Bloque 1. Tema 3.
Sistemas de numeración, códigos y representación interna de la información

Licenciatura en Matemática

Asignatura Informática

Curso 2008-2009

Objetivos

- Entender cómo se representa internamente la información en un computador.
- Comprender la relación entre la codificación externa e interna de la información: distintos códigos.
- Saber manejar las representaciones y operaciones en distintas bases.
- Conocer los distintos tipos de representación interna para los tipos de datos más habituales.
- Ser capaz de realizar conversiones a la notación IEEE-754.

Bibliografía

- Prieto, Lloris y Torres.
- Zamarreño y cols.

Contenidos

1. **Introducción**
2. **Sistemas de Numeración**
   1. Sistemas de representación más usuales
   2. Representación en base *b: decimal y octal*
   3. Representación en binario natural
   4. Operaciones en binario natural
   5. Notación y operaciones en complementos
3. **Códigos intermedios de representación**
   1. Códigos intermedios: octal y hexadecimal
   2. Códigos de E/S: BCD, ASCII, EBCDIC
   3. Detección de errores
4. **Representación interna de la información**
   1. Introducción
   2. Lógicos, caracteres y complejos
   3. Enteros: sin signo, signo y magnitud, complementos a uno y a dos, notación sesgada
   4. Los números reales: la notación IEEE 754

3.1. Introducción

- El ordenador procesa datos/información:
  - *instrucciones que forman el programa (indica* qué debe hacerse) y
  - *los datos con los que debe trabajar*
- **Instrucciones y datos deben almacenarse dentro del ordenador**
- La información internamente se representará mediante un **alfabeto**, utilizando un **código o representación interna**.
- Físicamente la información se presentará mediante una combinación de **bit**s (que viene de la expresión inglesa **bi**nary digi**t**, y que físicamente se convertirá en dos niveles de tensión diferenciados).

Externamente, manejamos información

- Utilizamos un lenguaje, también llamado código, que se construye sobre un alfabeto
- Si queremos comunicarnos con el ordenador, debemos usar un alfabeto que después se pueda representar dentro del mismo:
  - ***Caracteres Alfabéticos****:* Mayúsculas y minúsculas del alfabeto inglés.
  - ***Caracteres Numéricos:*** *Del cero al nueve.*
  - ***Caracteres Especiales:*** {,}, #, $, %, &, _, +. -. *, /, \, (,), ?, !, [,]
  - ***Caracteres de Control:*** Representan órdenes de control al ordenador: EOL, EOT, SYNC, ESC, BEEP, CTRL
  - ***Caracteres Gráficos:*** Permiten "dibujar" figuras o iconos elementales.

Generalmente nos referiremos a estas clases como:

- ***Caracteres alfanuméricos****: que abarcan las dos* primeras.
- ***Caracteres de texto:*** *que abarcan las tres primeras* categorías.

- Es necesaria, por lo tanto, una codificación externa mediante códigos de Entrada/Salida normalizados, que se puedan traducir de la notación externa a la representación binaria correspondiente.

$$\{a, b,\ldots,z, A, B,\ldots, Z, 0,\ldots, 9,(,),\ldots,[,]\} \rightarrow \{0,1\}^n$$

$$\alpha \rightarrow \beta$$

Donde β *está codificado mediante n bits.*

- Los códigos de Entrada/Salida, aunque normalizados, pueden variar en función del fabricante.
- La operaciones que normalmente realizamos con los símbolos tienen su equivalente en las operaciones sobre el alfabeto binario.
- Para facilitar la traducción entre los códigos externos (que nosotros entendemos) y los internos (en binario, difícil de comprender para el ser humano) existen lo que se denominan códigos intermedios, como pueden ser el código octal o el hexadecimal.

¿Qué vamos a ver en el tema?

- Conceptos sobre sistemas de numeración para entender las distintas representaciones.
- Revisar los códigos E/S e intermedios más conocidos.
- Estudiar la representación interna de los datos.

3.2. Sistemas de numeración

3.2.1. Sistemas de Numeración más usuales

- Sistemas de numeración aditivos
- Sistemas de numeración posicionales

  Un sistema de numeración en base *b* utiliza *b* símbolos de un alfabeto y el valor de cada número dependerá de la posición que ocupe cada símbolo.

  Por lo tanto, un número no será más que una secuencia de cifras (elegidas entre los b símbolos posibles).

3.2.2. Representación en base b

Sistema de Numeración Decimal, base 10

- Símbolos permitidos, *b*= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Ejemplo:

$1234{,}567 = 1*10^3 + 2*10^2 + 3*10^1 + 4*10^0 + 5*10^{-1} + 6*10^{-2} + 7*10^{-3}$

- Si generalizamos, para un número N cualquiera:

$N = \ldots + n_4 \cdot b^4 + n_3 \cdot b^3 + n_2 \cdot b^2 + n_1 \cdot b^1 + n_0 \cdot b^0 + n_{-1} \cdot b^{-1} + n_{-2} \cdot b^{-2} + \ldots$

Sistema de Numeración octal, base 8

- Símbolos permitidos, *b*= {0, 1, 2, 3, 4, 5, 6, 7}

  Ejemplos:

  $N=1234{,}567_{/8}$ en base 8:

  $1234{,}567_{/8} = 1*8^3 + 2*8^2 + 3*8^1 + 4*8^0 + 5*8^{-1} + 6*8^{-2} + 7*8^{-3} = 668{,}7324\ldots_{/10}$

  ¿Cuál es la representación en base 10 de $N = 13{,}78_{/8}$?

  ¿Cuál es la representación en base 10 de $N = 24{,}67_{/8}$?

3.2.3. Representación en binario natural

- **Representación en base dos**
  - En el sistema de Numeración en base dos o Binario, y sólo se usan dos elementos para representar cualquier número b={0, 1}.
  - Estos elementos del alfabeto se denominan cifras binarias o bits.

Ejemplo: Tabla de números en binario del 0 al 7 (con tres bits).

Ejemplo

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

¿ Cuántos bits son necesarios para representar los símbolos de un alfabeto?

{a, b,…,z, A, B,…, Z, 0,…, 9,(,),…,[,]} → $\{0,1\}^n$

$\alpha \rightarrow \beta$

- Para cada alfabeto tenemos: $|\alpha| = m$ y para un *n* dado tenemos $|\beta| = 2^n$.
- En general, $m \neq 2n$, pero necesitamos representar todos los caracteres, *m*: por lo tanto hay que exigir que $m \leq 2^n$
- Buscaremos el menor n entero que verifique: $n \geq log_2 m = 3{,}32 \log m$

  Por ejemplo:
  - ¿cuántos bits necesitamos para representar el código octal?

    $|\alpha| = 8 \rightarrow n \geq \log_2 8 = 3$
  - ¿Y el código hexadecimal?

    $|\alpha| = 16 \rightarrow n \geq \log_2 16 = 4$
  - ¿Y para un código cualquiera con 10 símbolos?

    $|\alpha| = 10 \rightarrow n \geq \log_2 10 = 3{,}32 \log 10 = 3{,}32$

  El menor entero que lo verifica, $n \geq 3{,}32$, es n = 4

Obviamente, si $2^n > m$, hay combinaciones de bits que no se usan.

3.2.3.1. Transformaciones entre bases

- **Transformación de base binaria a decimal**

  Aprovechando la expresión vista antes de N ($N = \sum n_i \cdot b^i$):

  $110100_{/2} = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^2 = 52_{/10}$

  $0{,}10100_{/2} = 1 \cdot 2^{-1} + 1 \cdot 2^{-3} = 0{,}625_{/10}$

  $10100{,}001 = 2^4 + 2^2 + 2^{-3} = 20{,}125_{/10}$

  Simplemente hay que sumar los pesos de las posiciones en las que hay un uno.

- **Transformaciones entre bases decimal y binario**

  Parte entera

  El nuevo número estará formado por los restos de las sucesivas divisiones por dos y el último cociente, teniendo en cuenta que éste será el MSB (*Most Signicative Bit*) y el primer resto será el LSB (*Least Signicative Bit*).

  Ejemplo: $36_{/10}$ =

  Parte fraccionaria

  Se obtiene al multiplicar sucesivamente por dos la parte fraccionaria.

  El nuevo número se forma con las partes enteras que serán siempre 0 ó 1.

  Ejemplo: $0{,}855_{/10}$ =

- **Ejemplo**:

  Obtén la representación en binario de $N = 66{,}125_{/10}$

  Solución = ¿?

- Ambos códigos no han de corresponderse en longitud, lo cual puede llevarnos a un **error de truncamiento** cuando tengamos un **número limitado de posiciones** en las que tengamos que representar el número **y no consigamos un número exacto**. Entonces, nos veremos obligados a detenernos.

- **Ejemplo**: buscad la representación binaria de la expresión decimal 0,6

3.2.4. Operaciones aritméticas con variables binarias

- Una variable binaria puede contener un número binario.

a	b	a + b	a - b	a * b	a / b
0	0	0	0	0	Ind.
0	1	1	1_{-1}	0	0
1	0	1	1	0	∞
1	1	0_{+1}	0	1	1

- Ejemplos

  $1110101 + 1110110_{/2}$=

  $1110001 – 1010111_{/2}$ =

  $1101010 \times 1010_{/2}$ =

  $1001011 / 110_{/2}$ =

3.2.4. Operaciones aritméticas con variables binarias (cont.)

En resumen:

- Multiplicar por dos es añadir un cero a la derecha o mover el punto decimal a la derecha.
- Dividir por dos es mover el punto decimal a la izquierda o eliminar un cero a la derecha.

Ejemplos

- $1101{,}101_{/2} * 10_{/2} = 11011{,}01_{/2}$
- $101{,}101_{/2} * 2^5{}_{/10} = 101{,}101 * 100000_{/2} = 10110100_{/2}$
- $11011_{/2} \div 2_{/10} = 11011 \div 10_{/2} = 1101{,}1_{/2}$
- $11010{,}01_{/2} \div 101{,}1_{/2} = 100{,}11_{/2}$

- Vista la dificultad de la operación resta, que afecta tanto a la resta como a la división, se busca una representación que permita realizar las restas como sumas: la notación en complementos.

3.2.5. Notación y operaciones en complementos

- Permite representar los números negativos.
- Facilita las operaciones de resta.

3.2.5.1. Complemento a la base menos uno de un número, Cb1

$Cb1_{/b}(N)$ el complemento a la base, *b,* menos uno de un número, *N*, se consigue restando cada una de las *n* cifras de N, de *b-1*:

$Cb1_{/b}(N) = b^n – N – 1 = (b^n - 1) - N$

Ejemplo, en la base decimal, b=10, la base menos uno es 9:

$Cb1_{/10}(25) = 99 – 25 = 74_{/10}$

Base 10: Complemento a 9

- $Cb1_{/10}(63) = 99 - 63 = 36_{/10}$
- $Cb1_{/10}(162) = 999 - 162 = 837_{/10}$

Base 2: Complemento a 1

En binario natural, base *b = 2, se habla de* complemento a 1:

- $Cb1_{/2}(10010) = 11111 – 10010 = 01101_{/2}$
- $Cb1_{/2}(101010) = 111111 - 101010 = 010101_{/2}$

3.2.5. Notación y operaciones en complementos (cont.)

Resumen: ***Para obtener el complemento a 1 de un número N en binario natural sólo es necesario cambiar los ceros por unos y viceversa.***

- **Resta en complemento a la base menos uno:** La operación resta se consigue *sumando al minuendo el complemento a la base menos uno del sustraendo*. Si existe un *acarreo, se elimina y se suma al resultado.*

  Minuendo y sustraendo deben tener el mísmo número de cifras.

Ejemplos:

- Complemento a 9:

  $77_{/10} – 63_{/10} = 77_{/10} + Cb1_{/10}(63) = 77 + 36 = \mathbf{1}13 \rightarrow 13 + \mathbf{1} = 14_{/10}$

  $1100_{/10} – 16_{/10} = 1100_{/10} + Cb1_{/10}(0016) = 1110+(9999 - 0016) = 1100 + 9983 = \mathbf{1}1083 \rightarrow 1083 + \mathbf{1} = 1084_{/10}$

- Complemento a 1:

  $1000111_{/2} – 10010_{/2} = 1000111_{/2} + Cb1_{/2}(10010) = 1000111 + (1111111-0010010) = 1000111 + 1101101 = \mathbf{1}0110100 \rightarrow 0110100 + \mathbf{1} = 0110101_{/2}$

  Como comprobación:

  $$\begin{array}{r} 1000111 \\ \underline{- 0010010} \\ 0110101 \end{array}$$

3.2.5. Notación y operaciones en complementos (cont.)

3.2.5.2. Complemento a la base de un número

$Cb_{/b}(N)$ *el complemento a la base, b, de un número, N, se consigue restando cada una de las cifras del* número, de b-1, y sumando uno al resultado:

$Cb_{/b}(N) = b^n - N = b^n - N - 1 + 1 = Cb1_{/b}(N) + 1$

Esto es, se consigue calculando el complemento a la base menos uno, y sumando uno al resultado.

Por ejemplo, en base 10:

$Cb_{/10}(25) = (99 - 25) + 1_{/10} = 74 + 1_{/10} = 75_{/10}$

Complemento a 10, para base decimal

- $Cb_{/10}(63) = (99 - 63) + 1_{/10} = 36 + 1_{/10} = 37_{/10}$
- $Cb_{/10}(162) = (999 - 162) + 1_{/10} = 837 + 1_{/10} = 838_{/10}$

Complemento a 2, para binario natural

En binario natural, base *b = 2,* se habla de complemento a 2:

- $Cb_{/2}(10010) = (11111 - 10010) + 1_{/2} = Cb1_{/2}(10010) + 1 = 01101 + 1_{/2} = 01110_{/2}$
- $Cb_{/2}(101010) = (111111 - 101010) + 1_{/2} = Cb1_{/2}(101010) + ½ = 010101 + 1_{/2} = 010110_{/2}$

3.2.5. Notación y operaciones en complementos (cont.)

Resumen: Para obtener el complemento a 2 de un número N, sólo es necesario cambiar los ceros por unos y viceversa y sumar uno al resultado.

- **Resta en complemento a la base:** La operación resta se consigue sumando al minuendo el complemento a la base del sustraendo. Si existe un acarreo, se desprecia.

  Ambos operandos deben tener el mismo número de cifras.

***Ejemplos*:**

- **Complemento a 10:**

  $77 - 63_{/10} = 77_{/10} + Cb_{/10}(63) = 77_{/10} + Cb1_{/10}(63) + 1_{/10} = 77 + (36 + 1) = \mathbf{1}14 \rightarrow 14_{/10}$

  $1100_{/10} - 16_{/10} = 1100_{/10} + Cb_{/10}(0016) = 1100_{/10} + Cb1(0016_{/10}) + 1_{/10} = 1110 + (9999 - 0016 + 1) =$
  $1100 + (9983 + 1) = \mathbf{1}1084 \rightarrow 1084_{/10}$
- **Complemento a 2:**

  $1000111_{/2} - 10010_{/2} = 1000111_{/2} + Cb_{/2}(10010) = 1000111_{/2} + (Cb1_{/2}(10010) + 1_{/2}) =$
  $1000111 + (1111111 - 0010010 + 1) = 1000111 + 1101101 + 1 = 1000111 + 1101110 =$
  $\mathbf{1}0110101 = 0110101_{/2}$

Sabiendo que es sencillo realizar sumas e inversiones dentro de un ordenador y que los ordenadores son capaces de realizar estas operaciones de forma veloz, se consigue:

- **Realizar las restas como una inversión (cambiar ceros por unos y viceversa) y una suma.**
- **Realizar las multiplicaciones como desplazamientos y sumas.**

3.3. Códigos intermedios de representación

- Habíamos visto en el apartado 3.1. que existía un código de E/S similar al que utilizan los seres humanos, α, y un código interno, β, derivado del binario que utilizan los ordenadores.

  $\{a, b, \ldots, z, A, B, \ldots, Z, 0, \ldots, 9, (,), \ldots, [,]\} \rightarrow \{0,1\}^n$

  $\alpha \rightarrow \beta$

- A mitad de camino entre α y β están los **códigos intermedios**, que son más amigables para el ser humano que la representación interna, pero que están más próximos a ésta que los códigos de E/S.
  - Generalmente los códigos de E/S se presentan mediante algún código intermedio.
  - Es inmediato pasar de los códigos intermedios a β, que será algún tipo de representación binaria.
  - Las representaciones intermedias utilizan bases potencias de 2: 8 ó 16.

3.3.1. Códigos intermedios: octal y hexadecimal

3.3.1.1. Código octal

- $b = 8 = 2^3$
- $B = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- Codificación: ¿Cuántos bits son necesarios para representar 8 caracteres distintos?

  Respuesta = 3
- Ejemplos
  - $71_{/8}$ es un número octal válido = $57_{/10}$
  - $87_{/8}$ no es un número octal válido, ¿por qué?

Equivalencias

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

Transformaciones entre bases

- **Transformación de base octal a decimal**

  Aprovechando la expresión ya vista: $N = \sum n_i \cdot b^i$ *, para b=8*

  – $55_{/8} = 5 * 8^1 + 5 * 8^0 = 45_{/10}$

  – $0{,}42_{/8} = 4 * 8^{-1} + 2 * 8^{-2} = 0{,}53125_{/10}$

  También podemos substituir las cifras octales por su equivalente binario:

  – $55_{/8} = 101101_{/2} = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45_{/10}$

  – $0{,}42_{/8} = 0{,}100010_{/2} = 2^{-1} + 2^{-5}{}_{/10} = 0{,}5 + 0{,}03125_{/10} = 0{,}53125_{/10}$

- **Transformaciones entre base decimal y octal**

  Parte entera: El nuevo número estará formado por los restos de las sucesivas divisiones por ocho y el último cociente (que será el MSB, mientras que el primer resto será el LSB).

  Parte fraccionaria: Se obtiene al multiplicar sucesivamente por ocho la parte fraccionaria. El nuevo número se forma con las partes enteras que serán siempre una cifra en octal.

  Ejemplo: $2001{,}2910_{/10} = 3721{,}2247_{/8}$

Transformaciones entre bases

- **Transformación de base binaria y octal**

  Existen dos formas distintas de hacerla:

  1. Binario → Decimal → Octal
     - De binario a Decimal, $N = \sum n_i \cdot b^i$, *con b=2*
     - Divisiones por ocho de la parte entera
     - Multiplicaciones por ocho de la parte fraccionaria

     Ejemplo:

     $101010_{/2} = 2^5 + 2^3 + 2^1 = 42_{/10} = 52_{/8}$

  2. Agrupando las cifras binarias en bloques de tres y convirtiéndolo en su equivalente octal, ya que $2^3 = 8$

     Ejemplo:

     $\underline{101}\ \underline{010}_{/2} = 52_{/8} = 5 * 8^1 + 2 * 8^0 = 42_{/10}$

     5 2

Transformaciones entre bases

- **Transformación de base octal a binaria**

  Caso inverso del segundo método del apartado anterior: desagrupar cada cifra en octal por su equivalente de tres cifras binario.

  Ejemplos:

  $537{,}24_{/8} = 101011111{,}010100_{/2}$

  $175{,}22_{/8} = 001111101{,}010010_{/2}$

  También se puede hacer utilizando la transformación a base decimal como paso intermedio.

  Los números en notación octal también se representan como $N_{/o}$. Ejemplo: $N = 854_{/8} = 854_{/o}$

3.3.1. Códigos intermedios: octal y hexadecimal

3.3.1.2. Código hexadecimal

- b = 16 = 2^4
- B = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}
- Codificación: ¿Cuántos bits son necesarios para representar 16 caracteres distintos?

  Respuesta = 4
- Ejemplos: ¿cuáles de estos números son válidos en hexadecimal y si lo son cuál es su equivalente decimal?
  - $1A_{/H}$
  - $F3_{/H}$
  - $G2_{/H}$
  - $CBA_{/H}$
  - $ACB_{/H}$

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

Transformaciones entre bases

- **Transformación de base hexadecimal a decimal**

  Aprovechando la expresión ya vista: $N = \sum n_i \cdot b^i$, para b=16
  - $ABC_{/H} = 10*16^2 + 11*16^1 + 12*16^0 = 2748_{/10}$
  - $11,AB_{/H} = 1*16^1 + 1*16^0 + 10*16^{-1} + 11*16^{-2} = ¿?_{/10}$
  - $1AB,2CD_{/H} = ¿?_{/10}$

  También podemos substituir las cifras hexadecimales por su equivalente binario (de 4 bits):
  - $ABC_{/16} = 101010111100_{/2}$
- **Transformaciones entre base decimal y hexadecimal**

  Parte entera: El nuevo número estará formado por los restos de las sucesivas divisiones por dieciséis y el último cociente (que será el MSB, mientras que el primer resto será el LSB).

  Parte fraccionaria: Se obtiene al multiplicar sucesivamente por dieciséis la parte fraccionaria. El nuevo número se forma con las partes enteras que serán siempre una cifra en hexadecimal.

  Ejemplo: $2008,2710_{/10} = ¿?$

Transformaciones entre bases

- **Transformación de base binaria y hexadecimal**

  Existen dos formas distintas de hacerla:

  1. Binario → Decimal → Hexadecimal
     - De binario a Decimal, *$N = \sum n_i \cdot b^i$, con b=2*
     - Divisiones por dieciséis de la parte entera
     - Multiplicaciones por dieciséis de la parte fraccionaria

     Ejemplo:

     $10101010_{/2} = 2^7 + 2^5 + 2^3 + 2^1 = 170_{/10} = AA_{/H}$
  2. Agrupando las cifras binarias en bloques de cuatro y convirtiéndolo en su equivalente hexadecimal, ya que $2^4 = 16$. Se hace a ambos lados de la coma decimal.

     Ejemplo:

     $\underline{1010}\ \underline{1010}_{/2} = AA_{/H} = 10 * 16^1 + 10 * 16^0 = 160 + 10 = 170_{/10}$

     A A

Transformaciones entre bases

- **Transformación de base hexadecimal a binaria**

  Caso inverso del segundo método del apartado anterior: desagrupar cada cifra en hexadecimal por su equivalente de cuatro cifras binario.

  Ejemplos:

  $F3DA_{/H} = 1111001111011010_{/2}$

  $CBA08_{/H} = 11001011101000001000_{/2}$

  También se puede hacer utilizando la transformación a base decimal como paso intermedio.

- Resumiendo: Teniendo en cuenta todos estos conceptos, ¿cuál sería el método más rápido para pasar el número 16275 a binario natural?

3.3.2. Códigos de E/S

- **Se utilizan códigos normalizados**
- **BCD: Binario Codificado en Decimal (Prieto, Lloris y Torres, Cap. 3)**
  - n = 6; se pueden codificar hasta *m = 64 caracteres.*
  - Puede llevar un séptimo bit como bit de vericación.
  - Disposición de los bits: $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

  Donde:
  - b_6 → *bit opcional de verificación (véase 3.3.3)*
  - $b_5\ b_4$ → *Bits de zona* → *00 para los números.*
  - $b_3\ b_2\ b_1\ b_0$ → *bits de posición: coinciden con los* números del 1 al 9 en notación binaria y el 0 como el 10., de ahí su nombre
  - No se representan las letras minúsculas.

Ejemplo:

Carácter	**BCD**	**Representación interna con 7 bits**
'0'	$112_{/o}$	1 0 0 1 0 1 0
'9'	$111_{/o}$	1 0 0 1 0 0 1
'A'	$061_{/o}$	0 1 1 0 0 0 1
'('	$034_{/o}$	0 0 1 1 1 0 0

- **EBCDIC: Código BCD extendido de Intercambio Normalizado**
  - Representación interna utilizada por IBM.
  - *n = 8; se pueden codificar hasta m = 256 caracteres.*
  - Disposición de los bits: $b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7$

  Donde:
  - $b_0\ b_1$*: marcan los distintos bloques del código:*
    - *11: cifras y mayúsculas,*
    - *10: minúsculas,*
    - *01: caracteres especiales,*
    - *00: caracteres aún no usados y caracteres de* control.
  - $b_2\ b_3\ b_4\ b_5\ b_6\ b_7$ *: se denominan bits de posición. Por ejemplo,* dentro de estos, los bits *b2 b3 para b0 b1 = 11:*
    - *00: A – I*
    - *01: J – R*
    - *10: S – Z*
    - *11: Numérico*

Ejemplo:

Carácter	**EBCDIC**	**Representación interna**
'0'	$F0_{/H}$	1 1 1 1 0 0 0 0
'9'	$F9_{/H}$	1 1 1 1 1 0 0 1
'A'	$C1_{/H}$	1 1 0 0 0 0 0 1
'('	$4D_{/H}$	0 1 0 0 1 1 0 1

- **ASCII: American Standard Code for Information Interchange o Código Normalizado Americano para Intercambio de Información**
  - *n = 7, y opcionalmente un bit adicional de verificación.*
  - Por lo tanto, puede representar hasta *m = 128 caracteres* distintos.
  - Distribución de los bits: $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

  Ejemplo:

Carácter	**ASCII 7 bits**	**Representación interna**
'0'	$060_{/o}$	0 1 1 0 0 0 0
'9'	$071_{/o}$	0 1 1 1 0 0 1
'A'	$101_{/o}$	1 0 0 0 0 0 1
'('	$050_{/o}$	0 1 0 1 0 0 0

El código ASCII es el más utilizado, y al que suelen hacer referencia todos los manuales de programación.

3.3.3. Detección de errores en la información codificada

$$\{a, b,\ldots,z, A, B,\ldots, Z, 0,\ldots, 9,(,),\ldots,[,]\} \rightarrow \{0,1\}^n$$
$$\alpha \rightarrow \beta$$

- Para cada alfabeto tenemos: $|\alpha| = m$ y para un *n* dado tenemos $|\beta| = 2^n$. En general, $m \neq 2^n$, pero necesitamos representar todos los caracteres, *m*: por lo tanto hay que exigir que $m \leq 2^n$ ¿Qué ocurre cuando $2^n > m$?

Eficiencia de un código: $\tau = m / 2^n$ $\quad 0 \leq \tau \leq 1$

– Proporción entre los símbolos que se representan y los que se podrían representar con n 0's/1's

– Un código es más eficiente cuantas menos combinaciones de bits desperdicie.

Ejemplo: Código ASCII, 95 símbolos

– Sin bit de verificación, $n = 7$: $m = 95$; $2^n = 128$; $\tau = 95/128 = 0{,}742$

– Con bit de verificación, $n = 8$: $m = 95$; $2^n = 256$; $\tau = 95/256 = 0{,}371$

Redundancia: $R = (1-\tau) * 100\%$

– La redundancia está inversamente relacionada con la eficiencia. Un código poco eficiente será redundante y viceversa.

Ejemplo: Código ASCII, 95 símbolos

– Sin bit de verificación, $n = 7$: $R = 25{,}8\%$

– Con bit de verificación, $n = 8$: $R = 62{,}9\ \%$

- **Bit de verificación**

– Un código redundante puede utilizarse para detectar errores al transmitir información.

- ¿Qué ocurre cuando transmitimos sobre un alfabeto de 8 símbolos y sólo utilizamos *3 bits?*
- ¿ Qué ocurre cuando transmitimos sobre un alfabeto de 10 símbolos y sólo utilizamos *4 bits?*

Bits de verificación: Son los bits redundantes que se añaden mediante algún algoritmo previamente fijado y que permiten encontrar errores en la transmisión, si dicho algoritmo se aplica en ambos extremos de la comunicación.

Es posible aplicar distintos algoritmos:

– **Bit de paridad**

- Añade al comienzo de cada secuencia un cero o un uno según alguno de los criterios siguientes:
  – *Paridad par*: el bit (cero o uno) hace que el número de unos sea par.
  – *Paridad impar*: el bit (cero o uno) hace que el número de unos sea impar.

- En el otro extremo del canal de comunicación, si se produce un error en la información se detectará ya que fallará la paridad.

Ejemplo: BCD utilizando criterios de paridad par e impar.

Mensaje Enviado	*Enviado con Paridad Par*	*Enviado con Paridad Impar*
1 000 001	**0**1 000 001	**1**1 000 001
1 011 011	**1**1 011 011	**0**1 011 011
1 010 000	**0**1 010 000	**1**1 010 000
1 101 000	**1**1 101 000	**0**1 101 000

– ¿Qué ocurre si falla un número par de bits?

Existen métodos basados en bit de paridad mucho más elaborados.

– **Verificación en cuenta fija**

- Cada símbolo o carácter enviado tiene un número fijo de ceros y unos.

Con ocho bits podemos exigir que haya cuatro ceros y cuatro unos.

- Para detectar el error se cuenta el número de ceros y de unos y debe coincidir.
- Ejemplo: Alfabeto = {A, B, C, D, E, F}, n = 4. Si usamos verificación en cuenta fija (2 ceros y 2 unos):

Símbolo	*Código*	*Símbolo*	*Código*
'A'	0011	'B'	0101
'C'	1001	'D'	0110
'E'	1010	'F'	1100

3.4.Representación Interna de la Información

3.4.1. Introducción

Hemos visto los códigos de E/S, pero esta representación no es eficiente dentro del ordenador.

– Habría que realizar una traducción del código E/S a la representación interna.

– Internamente se utiliza alguna variante del binario natural, que es mucho más compacta y fácil de operar que la traducción directa del código E/S:

Ej.: $N=253_{/10} = 062\ 065\ 063_{/o} = 0\ 110\ 010\ 0\ 110\ 101\ 0\ 110\ 011_{/2} = 11111101_{/2}$

– ¿Cómo se almacena internamente la información?

Anticipo (se tratará en más profundidad en el tema siguiente) :

- la información se agrupa internamente (en la ALU y en la CPU) en función de la longitud de palabra (número de bits) que procese el ordenador (por unidad de tiempo).
- El tamaño del código de representación interna será un múltiplo de la longitud de palabra, que suele ser de 8, 16, 32 ó 64 bits.
- Los distintos tipos de datos que se pueden almacenar en un ordenador tendrán distintas representaciones. El objetivo es agilizar los cálculos, ya que si se almacenasen directamente mediante su código de E/S se perdería la representación posicional de los números.

3.4.Representación Interna de la Información

3.4.2. Lógicos, caracteres y complejos

En los lenguajes de programación suele haber distintos tipos de datos básicos: enteros, reales, lógicos, caracteres y/o complejos. Estos tres últimos suelen representarse en función de los dos primeros.

Lógicos: Representarán el valor cierto (.T.) o falso (.F.), o bien una variable booleana (0 ó 1). Suele representarse como un caso especial de un entero: falso = 0 y cierto = 1.

Caracteres: Se representa su código de E/S (ASCII) directamente sobre palabras del ordenador.

Un conjunto de caracteres ocupará una colección de palabras consecutivas en la memoria (se puede representar más de un carácter por palabra).

Complejos de simple y de doble precisión: es el caso más sencillo de datos estructurados; internamente no se representa la i, sino que se almacenan **como dos números reales** de simple o doble precisión consecutivos; el procesador, en función del tipo, sabrá cómo interpretar esa información.

3.4.Representación Interna de la Información (cont.)

3.4.3. Representación interna de los datos de tipo entero

Fundamentos:

- se dispone de *n* posiciones, para el signo y el valor, en función de la longitud de palabra;
- la notación es $\underset{msb}{\underline{a_{n-1}}}\ a_{n-2} \ldots a_1 \underset{lsb}{\underline{a_0}}$

- Existen dos alternativas de representación:
  1. **Enteros sin signo**
     - n bits para representar el valor absoluto
- Ejemplo: para n = 8, el rango de valores que se puede representar: $0 \le N \le 2^{n-1}$

  $00000000_{/2} = 0_{/10}$

  $00000001_{/2} = 1_{/10}$

  ...

  $11111110_{/2} = 254_{/10}$

  $11111111_{/2} = 255_{/10}$
  2. **Enteros con signo**
     - El significado de los bits ($a_{n-1}a_{n-2}\ldots a_1a_0$) varía en función de la representación en:

3.4.Representación Interna de la Información (cont.)

- **Signo y magnitud:**
  - a_{n-1} *indica el signo: si es 1 entonces* $N \le 0$; si es 0, $N \ge 0$
  - Tanto si a_{n-1} *es 0, como si es 1,* $a_{n-2}, \ldots, a_1, a_0$ representan el valor absoluto del número en binario natural
  - Rango: $-2^{n-1}+1 \le N \le 2^{n-1}-1$
  - El cero tiene dos representaciones: ±0 (se pierde una opción de representación)
- **Complemento a 1:**
  - a_{n-1} *indica el signo, si es 1 entonces* $N \le 0$, si es 0, $N \ge 0$
  - si a_{n-1} *es 0:* $a_{n-2}, \ldots, a_1, a_0$ representan el valor absoluto del número en binario natural
  - si es 1 entonces $N \le 0$, $a_{n-2}, \ldots, a_1, a_0$ representan al número en complemento a 1
  - Rango: $-2^{n-1}+1 \le N \le 2^{n-1}-1$
  - El cero tiene dos representaciones: ±0 (se pierde una opción de representación)
- **Complemento a 2:**
  - Idem al anterior. Pero si es 1 entonces $N < 0$, $a_{n-2}, \ldots, a_1, a_0$ representan al número en complemento a 2
  - Rango: $-2^{n-1} \le N \le 2^{n-1}-1$
  - El cero tiene una única representación

3.4.Representación Interna de la Información (cont.)

- **Notación sesgada:**
  - el número que se representa siempre es positivo; pasamos de una representación de $-2^{n-1}+1 \le N \le 2^{n-1}-1$ a otra $0 \le M \le 2^{n-1}$, en la que los números negativos van antes que el cero o los números positivos en binario natural.
  - Esta notación es muy ventajosa para comparar dos números, sin tener en cuenta su signo.
  - Para n bits, se establece el sesgo, de forma genérica, como: $S = 2^{n-1}$
  - Internamente se representa M = N + S
  - Rango: $0 \le M \le 2^{n-1}$

    $-2^{n-1} \le N \le 2^{n-1}-1$
  - Ejemplo, para n = 4 bits:

    $S = 2^{n-1} = 2^{4-1} = 2^3 = 8$

    $N_1 = +6_{/10} \rightarrow M_1 = N_1 + S = 6 + 8 = 14_{/10} = 1110_{/s}$

    $N_2 = -5_{/10} \rightarrow M_2 = N_2 + S = -5 + 8 = 3_{/10} = 0011_{/s}$

Resumen

Signo y magnitud	Complemento a 1	Complemento a 2	Notación sesgada
0111 = 7	0111 = 7	0111 = 7	1111 = 7
0110 = 6	0110 = 6	0110 = 6	1110 = 6
0101 = 5	0101 = 5	0101 = 5	1101 = 5
0100 = 4	0100 = 4	0100 = 4	1100 = 4
0011 = 3	0011 = 3	0011 = 3	1011 = 3
0010 = 2	0010 = 2	0010 = 2	1010 = 2
0001 = 1	0001 = 1	0001 = 1	1001 = 1
0000 = + 0	0000 = + 0	0000 = 0	1000 = 0
1000 = -0	1111 = - 0	1111 = - 1	0111 = -1
1001 = -1	1110 = -1	1110 = -2	0110 = -2
1010 = -2	1101 =-2	1101 =-3	0101 = -3
1011 = -3	1100 = -3	1100 = -4	0100 = -4
1100 = -4	1011 = -4	1011 = -5	0011 = -5
1101 = -5	1010 = -5	1010 = -6	0010 = -6
1110 = -6	1001 = -6	1001 = -7	0001 = -7
1111 = -7	1000 = -7	1000 = -8	0000 = -8

- **Ejercicio: Represéntense en las cuatro notaciones los números 1492 y -2008, sabiendo que se dispone de una precisión de 2 bytes.**

3.4.Representación Interna de la Información (cont.)

- La representación más utilizada es el complemento a dos, dada su facilidad para realizar las restas.

  N_{min} y N_{max} para distintas longitudes de palabra:

		N_{min}	
Precisión	N_{max} $2^{n-1}-1$	**Complemento a 1** $-2^{n-1}+1$	**Complemento a 2** -2^{n-1}
n = 8 bits	127	-127	-128
n = 16 bits	32767	-32767	-32768
n = 32 bits	2147483649	-2147483649	-2147483650
n = 64 bits	$9{,}22337 \cdot 10^{18}$	$-9{,}223372 \cdot 10^{18}$	$-9{,}223372 \cdot 10^{18}$

- Existen otras formas de representar enteros:
  - Representación de enteros en BCD
    - BCD empaquetada: dos dígitos decimales por byte.
    - BCD desempaquetada: un dígito decimal por byte.
  - En este tipo de representaciones se suele utilizar un byte para el signo (0000 para números positivos y 0001 para números negativos).

3.4.4. Los números reales: la notación IEEE 754

- Al representar números reales podemos tener problemas de:
  - precisión: ¿Cómo representar π?
  - magnitud: ¿Cómo representar los números reales (especialmente aquellos que son muy grandes o muy pequeños)?
    - Por ejemplo: $C = 2{,}9979 * 10^8$ *m/s*
    - Carga fundamental = 1, $602 * 10^{-19}$ *C*
- **Notación exponencial, científica o en coma flotante**

  Cualquier número se puede representar de la forma:

  $N = M * B^E$

  donde *N es el número, M es la mantisa, B es la base* y *E es el exponente. Esta representación puede modific*arse, conservando el valor de *N, si se reajustan* adecuadamente *M y E.*

  $13257{,}3285 = 13257{,}3285*10^0 = 1{,}32573285*10^4 = 0{,}132573285*10^5 = 132573285*10^{-4}$

3.4.4. Los números reales: la notación IEEE 754

Como hemos visto en el ejemplo anterior:

- Aumentar en una unidad el valor de E supone dividir M por B:

  $123{,}456 = 123{,}456*10^0 = 12{,}3456*10^1 = 1{,}23456*10^2$
- Disminuir en una unidad el valor de E supone multiplicar M por B:

  $123{,}456 = 123{,}456 * 10^0 = 1234.56 * 10^{-1} = 12345{,}6*10^{-2}$

Estas tareas, dentro de un ordenador, ya vienen integradas con los nuevos procesadores. Antes, había que realizarla con los co-procesadores aritméticos. Si no se dispusiese de un co-procesador, sería necesario realizarlo via software.

También puede ser necesario realizar estas operaciones mediante un programa si queremos disponer de una precisión mayor de la que realmente tiene nuestro ordenador.

Al principio cada fabricante tenía su propia notación. Entre 1977-1985 se crea la norma IEEE-754, actualmente aceptada. Esto facilita el desarrollo de software aritmético genérico.

3.4.4.1. Representación interna de reales: notación IEEE 754

- *E* ha de ser entero,
- *B*=2 siempre, por lo tanto se puede omitir
- Sólo es necesario almacenar *M* y *E* con sus respectivos signos

s	e	m

Longitud de los campos:

$|s| = 1$,
$|e| = n_e$,
$|m| = n_m$,
$n = 1 + n_e + n_m$

Pueden utilizarse los algoritmos de comparación de enteros para comparar números reales.

- Signo
  - s = 0 para los positivos; s = 1 para los negativos
- Exponente
  - Entero en notación sesgada, con sesgo $S = 2^{ne-1}-1$
  - Si $N = M \cdot B^E$ → $e = E + S$
  - Ejemplo: $n_e = 8$ → $S = 2^{8-1}-1 = 2^7 - 1 = 127$

3.4.4.1. Representación interna de reales: notación IEEE 754

- Mantisa **Normalizada y Empaquetada**
  - Normalizada: Una mantisa está normalizada cuando su bit = 1 más significativo se encuentra en la posición de las unidades. En caso contrario se dice que la mantisa está *denormalizada*.
  - Empaquetada: Se dice que la mantisa está empaquetada cuando sólo se almacena la parte fraccionaria del número normalizado.
- La mantisa normalizada tendrá la forma *M = 1.m*
- Si la mantisa está empaquetada sólo será necesario almacenar *m.*
- Se evita perder bits signicativos cuando se realicen varias operaciones consecutivas. Además, siempre se tendrán n_m bits signicativos.
- La unidad aritmético lógica (ALU) tiene que desempaquetar de la representación interna antes de realizar las operaciones. También tiene que empaquetar el resultado antes de volver a guardarlo en la memoria del ordenador.

3.4.4.1. Representación interna de reales: notación IEEE 754

- Mantisa **Normalizada y Empaquetada**

Ejemplo: Obtener la representación interna de las mantisas de los siguientes números teniendo en cuenta que $n_m = 12$: $N_1 = 1101{,}11011011 \cdot 2^{-7}$ y $N_2 = 0{,}00011000110 \cdot 2^{24}$

N	N con m normalizada	m empaquetada
$1101{,}11011011 \cdot 2^{-7}$	$1{,}10111011011 \cdot 2^{-4}$	101110110110
$0{,}00011000110 \cdot 2^{24}$	$1{,}1000110 \cdot 2^{20}$	100011000000

Casos Especiales

s	e	m

- $e = e_{min} = 0$ y $m = 0$ → $N = 0$
- $e = e_{min} = 0$ y $m \neq 0$ → la mantisa se almacena denormalizada y $S = 2^{ne-1}$ → $E = -2^{ne-1}+2$
- $e = e_{max} = 11\ldots1$ y $m = 0$ → $N = +/- \infty$
- $e = e_{max} = 11\ldots1$ y $m \neq 0$ → N = NaN: Not a Number → indeterminada

3.4.4.2. Precisión y redondeo

- Es difícil representar un número de forma exacta con n_m *+1 bits.*
- Modelo de redondeo: **Redondeo al par**
  - Considera el bit menos significativo, que está en la posición $-n_m$ y los bits que ocupan las posiciones $-n_m-1$ y $-n_m-2$ del número a redondear, que se denominan bit de redondeo y bit retenedor, respectivamente.
  - El valor del bit de redondeo es el que se obtiene al realizar la operación.
  - El valor del bit retenedor es 1 si alguna de los bits en las posiciones $-n_m-2$, $-n_m-3$,… es 1. En caso contrario, vale 0.
    - Si el bit de redondeo es 0, se trunca.
    - Si el bit de redondeo es 1 y bit retenedor es 0, sumo 1 al bit en n_m, *salvo que ya sea par* ($m_{n_m} = 0$).
    - Si el bit de redondeo es 1 y el bit retenedor es 1, se suma 1 al bit en la posición n_m.

- **Ejemplos de redondeo**

Resultado en la ALU	Acción	Mantisa Redondeada
1.01101 | 10	Sumar 1	1.01110
1.01100 | 10	Truncar	1.01100
1.01100 | 11	Sumar 1	1.01101
1.01100 | 00	Truncar	1.01100

3.4.4.3. Problemas de precisión

- Hay que prestar atenresolución parcial de las ayudas de bolsas de viaje ción a tres casos:
  - Underflow o desbordamiento al cero.
  - Overflow o desbordamiento.
  - Comparación de dos números reales muy próximos.
- Los dos primeros pueden provocar que un número no pueda representarse en nuestro ordenador.
- El tercero puede provocar que concluyamos que dos números son iguales cuando realmente no lo son.

Resumen diferentes precisiones

- Los números reales podrán representarse, al menos, con dos precisiones: simple y doble

	n	n_m	n_e	S	E_{min}	E_{max}
Simple	32	23	8	127	-126	127
Doble	64	52	11	1023	-1022	1023

- Resumiendo los casos anteriores con simple precisión:

e	m	N
e =255	m ≠ 0	N = NaN
e =255	m = 0	$N= (-1)^S \cdot \infty$
0 < e < 255		$N= (-1)^S \cdot 2^{e-127}[1.m]$
e = 0	m ≠ 0	$N= (-1)^S \cdot 2^{e-126}[0.m]$
e = 0	m = 0	$N= (-1)^S \cdot 0$

- Resumiendo los casos anteriores con doble precisión:

e	m	N
e =2047	m ≠ 0	N = NaN
e =2047	m = 0	$N= (-1)^S \cdot \infty$
0 < e < 2047		$N= (-1)^S \cdot 2^{e-1023}[1.m]$
e = 0	m ≠ 0	$N= (-1)^S \cdot 2^{e-1022}[0.m]$
e = 0	m = 0	$N= (-1)^S \cdot 0$

- Formas de almacenar los números en las palabras de la memoria:
  - Palabras de 16 bits: simple (32) y doble precisión (64 bits)

s	e (8 bits)	m (7 bits)
m (16 bits)		

s	e (11 bits)	m (4 bits)
m (16 bits)		
m (16 bits)		
m (16 bits)		

- Formas de almacenar los números en las palabras de la memoria:
  - Palabras de 32 bits: simple (32) y doble precisión (64 bits)

s	e (8 bits)	m (23 bits)

s	e (11 bits)	m (20 bits)
m (32 bits)		