estos destinan 8 bits para el exponente, 23 bits para la mantisa y un bit para el signo de la mantisa (fig. 3.5).

Describiendo las partes en detalle tenemos:

- Se define 2 como base de exponenciación.
- El signo (*s*) vale 1 para los negativos y 0 para los positivos.
- *Exp* es un número positivo igual al exponente más 127; lo cual quiere decir que si *exp* vale 01H, el verdadero exponente es 1 - 127, o sea -126. Si *exp* vale FFH, el verdadero exponente será 255 - 127 = 128.
- Se codifica así para facilitar la comparación de exponentes; recuerde que los números negativos en complemento a dos, son, desde el punto de vista del procesador, números positivos grandes. Hay máquinas que no comparan correctamente números en complemento a dos o que ni siquiera los usan. En esta representación, no hay problemas con tales máquinas.
- *Mant* representa la mantisa. Aquí también tenemos unas sutilezas: primero que todo, como vimos anteriormente, el punto se supone a la izquierda del número; es decir, si *mant* vale 101011, esto representa el número 0.101011.
- Una vez llegados a este punto, podemos notar otra cosa: puesto que *mant* siempre empieza por 1, ¿para qué representarlo explícitamente dado que sabemos que necesariamente está ahí? Por lo tanto, nuestro ejemplo se escribe en realidad: 01011, y sería interpretado como la mantisa 1.01011. Naturalmente, el exponente debe ajustarse en consecuencia.

La siguiente tabla resume las convenciones descritas y presenta unas nuevas:

| Valor de *Exp* | Valor del número representado |
|---|---|
| $0 < exp < 255$ | $(-1)^s * 2^{(exp-127)} * (1.mant)$ |
| $exp = 0$ y $mant = 0$ | $(-1)^s * 0$ |
| $exp = 0$ y $mant \neq 0$ | $(-1)^s * 2^{-126} * (0.mant)$ |
| $exp = 255$ y $mant = 0$ | $(-1)^s * \infty$ |
| $exp = 255$ y $mant \neq 0$ | No es un número válido |

La primera línea de la tabla resume las convenciones antes descritas. Note que, en


**Fig. 3.5. Formato flotante de precisión simple**

este caso, el exponente no puede tomar los valores 0 y 255; estos valores se reservan para casos especiales, como se mostrará a continuación.

Un problema de este esquema es que el cero no se puede representar puesto que suponemos que *mant* siempre empieza por uno. Para arreglar este defecto se define el cero como se muestra en la segunda línea de la tabla.

La tercera línea indica cómo representar valores excepcionalmente pequeños. En efecto, según la representación normal —primera línea— el número más pequeño es $1.0*2^{-126}$. Esta convención permite representar valores aun menores, puesto que no se considera que haya un 1 implícito. Por ejemplo, se podría escribir el número $0.25*2^{-126}$. Esta convención evita que los resultados se redondeen a cero prematuramente; si no existiera, todo resultado en este rango sería cero.

La segunda entrada de la tabla es un caso particular de la tercera (¿por qué?).

La cuarta entrada presenta una convención especial: cuando un cálculo da por resultado un número tan grande que no puede representarse, se le asigna este valor para indicar que se sobrepasó la capacidad de representación. Eventualmente, puede utilizarse para indicar un valor "infinito", por ejemplo, 1/0. En ese caso puede desarrollarse una seudo matemática del estilo $\infty + 1 = \infty$.

La quinta línea indica como representar un resultado que definitivamente no admite interpretación. Perversiones del estilo 0/0 ó $\infty - \infty$.

En cuanto a los números de *precisión doble*, estos no tienen mayor problema; es la misma representación anterior sólo que admite valores más grandes para *exp* y *mant*: 11 y 52 bits respectivamente. La tabla correspondiente es:

| Valor de *Exp* | Valor del número representado |
|---|---|
| $0< exp < 2047$ | $(-1)^s*2^{(exp-1023)}*(1.mant)$ |
| $exp = 0$ y $mant = 0$ | $(-1)^s*0$ |
| $exp = 0$ y $mant \neq 0$ | $(-1)^s*2^{-1022}*(0.mant)$ |
| $exp = 2047$y $mant = 0$ | $(-1)^s* \infty$ |
| $exp = 2047$y $mant \neq 0$ | No es un número válido |

En cuanto a la *precisión extendida*, aunque parecida a las anteriores, presentan una diferencia: no hay un 1 implícito en la mantisa (fig. 3.6). En esta representación se indica explícitamente cuál es el valor del dígito antes del punto decimal:

| Valor de *Exp* | Valor del número representado |
|---|---|
| $0\leq exp < 32767$ | $(-1)^s*2^{(exp-16383)}*(i.mant)$ |
| $exp = 32767$ y $mant = 0$ | $(-1)^s* \infty$ |
| $exp = 23767$ y $mant \neq 0$ | No es un número válido |

Ejemplo de precisión simple: $-\ 11.101*10^{-110}$ Cuya codificación es:

| 1 | 01111010 | 110100… [*17 ceros*] |
|---|---|---|

El mismo número, en precisión extendida, es:

| 1 | 011111111111010 | 1 | 110100… [*57 ceros*] |
|---|---|---|---|

Es conveniente anotar que el estándar de la IEEE define la precisión extendida como opcional, y no pone mayores restricciones al formato. Los tamaños para la mantisa y el exponente que presentamos son los mínimos recomendados.

En realidad, se supone que los formatos efectivamente utilizados son el simple y el doble; la precisión extendida sirve únicamente para efectuar cálculos intermedios sin perder precisión. También por esta razón, no se utiliza el uno implícito al principio de la mantisa sino que se representa el dígito explícitamente; en efecto, dado que el número se va a utilizar en cálculos intermedios, es conveniente tener todos los dígitos de la mantisa disponibles.

Por último, la IEEE impone una serie de restricciones sobre las operaciones en sí. Por ejemplo, se exige que haya precisión hasta el último dígito de la mantisa. También define los modos de aproximación: truncación, redondeo por arriba, por abajo y por aproximación.

## 3.5 LOS VALORES LÓGICOS

Los valores lógicos no tienen una representación estándar; los lenguajes de alto nivel usan diversas representaciones:

- Utilizar únicamente el bit menos significativo de la palabra para almacenar el valor: 1 es "verdadero" y cero es "falso".
- Una palabra en ceros (`00000000H`) para "falso" y, en unos (`FFFFFFFFH`) para "verdadero".
- El número cero representa "falso"; cualquier otro valor representa "verdadero" (esta es la convención usada en C).

Java va más lejos: especifica que las variables booleanas tienen dos valores, pero no dice cómo se representan ni qué espacio ocupan; eso queda oculto al programador.

En cuanto a las operaciones, es conveniente hacer aquí una distinción: algunos lenguajes de alto nivel diferencian entre las operaciones lógicas abstractas y operaciones lógicas bit a bit. Las primeras operan sobre variables y expresiones de tipo booleano; sobre el valor representado. Las segundas operan sobre la representación, sobre los bits que componen físicamente una cierta entidad. La entidad en sí no tiene que ser de tipo booleano, así que es posible hacer un Y lógico bit a bit entre dos números enteros, y el resultado será el patrón binario resultante de hacer el Y entre las parejas de bits respectivos:

```
  10101001
& 11000011
= 10000001
```

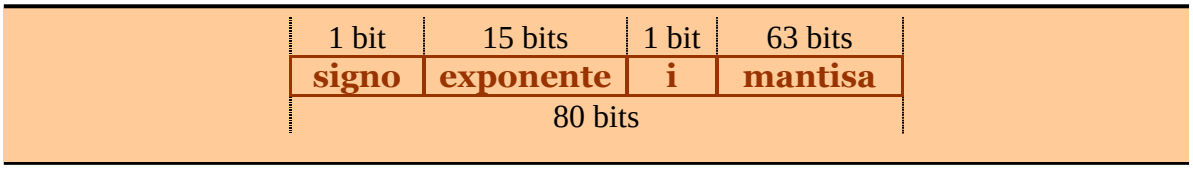


**Fig. 3.6. Formato flotante de precisión extendida**

Lenguajes como C y Java tienen diferentes operadores para diferenciar entre los dos tipos de operación:

| Operación | Bit a bit | Lógico |
|---|---|---|
| **Y** | & | && |
| **O** | \| | \|\| |
| **negación** | ~ | ! |
| **O-excluyente** | ^ | |

Las dos primeras representaciones mencionadas al comienzo de la sección, permiten efectuar las operaciones directamente sobre la representación. Es decir, se puede hacer el O, Y, etc. bit a bit sobre la representación, obteniendo el resultado correcto. En el caso particular de la primera representación, no hay que olvidar que sólo se utiliza el bit menos significativo; por ejemplo, al hacer la negación de "verdadero", `00000001B`, se obtiene `11111110B`; si solamente se tiene en cuenta el bit menos significativo, el resultado es el deseado.

La tercera representación no permite utilizar las operaciones bit a bit directamente. Por ejemplo, si tenemos los valores `00000001B` y `00000010B` y efectuamos un Y, el resultado es `00000000B`: ¡"verdadero" Y "verdadero" da "falso"! En el caso de esta representación, se debe utilizar un algoritmo que se encargue de calcular el resultado.

## 3.6 Seguridad de los datos

Hasta el momento hemos tratado la representación de los datos desde una perspectiva meramente funcional: cómo representar datos en binario de manera que posteriormente podamos procesarlos y decodificarlos.

Sin embargo, existe otra perspectiva, más pragmática si se quiere, cual es la de la seguridad. Los datos, y la información en general, están expuestos a riesgos: los datos se pueden perder, pueden ser distorsionados o divulgados de manera no autorizada.

En esta sección y la siguiente trataremos someramente algunos aspectos de la seguridad de los datos relacionados con aspectos de codificación. En concreto: confidencialidad y autenticidad —en esta sección— y disponibilidad e integridad —en la siguiente—.

### Confidencialidad

Con frecuencia es importante limitar la divulgación de los datos a un conjunto de personas autorizadas. Esto, por supuesto, se puede lograr por diversos medios; aquí nos concentraremos en lo relacionado con codificación: el uso de la criptografía. La criptografía es un campo amplio, y aquí solo trataremos algunos aspectos de la misma.

Antes hemos mencionado que la representación de la información requiere de unas convenciones que permitan codificarla y decodificarla; obviamente, para tener acceso a la información, es necesario conocer las convenciones.

En el caso de la criptografía, se habla de *cifrar* y *descifrar* la información, y su objetivo es evitar que cualquiera pueda tener acceso a la información.

¿Cómo lograr esto? En principio parece sencillo: para descifrar un texto es necesario conocer las convenciones usadas para representarlo, luego basta con ocultar esas convenciones.

En realidad no es tan simple: por medio de diversas técnicas —y de la inteligencia y el sentido común— es posible descubrir las convenciones, luego estas tienen que ser "resistentes" a este tipo de ataques. Por otro lado, hoy en día, por razones que no entraremos a profundizar, se considera conveniente que las convenciones se dividan en dos partes: un método de cifrado —que debe ser público y conocido— y una clave —la cual se mantiene secreta—.

En primer lugar, se habla de dos tipos de algoritmos: los de clave simétrica y los de clave pública. En los primeros, se usa la misma clave para cifrar y descifrar; en los segundos, se usan dos claves distintas, una de la cuales es privada y la otra pública.

En los algoritmos de clave simétrica, en general, se toma el texto como un bloque de bits, se parte en subloques y cada uno de ellos se cifra realizando operaciones con la clave. Un ejemplo muy sencillo es el siguiente: se toma una clave de $n$ bits, se parte el texto en bloques de $n$ bits y, para cada bloque, se calcula el O-excluyente con la clave. Note que para descifrar se procede de la misma manera. El anterior es solo un ejemplo, y por supuesto se dispone de algoritmos más complejos.[9]

En los algoritmos de clave pública, cada persona dispone de dos claves: la pública (que la notaremos por *Pu*) y la privada (que la notaremos *Pr*). Quien desee enviar un mensaje lo cifra usando la clave *Pu* del receptor —esta clave puede ser conocida por cualquiera—; quien lo recibe lo decodifica usando su clave *Pr*, que solo él conoce.

### Autenticidad

Existen algoritmos de clave pública que son simétricos: se puede cifrar con *Pu* y descifrar con *Pr*, o se puede cifrar con *Pr* y descifrar con *Pu*. La utilidad de lo primero es clara: puesto que se requiere *Pr* para descifrar el mensaje, solo el receptor puede enterarse del contenido del mensaje. Lo segundo no parece ser útil: si cifro con mi clave privada y envío el mensaje, cualquiera podría descifrarlo usando la clave pública. Así es, pero este último mecanismo no pretende garantizar la confidencialidad sino la autenticidad: si alguien puede descifrar el mensaje usando mi clave pública, quiere decir que fue cifrado con mi clave privada, y, por ende, que yo envié el mensaje (puesto que solo yo conozco la clave privada).

Para enviar un mensaje que garantice autenticidad y confidencialidad, se procede de la siguiente manera: se cifra el mensaje con la propia clave privada, y el resultado se cifra con la clave pública del receptor; lo primero garantiza la autenticidad y lo segundo la privacidad, ya que solo puede ser descifrado por el dueño de la clave privada. Esta es una forma de *firma digital*.

---

[9] Pero, efectivamente, el O-excluyente es una operación que interviene frecuentemente en ellos.

### Esquemas híbridos

Los algoritmos de clave pública son computacionalmente pesados, no así los de clave simétrica. Por esto, se desarrollaron los métodos híbridos: la información en sí se transmite usando un algoritmo de clave simétrica; pero, previo a enviar la información, se envía una comunicación cifrada con clave pública cuyo contenido es la clave simétrica que será utilizada en el resto de la comunicación. Entre otras ventajas, esto permite establecer la clave dinámicamente —en el mismo momento de la comunicación— en lugar de tener una clave predeterminada; incluso se puede tratar de un número aleatorio.

## 3.7 INTEGRIDAD Y DISPONIBILIDAD DE LOS DATOS

La codificación también está relacionada con la problemática de la integridad de los datos: la información está expuesta a riesgos, accidentales o malintencionados, que la pueden modificar o destruir; para evitar esto, se puede recurrir a diversas estrategias, pero, en particular, hay técnicas de codificación que ayudan a prevenir o, por lo menos, a detectar alteraciones en la información.

Un error consiste en que un bit cambia de cero a 1 ó de 1 a cero. Los errores también pueden ser múltiples y, en particular, pueden presentarse en ráfagas (varios bits seguidos que cambian de valor).[10]

Las posibles fuentes de errores son múltiples y dependen del proceso o dispositivo del que estemos hablando. En términos generales podemos decir que los errores se originan debido a dispositivos defectuosos, a fallos fortuitos de los mismos y a accidentes durante el almacenamiento o transferencia de la información.

Veamos un ejemplo. En las memorias hechas de material semiconductor, se pueden presentar defectos que hagan que un cierto bit esté “pegado” a un valor, de manera que, sin importar qué se escriba en él, siempre presentará el mismo valor. También puede ocurrir que la carga almacenada en un bit —un bit en 1— se descargue, con lo cual el bit se trasforma en cero; este caso es más complicado de detectar porque si se almacena un cero, el bit funcionará correctamente, pero si se almacena un 1, fallará (o, peor aun, funcionará o fallará según lo que se demore en realizarse la siguiente lectura). Aparte de lo anterior, pueden presentarse accidentes; por ejemplo, del espacio exterior llegan rayos cósmicos a la tierra,[11] si un rayo cósmico golpea un bit, lo puede modificar.

Otra situación se presenta en comunicaciones. Cuando se transmite información por algún medio, la señal está sometida a atenuación, distorsión y ruido, todo esto puede causar que un bit no sea correctamente recibido en el destino (ver fig. 3.7).

---

[10] Este tipo de errores se presenta en comunicaciones. Es natural porque los bits están circulando uno detrás de otro por algún medio; si en un punto se presenta una interferencia durante un cierto tiempo, afectará los bits que alcance a pasar por ese punto.

[11] Los rayos cósmicos son, en realidad, partículas de alta energía que bombardean la tierra desde el espacio: protones, electrones y iones de átomos.

Otra fuente de adulteración es la manipulación malintencionada de los datos; esto es más difícil de controlar puesto que se trata de acciones inteligentes y con un propósito y no simples errores azarosos. En particular, las manipulaciones pueden haber sido planeadas para evadir o sobrepasar los mecanismos de protección.

### Redundancia

En general, las técnicas de codificación para proteger los datos están basadas en la redundancia. Según el Diccionario de la Real Academia Española, una de las acepciones de redundancia es: "Cierta repetición de la información contenida en un mensaje, que permite, a pesar de la pérdida de una parte de este, reconstruir su contenido", o detectar que ha sido modificado, agregaríamos nosotros.

Si la información de un dato está repetida y se modifica parte del mismo, se genera una inconsistencia con la réplica; esta inconsistencia permite detectar el error, y, en ocasiones, incluso corregirlo.

Note que en la definición del DRAE dice "*cierta* repetición"; esto porque, aunque la redundancia más simple consiste en repetir literalmente la información, con frecuencia no se hace esto, sino que se genera una cierta relación entre las partes del mensaje pero sin repetirlas literalmente.

Por ejemplo, una forma usual de redundancia se presenta en los números de las cuentas bancarias: al final del número se agrega el llamado *dígito de verificación*; este dígito es un valor que se calcula a partir de los restantes dígitos de la cuenta. Cuando se necesita verificar el número, se recalcula el dígito de verificación, si no es igual al dígito final, se sabe que ha ocurrido algún tipo de error.

### Bit de paridad

El bit de paridad es una especie de "dígito de verificación" muy usado en informática; es una de las formas más simples de redundancia. A continuación se describirá su funcionamiento.

Se tiene un dato de $n$ bits ($d_n \dots d_0$), y se le adiciona un bit extra ($d_n \dots d_0 d_p$). El valor de $d_p$ se determina de manera tal que el número total de unos en ($d_n \dots d_0 d_p$) sea par; así, si el número total de unos en ($d_n \dots d_0$) es par, $d_p = 0$, si es impar, $d_p = 1$. La anterior es la llamada *paridad par*, también existe la *paridad impar* (el número total de unos es impar).

Cuando se almacena o transmite un dato, se le agrega el bit de paridad $d_p$. Cuando se recupera o recibe, se genera de nuevo el bit de paridad a partir de ($d_n \dots d_0$) y se compara con $d_p$; si son diferentes, se sabe que ocurrió un error; si son iguales, se

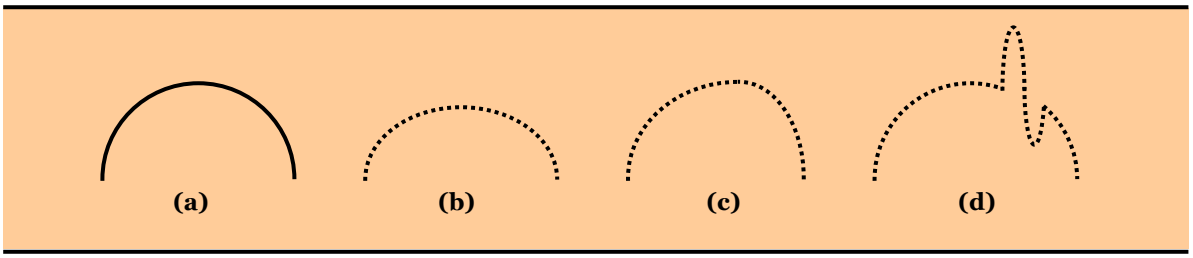


**Fig. 3.7. (a) Onda original (b) con atenuación (c) con distorsión (d) ruido**

supone que en principio la información es correcta. Sin embargo, no necesariamente es así; si un solo bit cambia —pasa de 1 a 0 ó de 0 a 1—, este cambio será detectado por el bit de paridad, pero si dos bits cambian, la paridad será igual y no se podrá detectar el error. En general, se puede detectar si ocurre un número impar de errores, pero no, si se produce un número par de errores.

Matemáticamente podemos definir el bit de paridad par así: $d_p = \left( \sum_{i=0}^{n} d_i \right)$ módulo 2

O también así: $d_p = d_n \oplus \ldots \oplus d_1 \oplus d_0$ (donde $\oplus$ es el operador O-excluyente). En la práctica, calcular el O-excluyente de los bits es el método usado para calcular el bit de paridad.[12]

Este uso simple del bit de paridad constituye un código de tipo SED (*Single Error Detecting*).

### Verificación de redundancia vertical

La *verificación de redundancia vertical* (también llamada *transversal*) es un método que se utiliza cuando, más que un dato aislado, se maneja un bloque de información.

El bloque se parte en subloques, de algún tamaño específico —8, 16, 32 bits, etc.—, y a cada subloque se le calcula la paridad. Puede imagina el bloque como una matriz a la cual se le calcula un bit de paridad para cada fila; este bit se incorpora al final de la fila correspondiente.

Este sistema permite detectar cualquier número impar de errores, y algunos de un número par de errores, pero esto último no puede ser garantizado (por ejemplo, no serán detectados si todos ocurren en el mismo subloque).

### Verificación de redundancia horizontal

La *verificación de redundancia horizontal* —también llamada *longitudinal*— se utiliza en las mismas circunstancias que la de redundancia vertical, solo que, en este caso, se calcula un bit de paridad para cada columna.

Para efectos prácticos, esta redundancia se calcula haciendo el O-excluyente bit a bit de todas las líneas; es decir, se calcula

$$FilaParidad = Fila_1 \oplus Fila_2 \oplus \ldots \oplus Fila_n$$

Y esta fila se agrega como una fila más al final del bloque.

Este método funciona mejor que el anterior cuando los errores tienden a presentarse en ráfagas; en efecto, detectará cualquier ráfaga, excepto si esta genera un número par de errores en todas las columnas afectadas.

En principio, este método y el anterior solo sirven para detectar errores; sin embargo, la combinación de los dos es un primer ejemplo de un código que permite recuperar errores, siempre y cuando se trate de un número impar de errores y todos

---

[12] Porque el O-excluyente es más fácil y rápido de calcular que la suma.

ocurran en la misma fila.[13] Para esto se procede de la siguiente manera: se recalculan las paridades —transversales y longitudinales— del bloque, y se comparan con las que vienen adjuntas en el bloque; si son iguales, suponemos que no hay errores; si son diferentes indican un error en la fila o columna respectiva: el cruce de la fila y las columnas con errores identifican los bits erróneos (ver fig. 3.8); basta con negar los bits afectados para recuperar la información original.

De manera operativa, si $Fila_i$' es la fila recibida —errónea—, la original es:

$$Fila_i = ParidadCalculada \oplus ParidadRecibida \oplus Fila_i'$$

Como comentario al margen, mencionaremos que los discos RAID usan un principio similar para recuperarse de fallos en su funcionamiento.

**Suma de verificación**

La *suma de verificación* se calcula como la redundancia longitudinal pero se usa la operación suma en lugar del O-excluyente:

$$SumaVerificación = Fila_1 + Fila_2 + \ldots + Fila_n$$

La razón para hacerlo es la siguiente: como mencionamos anteriormente, si se comete un número par de errores en una columna, el O-excluyente no lo detecta, en tanto que la suma sí puede hacerlo —por lo menos en algunos casos—; en efecto, aunque la suma en la columna no cambia, el acarreo a la siguiente puede cambiar, luego el resultado total de la suma cambia, con lo cual se descubre que ocurrió un error.

El anterior razonamiento no es válido para la columna correspondiente al bit menos significativo, puesto que este no recibe ningún acarreo. Para subsanar este problema, se suman todas las filas y el acarreo que genere la suma total se adiciona de vuelta al resultado. Por ejemplo, supongamos que las filas son de un byte, y que tenemos la suma:

$$(23)_{16} + (A5)_{16} + (2E)_{16} + (48)_{16} = (13E)_{16}$$

Puesto que estamos trabajando con filas de un byte, deberíamos conservar como resultado 3E y perder el 1, pero, en lugar de eso, sumamos este 1 al resultado y

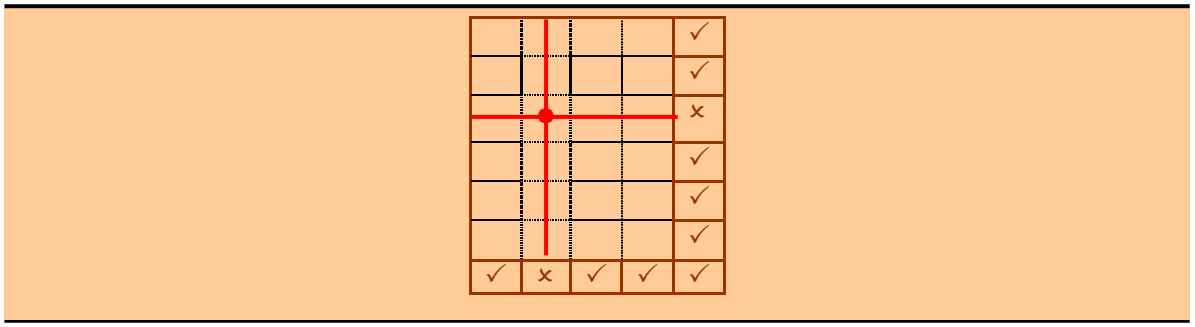


**Fig. 3.8. Detección de bits erróneos.**

obtenemos 3F como suma de verificación.

[13] También funcionaría en una columna, pero es muy improbable que esto se presente.

Este método de verificación se usa en redes de computadores —por ejemplo, se usa en el protocolo TCP—.

En términos generales, cuando se hace control de errores, es conveniente que cada bit del mensaje afecte varios bits de verificación, y que el conjunto de bits de verificación afectados sea diferente para cada bits del mensaje; esto aumenta la probabilidad de detectar un error múltiple, puesto que aunque el conjunto de errores pase sin ser detectado en un bit de verificación, habrá otro (u otros) que se verá afectado.

Desde este punto de vista, el O-excluyente de la redundancia longitudinal no es muy conveniente, puesto que cada bit del mensaje afecta solo un bit de verificación y todos los bits de una columna afectan el mismo; en tanto que la suma de verificación es una mejora porque los bits del mensaje afectan varios bits de verificación —debido al acarreo de la suma—. El método que se presenta a continuación se constituye en una mejora: usa la división como operación para generar dependencias más complejas entre los bits.

### Verificación de redundancia cíclica

La *verificación de redundancia cíclica* consiste en tratar todo el bloque de información como un solo número binario y dividirlo por un cierto número binario específico especialmente seleccionado para la tarea. El residuo de esta división es el valor de verificación que se agregará al mensaje.

Sin embargo, la división es una operación bastante costosa, especialmente cuando los números son muy grandes, por lo cual se utiliza una "aritmética simplificada", la cual procederemos a explicar.

#### *Suma y resta módulo 2*

La suma es una operación bastante parecida al O-excluyente; en efecto, si sumamos dos bits y no tenemos en cuenta el acarreo, las dos operaciones son iguales. Es decir:

$$(a + b) \text{ módulo } 2 = a \oplus b$$

En consecuencia, para simplificar la aritmética, no utilizaremos la suma como tal, sino la suma bit a bit módulo 2; lo cual es equivalente a decir que usaremos el O-excluyente bit a bit.

Curiosamente, lo mismo se cumple con la resta: $(a - b) \text{ módulo } 2 = a \oplus b$

En consecuencia tenemos: $(a + b) \text{ módulo } 2 = (a - b) \text{ módulo } 2 = a \oplus b$

En conclusión, en la aritmética bit a bit módulo 2, la suma y la resta son iguales, y son iguales al O-excluyente.

#### *División módulo 2*

El algoritmo de división funciona como el usual, con algunos ajustes:

- Si el divisor tiene $k$ dígitos binarios, los números que se separan del dividendo siempre deben ser considerados como números de $k$ bits, para lo cual, si es necesario, se pueden agregar ceros a la izquierda.

- Se dice que el divisor "cabe" en los dígitos separados del dividendo si estos empiezan por 1.
- Si el divisor cabe en los dígitos separados del dividendo, se resta usando resta bit a bit módulo 2.

Veamos un ejemplo:

```
 101010 | 11
-11     | 11001
 011
 -11
  000
  -00
    01
   -00
     10
    -11
     01
```

El cociente es 11001 y el residuo es 1.

Como comentario de cierre, diremos que en la literatura se suele describir esta aritmética como "aritmética de polinomios", esto porque se modelan los números por medio de su polinomio respectivo; por ejemplo, el número 1101 se toma como el polinomio: $\mathbf{1}*x^3 + \mathbf{1}*x^2 + \mathbf{0}*x + \mathbf{1} = x^3 + x^2 + 1$.

### *CRC (Cyclic Redundancy Check)*

Dado lo anterior, el algoritmo de CRC es sencillo:

- Se selecciona un divisor, el cual debe ser un estándar conocido por todos los que manipulen el bloque de información en cuestión.[14] Este divisor se conoce como el *polinomio generador*.
- Suponiendo que el divisor tiene $k$ dígitos, se le agregan al bloque de información, al final, $k$-1 ceros.
- Se divide el bloque de información —tomado como un gran número— por el polinomio generador usando la operación de división antes descrita.
- Se reemplazan los $k$-1 ceros agregados al final del bloque con los $k$-1 dígitos del residuo.

Para verificar el bloque, basta con dividirlo por el polinomio generador; el residuo debe ser cero, de lo contrario, ha ocurrido un error.

[14] La selección de este número no es trivial, y se han propuesto varios divisores estándares bien diseñados.

Por ejemplo, si el bloque de información es 10101 y el divisor es 11 (es decir, el polinomio generador es $x + 1$), como el divisor tiene dos dígitos, le agregamos un cero al bloque de información (101010) y lo dividimos por 11. Esta es la división que hicimos anteriormente, la cual nos dio un residuo de 1; agregamos el residuo en lugar del cero (101011) y este es el valor codificado. Puede verificar que al dividirlo por 11, obtenemos un residuo de cero; también puede verificar que si cambia cualquier bit, el residuo es 1.

Esta técnica se ha usado para control de errores en redes —en Ethernet, por ejemplo— y en discos magnéticos y ópticos.

### Código de Hamming

Como mencionamos anteriormente, con el propósito de incrementar la probabilidad de detectar un error múltiple, un buen código de detección de errores hace que:

- Cada bit de datos participe en el cálculo de varios bits de paridad.
- Distintos bits de datos afecten los bits de paridad de manera diferente.

Para desarrollar un código con detección de errores, es necesario fortalecer la segunda condición: cada bit de datos debe afectar de manera única los bits de paridad; esto porque, si dos bits de datos afectan de la misma forma los bits de paridad, y se detecta un error, no se sabrá cuál de ellos produjo el error, y, por ende, no sabremos cuál corregir.

En consecuencia, el arte y la ciencia de crear un código con corrección de errores consiste en distribuir los bits de datos de manera tal que cada uno de ellos afecte un conjunto diferente de bits de paridad. ¿Cómo lograr esto?

A continuación presentaremos un ingenioso sistema de corrección de errores diseñado por Richard Hamming.

Supongamos que tenemos $n$ bits de datos ($d_i$) y $k$ bits de paridad ($p_i$) —en los ejemplos usaremos $n = 4$ y $k = 3$—; el problema consiste en decidir cuáles bits de datos participan en el cálculo de cada uno de los bits de paridad, de manera tal que, si se produce un error, podamos identificar el bit erróneo; es decir, en caso de fallo, cada bit de datos debe generar un patrón de error único.

La idea de Hamming consiste en numerar los bits de datos empezando en uno (en nuestro caso, $d_4d_3d_2d_1$); después se toman los subíndices en binario sobre $k$ bits (en nuestro ejemplo, $k = 3$, y tenemos: $d_{100}d_{011}d_{010}d_{001}$).

Los bits que componen el subíndice binario indican en cuáles bits de paridad participa el bit de datos en cuestión: si el $i$-ésimo bit está en 1, indica que el bit participa en el cálculo de $p_i$, y si está en 0, que no participa. Por ejemplo, tomemos $d_3$: 3 en binario es 011, puesto que el primer y segundo bits están en 1, esto indica que $d_3$ participa en $p_2$ y en $p_1$, en cambio no participa en $p_3$ (porque el tercer bit está en cero); En cuanto a $d_4$, puesto que 4 en binario es 100, esto indica que $d_4$ participa en $p_3$, pero no participa en $p_2$ y en $p_1$.

En nuestro ejemplo tendríamos:

$$p_3 = d_4$$
$$p_2 = d_3 \oplus d_2$$
$$p_1 = d_3 \oplus d_1$$

En la figura 3.9 puede ver cómo reaccionan los bits de paridad ante un fallo de cada uno de los bits de datos. Por ejemplo, si $d_1$ falla, $p_1$ va a señalar un error —porque $d_1$ participa en el cálculo de $p_1$—, pero $p_2$ y $p_3$ no se dan por enterados —porque $d_1$ no participa en su cálculo—.

Adicionalmente, note que si interpreta ✓ como cero y × como 1, los bits de paridad indican cuál bit de datos falló; por ejemplo, el patrón de error de $d_3$ es ✓××, lo cual corresponde a 011, es decir, 3 en binario. Con esto podemos identificar el bit erróneo y podemos proceder a corregirlo.

Nunca falta la pregunta insidiosa: puesto que los bits de paridad se almacenan o se transmiten junto con los bits de datos, ¿qué ocurre si falla un bit de paridad? ¡Detectaríamos un falso error y "corregiríamos" un bit que es correcto!

Hamming también dio una respuesta ingeniosa a este problema: los bits de paridad se mezclan con los de datos y se numeran con ellos, pero teniendo cuidado con que a los bits de paridad les toquen las posiciones que son potencias de dos. En nuestro ejemplo, tomaríamos los 4 bits de datos y los 3 de paridad y los numeraríamos de 1 a 7, pero los bits $d_4$, $d_2$ y $d_1$ serían los bits de paridad. Es decir, el dato sería: $d_7 d_6 d_5 d_4 d_3 d_2 d_1$ (los bits de paridad están en rojo).

Con esto, las ecuaciones anteriores se transforman en:

$$d_4 = d_5 \oplus d_6 \oplus d_7$$
$$d_2 = d_3 \oplus d_6 \oplus d_7$$
$$d_1 = d_3 \oplus d_5 \oplus d_7$$

Ahora, supongamos que falla el bit de paridad $d_2$, pero $d_4$ y $d_1$ están bien; esto nos daría el patrón de error ✓×✓, lo cual indica que falló el bit 2 (010), lo cual es preciso. Es decir, con esta numeración, si los bits de paridad fallan, se "autoincriminan".

Este es un ejemplo de un código de tipo SEC (*Single Error Correcting*).

| Bit de datos que falla | $p_3$ | $p_2$ | $p_1$ |
|---|---|---|---|
| Ninguno | ✓ | ✓ | ✓ |
| $d_1$ | ✓ | ✓ | × |
| $d_2$ | ✓ | × | ✓ |
| $d_3$ | ✓ | × | × |
| $d_4$ | × | ✓ | ✓ |

**Fig. 3.9. Bits de paridad en código de Hamming**

Esta técnica se usa para detectar fallas en memorias RAM.

### Seguridad e integridad

**¿Qué es ...**

**... una función de *hash*?** Supongamos que estamos manejando datos que pertenecen a un conjunto `A`; este conjunto es muy grande y, en la práctica, relativo a su tamaño, estamos usando pocos de sus elementos.

En estas circunstancias, puede ser conveniente "proyectar" los elementos de `A` a otro conjunto `B` cuya cardinalidad sea más cercana al número de elementos que realmente estamos utilizando. Una función de *hash*, *h*, es una función de `A` en `B`, que se encarga de hacer esta traducción.

Aunque no se puede garantizar, la función de *hash* trata de lograr que si $a \neq b$, y $a$ es "cercano" a $b$, entonces $h(a) \neq h(b)$. Si $h(a) = h(b)$, se dice que ha ocurrido *una colisión*.

Los mecanismos antes expuestos son más bien de bajo nivel, es decir, basados en hardware o en software muy cercano al hardware. En cualquier caso, están pensados más para errores y accidentes, no para manipulación deliberada de la información; esta situación se trata en niveles de software más altos y se usan técnicas criptográficas.

Una técnica consiste en usar funciones de *hash* unidas a algoritmos de clave pública. A continuación se describe su funcionamiento.

Una función criptográfica de *hash*, *h*, recibe como parámetro un mensaje y retorna un valor con un cierto número determinado de bits. La función debe cumplir dos condiciones:

- Dado un cierto valor $p$, debe ser imposible en la práctica encontrar un mensaje $m$ tal que $h(m) = p$.
- Debe ser imposible en la práctica encontrar dos mensajes $m$ y $m'$ tales que $h(m) = h(m')$.[15]

Dado un mensaje $m$, se calcula $h(m)$, se cifra con la llave privada de quien envía y se agrega este resultado al mensaje —la información en sí del mensaje puede quedar descubierta—. En otros términos, el mensaje es: $(m, Pr(h(m)))$.

Puesto que el resultado está cifrado, se garantiza la autenticidad, y como debe ser igual a $h$ aplicada al mensaje, quien lo recibe puede verificar que los datos no han sido adulterados. Es decir, quien lo recibe compara: $h(m) = Pu(Pr(h(m)))$. Si el mensaje ha sido modificado, el $h(m)$ calculado, no coincidirá con el $Pu(Pr(h(m)))$ enviado en el mensaje.

Este sistema se constituye también en una forma de firma digital.

**Disponibilidad**

[15] Esto no quiere decir que no existan tales $m$ y $m'$, sino que en la práctica es muy difícil encontrarlos.

La disponibilidad se puede aumentar mediante la redundancia en el hardware: tener equipos o dispositivos replicados con la misma información.

Sin embargo, la codificación de la información puede ayudar por medio de los códigos con recuperación de errores; en efecto, si se presentan fallas más bien leves en el hardware, estos códigos pueden, de todas maneras, recuperar la información.

Por ejemplo, los discos RAID usan replicación de la información (discos "espejo") y sistemas parecidos a una mezcla de verificación de redundancia horizontal y vertical.

## 3.8 CONSIDERACIONES GENERALES

Cuando hablamos de representar, quiere decir que tenemos un conjunto de $m$ entidades —"significados", si se quiere—, que serán representados por medio de $m$ símbolos. Para esto necesitamos $m$ significantes, $m$ entidades físicamente distintas y reconocibles entre ellas.

Si nos limitamos al caso del sistema binario, todos los significantes estarán constituidos por un cierto número $n$ de dígitos binarios, lo cual implica que podemos tener $2^n$ símbolos diferentes. Esto es una limitante, porque si $m$ no es potencia de 2, tendremos códigos desperdiciados; la representación no será compacta, y gastaremos más espacio del teóricamente necesario.

Este es el caso de la representación BCD: se representan los 10 dígitos decimales usando 4 bits, luego en cada dígito de desperdician 6 códigos; la representación de números en BCD consume más espacio que la binaria.

En ocasiones un código no es compacto por otro tipo de consideraciones. Por ejemplo, el código ASCII ocupa 8 bits, pero usa 7, puesto que el más significativo siempre vale cero; de esta manera los códigos de los caracteres siempre son números positivos sin tener que entrar en consideraciones de cómo se representa el signo en una máquina específica. Pero para lograr este efecto desperdiciamos códigos.[16]

En principio, cada codificación representa una entidad, pero, en el caso de los códigos de detección de errores, es necesario que algunas de estas codificaciones sean inválidas; las codificaciones inválidas representan errores. Esto quiere decir que se necesitan más bits. Note que si tenemos un cierto código que detecta errores de un bit, es necesario que, para cada codificación, todas las que difieran de ella en un bit sean inválidas.

Este concepto se puede expresar en términos de una métrica llamada *distancia de Hamming*: si se tiene dos codificaciones $c_1$ y $c_2$, su distancia es el número de unos que hay en $c_1 \oplus c_2$; note que el número de unos en esta expresión indica en cuántos bits difieren $c_1$ y $c_2$, o, dicho de otra manera, cuántos cambios de un bit habría que hacer para transformar $c_1$ en $c_2$ —o viceversa—.

[16] Representamos 128 caracteres cuando podrían ser 256.

En un código con detección de errores de un bit, se requiere que haya una distancia de Hamming de 2 entre codificaciones vecinas: todas las vecinas de una codificación válida deben ser inválidas, pero dos codificaciones válidas pueden compartir vecinas.

Ahora, en un código con corrección de errores de un bit, para cada codificación válida, todas sus vecinas —las que se encuentran a una distancia de Hamming de 1— son cuasi-válidas, puesto que denotan la ocurrencia de un error pero permiten identificar la entidad. También se podría decir que en un código con corrección de errores de un bit, cada entidad se representa con un conjunto de codificaciones: la que le corresponde y todas aquellas que difieren de esta solo en un bit. En este tipo de códigos, se requiere que haya una distancia de Hamming de 3 entre codificaciones válidas; así, dos codificaciones válidas pueden cambiar en un bit cada una y todavía no se meten en el conjunto de la otra. La figura 3.10 ilustra estos conceptos.

Como se puede ver en la fig. 3.10, detectar o corregir errores implica usar más bits para codificar el mismo número de entidades, lo cual implica que son más caras —puesto que consumen más espacio de almacenamiento, o más ancho de banda—. No debemos olvidar estos dos principios:

- Nunca podemos estar completamente seguros.
- Mayor seguridad implica mayor costo.

Además, el concepto de riesgo no es absoluto; algunos dispositivos y medios tienen una mayor probabilidad de fallo que otros, y las implicaciones también cambian: algunas fallas son más graves que otras.

En consecuencia, la seguridad no es absoluta; se debe elegir el nivel de seguridad requerido por la aplicación, o el riesgo que se está dispuesto a correr, o la seguridad que se puede —o se está dispuesto a— pagar.

Como mencionamos, los diferentes medio y dispositivos tienen probabilidades de error diferentes; por ejemplo, los CD son más susceptibles a cometer errores que los discos duros o las memorias semiconductoras; y las ráfagas de errores son más probables en comunicaciones y discos que en memorias semiconductoras. Por otro lado, los diversos medios de control de errores ofrecen diferentes tipos y niveles de protección; en consecuencia, el tipo y nivel de protección debe ser elegido acorde

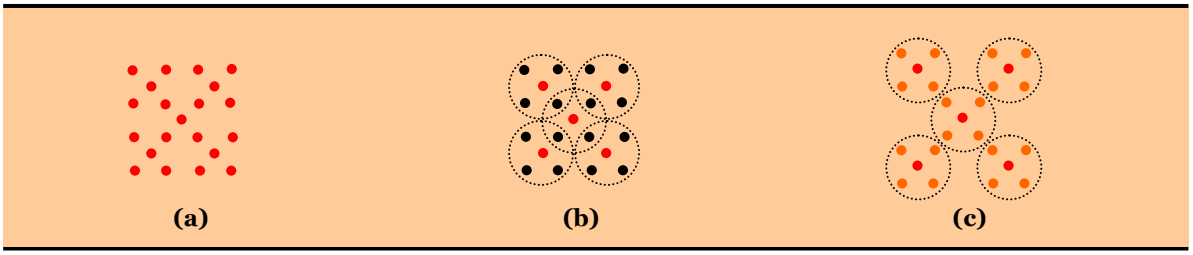


**Fig. 3.10. Código (a) compacto (b) con detección de 1 bit (c) con corrección de 1 bit**

con el tipo y nivel de errores que se presentan en el dispositivo en cuestión.

En ocasiones, tanta preocupación por los errores y las fallas parece frisar con la paranoia; no se debe olvidar lo siguiente: cualquier probabilidad de error, por pequeña que ella sea, y dados los inmensos volúmenes de información que se almacenan y procesan en los computadores, se volverá una certeza; no es un asunto de si "sí" o "no", sino de "cuándo".

Como comentario de cierre diremos que hay otros tipos de codificaciones que persiguen objetivos diferentes. Por ejemplo, las técnicas de comprensión también son codificaciones; solo que en este caso —contrario al control de errores— se pretende eliminar toda redundancia para que la información ocupe menos espacio. Se disminuye el costo de almacenamiento y transmisión, pero se aumenta el de procesamiento; nada es gratis.

También, aquí hemos estudiado esencialmente códigos de tamaño fijo, pero hay códigos de tamaño variable; es decir, códigos donde las codificaciones pueden usar un número diferente de bits. Este es el caso en los algoritmos de comprensión, así como en Unicode representado con UTF-8 o UTF-16.

---

**EJERCICIOS**

**1-** Interprete la secuencia de bits como:

| Representación | 0100 1000 0110 1001 |
|---|---|
| Natural —número sin signo— | |
| Entero en signo y magnitud | |
| Entero en complemento a 2 | |
| Caracteres —ASCII— | |

**2-** Interprete las siguientes secuencias de bits:

**a-** `FFD8H` como entero en complemento a dos.

**b-** `0231H` como BCD.

**c-** `8C60H` en punto flotante —4 bits de exponente, el signo y 11 de mantisa—.

**d-** `3734H` caracteres ASCII.

**3-** **a-** ¿Qué número representa `8000H` interpretado en complemento a dos?

**b-** ¿Es posible codificar $2^{15}$ usando complemento a 2 sobre 16 bits?

**4-** Codifique en binario:

**a-** `115` en complemento a dos usando 8 bits.

**b-** `-83` en complemento a dos y en signo y magnitud usando 8 bits.

**c-** `4895` en BCD.

**d-** "HoLa?" en ASCII.

**5-** En el lenguaje C, hay dos tipos de enteros: los `int`, que suelen tener un rango desde `-2147483648` hasta `2147483647`; y los `unsigned int` suelen tener un rango de 0 hasta `4294967295`. ¿Cómo se explican estos valores?

**6-** **a-** Complete la siguiente tabla (en los dos casos, represéntelos sobre 8 bits):

| Número | Complemento a 2 | Signo y magnitud |
|---|---|---|
| 37 | | |
| -37 | | |

**b-** Extienda los resultados de la parte a- para que queden en 16 bits.

**7-** Se están representando entidades usando 12 bits.

**a-** ¿Cuántos valores distintos se pueden representar?

**b-** Se están representando números con signo, y se desea que positivos y negativos estén lo más balanceados posible, ¿cuál sería el rango de números?

**c-** En general, para representar *n* entidades, ¿cuántos bits se necesitan?

**8-** Para cada uno de los siguientes valores indique cuál es el número mínimo de bits para poder representarlos (sin signo).

**a-** `4095, 4096, 4097, 741, 0`

**b-** En general, si se tiene un número *m*, ¿cuántos bits se necesitan?

**9-** Se quiere desarrollar un computador con elementos de tres estados ("*trits*").

**a-** ¿Cuántos trits piensa usted que debería tener un byte?

**b-** En general, si se desea representar entidades usando *n* bits, ¿cuántos trits se necesitan?

**10-** Las cadenas de ADN están compuestas por secuencias de 4 moléculas llamadas *bases*: A, C, G y T. Las bases se interpretan agrupándolas de 3 en 3; cada grupo de 3 se llama un codón. Las proteínas, a su vez, son secuencias de aminoácidos. Puesto que cada codón codifica un aminoácido, una secuencia de codones codifica la "fórmula" para construir una proteína.

**a-** Según lo anterior, ¿cuántos aminoácidos podría haber?

**b-** En realidad hay 20 aminoácidos diferentes; de los códigos restantes, unos se usan para otros efectos, y otros se usan para codificación redundante de algunos aminoácidos. ¿Por qué puede ser útil tener codificaciones redundantes para algunos aminoácidos?

**11-** Hasta ahora hemos trabajado las cantidades de bits como números enteros. En ocasiones tiene sentido trabajar con números fraccionarios de bits.

**a-** En BCD se usan 4 bits para representar 10 valores diferentes. En teoría, y usando cantidades fraccionarias de bits, ¿cuántos bits se necesitarían para codificar 10 valores?

**b-** En consecuencia, ¿qué porcentaje de los bits se está desperdiciando?

**c-** 8 dígitos BCD se representan en 32 bits; según lo anterior, ¿cuántos bits está siendo usados efectivamente?

**d-** ¿Cuántos valores binarios pueden ser representados con el número de bits obtenido en el punto anterior? ¿Coincide con los valores BCD que se pueden representar en 32 bits?

**12-** El algoritmo presentado en el texto para decodificar el valor de un número representado en complemento a 2 maneja dos casos: si la representación empieza por 1 ó si empieza por cero. Existe un algoritmo para encontrar el valor sin manejar casos: si se tiene el número $(d_n \ldots d_0)$ en complemento a dos, su valor viene dado por $-d_n * b^n + \sum_{i=0}^{n-1} d_i * b^i$ . Demuestre esta afirmación.

**13-** **a-** Se tienen los números positivos $(1011011)_2$ y $(10101)_2$; reste el segundo del primero usando complemento a 2.

**b-** Repita el ejercicio anterior pero esta vez restando el primero del segundo.

**14-** Se tiene la siguiente suma:

```
   B741H
+  A326H
```

**a-** ¿El método de suma cambia si los números están representados en complemento a dos o si son números sin signo?

**b-** ¿El resultado es válido si los números están en complemento a dos? ¿Y si son números sin signo?

**c-** si el segundo operando es E326H ¿cambian las respuestas de la parte b-?

**15-** En la sección de operaciones aritméticas, se presentó un método de multiplicación para números de $n$ bits.

**a-** ¿El método funciona con números negativos (representados en complemento a 2)? Justifique matemáticamente.

**b-** ¿Qué ocurre si se toma como respuesta solamente los $n$ bits menos significativos? (considerando como desbordamiento a los números que necesita más de $n$ bits para su representación).

**16-** Se tienen números de punto flotante con 8 bits de exponente, 1 bit de signo y 15 bits de mantisa. Represente $12.25*2^{-15}$ usando estas convenciones.

**17-** Sin hacer cálculos ¿Cómo se puede saber si un número binario es par o impar? ¿Funciona su método si el número está en complemento a 2?

**18-** Otra forma de sumar dos números BCD (xx, yy) es la siguiente:

```
   yy
   66H
+  xx
   --
   rr
```

**a-** En este caso ¿Cómo se detecta si los dígitos del resultado son correctos o no? ¿Cuáles son los valores para corregir la respuesta?

**b-** Dado que el computador no puede sumar los tres números de una vez, sino primero dos y al resultado le suma el tercer número ¿Afecta el orden de la suma su respuesta en la parte a-? (i.e. (yy+66H)+xx contra (yy+xx) + 66H). Ayuda: cuidado con el acarreo.

**c-** ¿Qué ventajas y desventajas le ve a este método con respecto al explicado en el texto?

**19-** Se tienen 2 números de punto flotante, $x$ y $y$.

**a-** ¿En qué casos $x*y$ produce un desbordamiento?

**b-** ¿En qué casos $x+y$ produce un desbordamiento?

**c-** ¿En qué circunstancias $x+y = x$? (¡Atención!, no sólo si $y = 0$).

**d-** Si se tienen 3 número de punto flotante ¿Siempre da lo mismo calcular $(x+y) + z$ que $x + (y + z)$?

**20-** Se tienen números de punto flotante con $n$ bits de exponente (en complemento a dos), uno para el signo y $m$ de mantisa; la mantisa es de la forma 0.mant. La base de exponenciación es dos.

**a-** ¿Cuál es el máximo número que se puede representar?

**b-** ¿Cuántos números en total se pueden representar?

**c-** ¿Cuántos números se pueden representar en el rango 0 a 1?

**21-** Resuelva el punto anterior, con las mismas convenciones, pero con 16 como base de exponenciación —es decir, números de la forma $A*16^B$—. Tenga presente que siguen siendo números binarios.

**22-** Con la misma representación del punto anterior —$A*16^B$—:

**a-** Se tiene un número cuya mantisa, en binario, empieza por 0.01, ¿es posible normalizarlo a la forma 0.1?

**b-** ¿En qué casos se puede normalizar la mantisa?

**c-** ¿Cómo podría definirse la normalización para este tipo de números?

**23-** Se tiene un número $n$ con la misma representación del punto anterior:

**a-** Si se divide $n$ por 16, ¿qué valor cambia en la representación $n$?, ¿cómo cambia? Suponga que no ocurre ningún desbordamiento.

**b-** Si se divide $n$ por 2, ¿qué valor cambia en la representación $n$?, ¿cómo cambia? Suponga que no ocurre ningún desbordamiento.

**c-** Si se divide $n$ por 2 cuatro veces seguidas ¿qué valor cambia en la representación $n$?, ¿cómo cambia? ¿El resultado siempre es igual al de la parte a-?

**24-** El objetivo de este ejercicio es desarrollar una representación para los números de punto fijo. En esta representación, los números tienen una parte entera y una fraccionaria; no hay exponente. De los *n* bits usados para la representación, se reserva una cantidad fija para la fracción. Por ejemplo, si se usan 8 bits con 3 para la fracción, el número 1110.01 se representa por: 01110010 donde se supone que el punto decimal está entre el tercero y el cuarto bit —numerando de derecha a izquierda—.

**a-** ¿Cómo se efectúa la suma de números en punto fijo?

**b-** Desarrolle una representación para los números negativos en punto fijo. La suma desarrollada en a- ¿Funciona con su representación?

**c-** Desarrolle un método para hacer la multiplicación en punto fijo.

Utilice solo aritmética entera (+, -, *, etc.) y corrimientos.

**25-** En el texto se mencionó una técnica simple de cifrado con clave simétrica: efectuar el O-excluyente de la clave con los subloques del mensaje. ¿Qué ocurre si en el mensaje hay largas secuencias de ceros? ¿Cómo se vería el resultado? ¿Y si son secuencias de unos?

**26-** Los métodos de cifrado se basan en que el número de posibles claves es tan grande que no es posible probarlas todas sistemáticamente. Suponga que se están usando claves de 128 bits, y que se pueden probar un millón de claves por segundo. ¿Cuánto tiempo se necesitaría para probarlas todas? Exprese el resultado en años.

**27-** La transliteración es una técnica de cifrado que consiste en reemplazar cada letra por otra —la 'A' por la 'M', la 'B' por la 'J', etc.—. Por otro lado, el uso de las letras en un idioma presenta una frecuencia característica. Si se sabe en qué idioma se realizó una transliteración, ¿cómo se puede romper la clave?

**28-** La técnica del punto anterior se puede mejorar así: se selecciona un número decimal, y se escribe repetidas veces debajo de las letras del mensaje. Luego, se cambia cada letra por la que se encuentra tantas posiciones adelante en el alfabeto como indica el número que le corresponde. Por ejemplo, si el número es 123, y el mensaje es "barcos", se convertiría en: "ccudqv". ¿Se puede romper esta clave con la técnica usada en el punto anterior? Explique.

**29-** Una técnica, simple pero ineficiente, de redundancia para detectar errores consiste en enviar dos copias del mensaje; el receptor efectúa el O-excluyente de las dos copias, y si le da cero no hay errores. Diseñe una técnica basada en el mismo principio —redundancia pura— pero que permita corregir un error.

**30-** En el texto se mencionó que la verificación de redundancia vertical permite corregir algunos errores con un número par de bits. Describa en qué condiciones se pueden detectar estos errores (los bits pueden estar en varios subloques y no necesariamente están contiguos).

**31-** En el texto se mencionó que la combinación de verificación de redundancia horizontal y vertical permite corregir la fila errónea con la fórmula:

$$Fila_i = ParidadCalculada \oplus ParidadRecibida \oplus Fila_i'$$

Explique por qué y cómo funciona esta fórmula.

**32-** Se están representando valores de $n$ bits más un bit de paridad ($d_n \ldots d_1 d_p$).

**a-** Si $x$ y $y$ son dos de estas codificaciones, ¿es $x \oplus y$ una codificación válida? (el O-excluyente incluye el bit de paridad). Demuestre su respuesta.

**b-** Sean las codificaciones $v_1$ = `(0…011)`, … $v_2$ = `(0…0101)`, $v_n$ = `(10…01)`, es decir, para $v_i$, $d_i = 1$, $d_0 = 1$, y los demás $d_j = 0$. Se tiene el valor $c = (c_n \ldots c_1)$, muestre que $(c_n \bullet v_n \oplus \ldots \oplus c_2 \bullet v_2 \oplus c_1 \bullet v_1)$ es la codificación de $c$.

**33-** Desarrolle un código de Hamming para corregir un error en 8 bits de datos. ¿Cuántos bits de paridad se requieren? Escriba las respectivas ecuaciones.

**34-** Se quiere desarrollar un código que pueda detectar hasta $n$ errores:

**a-** ¿Cuál debe ser la distancia de Hamming entre codificaciones vecinas?

**b-** ¿Y si se trata de un código para corregir hasta $n$ errores?

**35-** En el texto se mencionó que un código de corrección de errores se puede ver como si cada entidad estuviera representada por un conjunto de codificaciones. Ahora, si se quiere representar $2^k$ entidades, se usan $p$ bits de paridad y se quiere poder corregir un error:

**a-** ¿De qué tamaño debe ser cada conjunto? Expréselo en términos de $k$ y $p$.

**b-** ¿Cuántas codificaciones en total debe tener el código?

**c-** Encuentre una ecuación que caracterice a $p$ en términos de $k$. Es decir, una ecuación que solo dependa de $p$ y $k$.

**d-** Basado en la ecuación anterior, encuentre cuántos bits de paridad se necesitan para los siguientes valores de $k$: 4, 8, 16, 64.

**36-** En un código de Hamming de corrección de un bit, ¿qué puede ocurrir si se presentan dos errores?

**37-** Se está transmitiendo por un cable a 10 Megabits/s. En un punto del cable se presenta una perturbación durante `1/10.000` de segundo, ¿cuántos bits alcanza a afectar?

**38-** Se está transmitiendo por un cable, y la probabilidad de que un byte se altere es de `1/1'000.000`. Encuentre la probabilidad de que un mensaje llegue bien para los siguientes tamaño de mensaje: 1 Kilobyte, 1 Megabyte, 10 Megabyte. Suponemos que las alteraciones que ocurren son eventos independientes.