ESCUELA DE INGENIERÍAS INDUSTRIALES

T2
REPRESENTACIÓN DE LA INFORMACIÓN

Fundamentos de Informática

Departamento de Ingeniería de Sistemas y Automática. EII.

Universidad de Valladolid

Índice

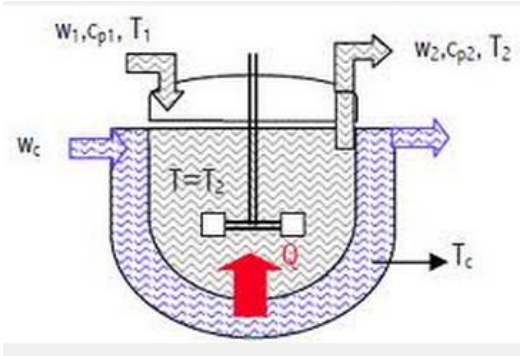
1. Introducción
2. Sistemas de numeración
3. Representación de números enteros.
4. Aritmética de enteros.
5. Representación de números reales.
6. Desbordamiento de la representación
7. Representación de caracteres.
8. Conversión entre tipos de datos

Introducción

La **información** es un conjunto de datos, cuyo significado depende del contexto. Su propósito puede ser el de reducir la incertidumbre o incrementar el conocimiento acerca de algo.

- Nombre
- Edad
- Sexo
- DNI
- NIA
- Altura, peso

- No. de serie
- Potencia
- Eficiencia
- Par motor
- Modelo



- Dimensiones
- Composición química de reactivos
- Caudales
- Caudal del refrigerante.
- Potencia del agitador
- Temperatura

Introducción

La **representación de la información** estudia las distintas formas en que se puede comunicar y acceder a la información.

Para la ejecución de un programa, al ordenador debemos suministrarle dos tipos de información:

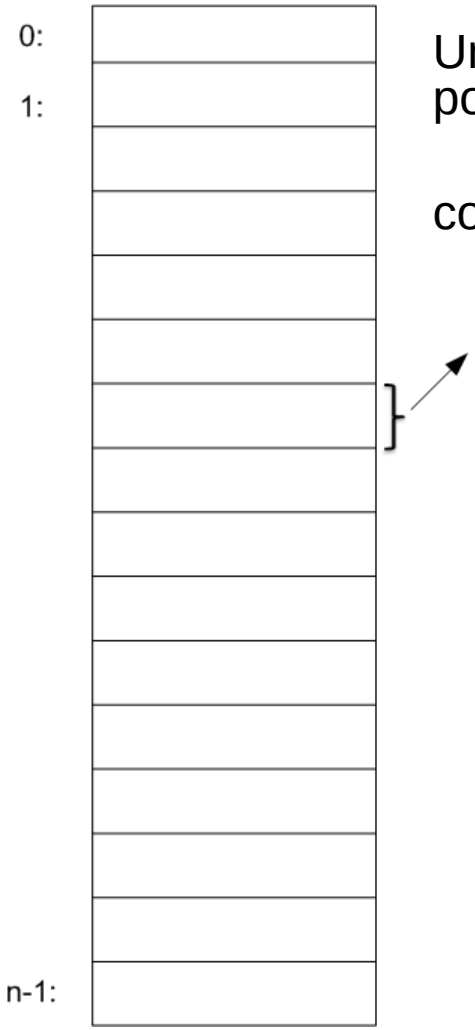
- las **instrucciones** que forman el programa
- los **datos** con los que debe operar ese programa.

Por razones de eficiencia, tanto las instrucciones como los datos asociados a un programa son patrones de **bits** que coexisten en la memoria del ordenador.

Introducción

¿Cómo almacena el ordenador la información?

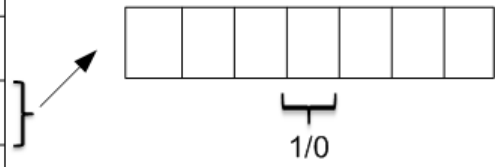
- La memoria del ordenador es una tabla de **n** octetos o **bytes**. Cada **byte** está compuesto de **8** bits.
- El **bit** es la unidad de información.



Un **bit** sólo distingue entre dos posibilidades:

carga/no carga

convención: 1 / 0



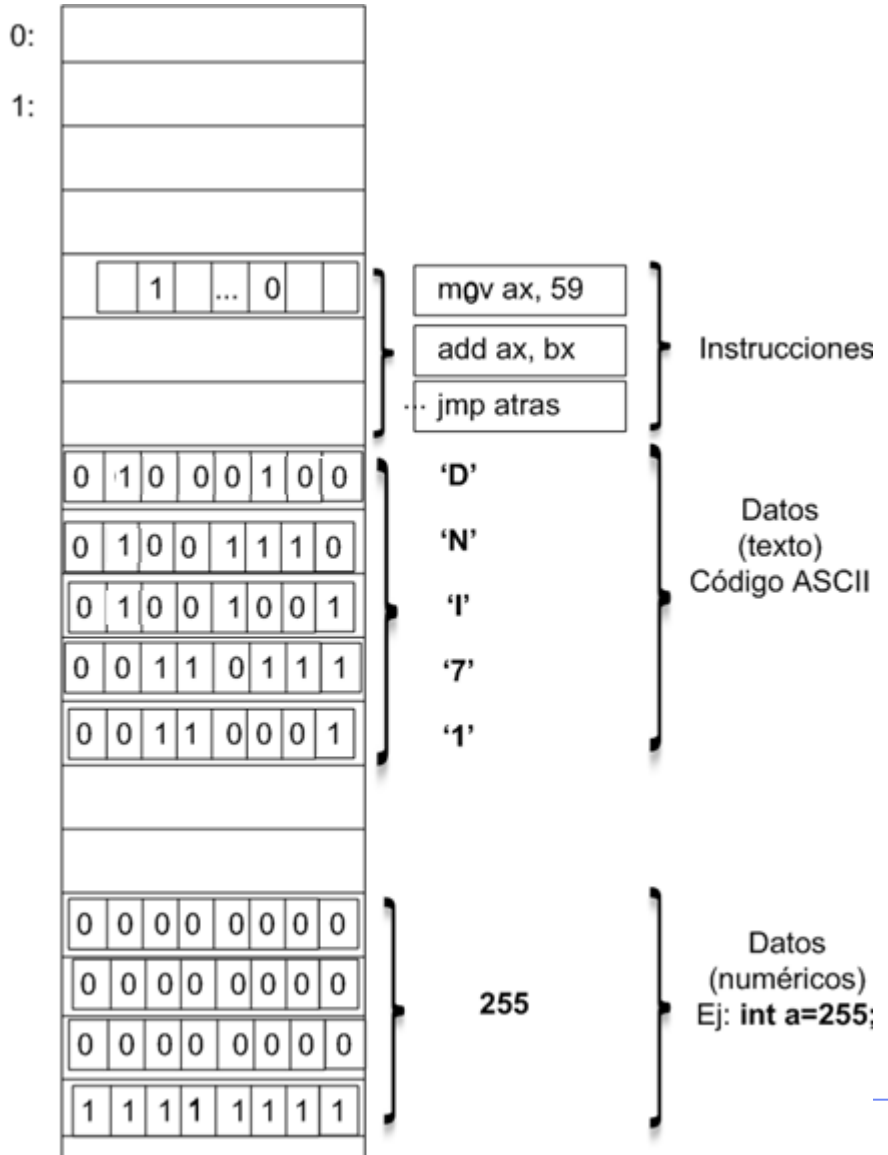
Esta es la solución más robusta y eficiente.

Introducción

¿Cómo almacena el ordenador la información?

En la zona de memoria donde se almacena un programa conviven:

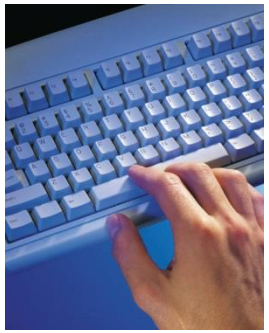
- **Instrucciones**
- **Datos**: texto, valores numéricos (enteros, reales,…)



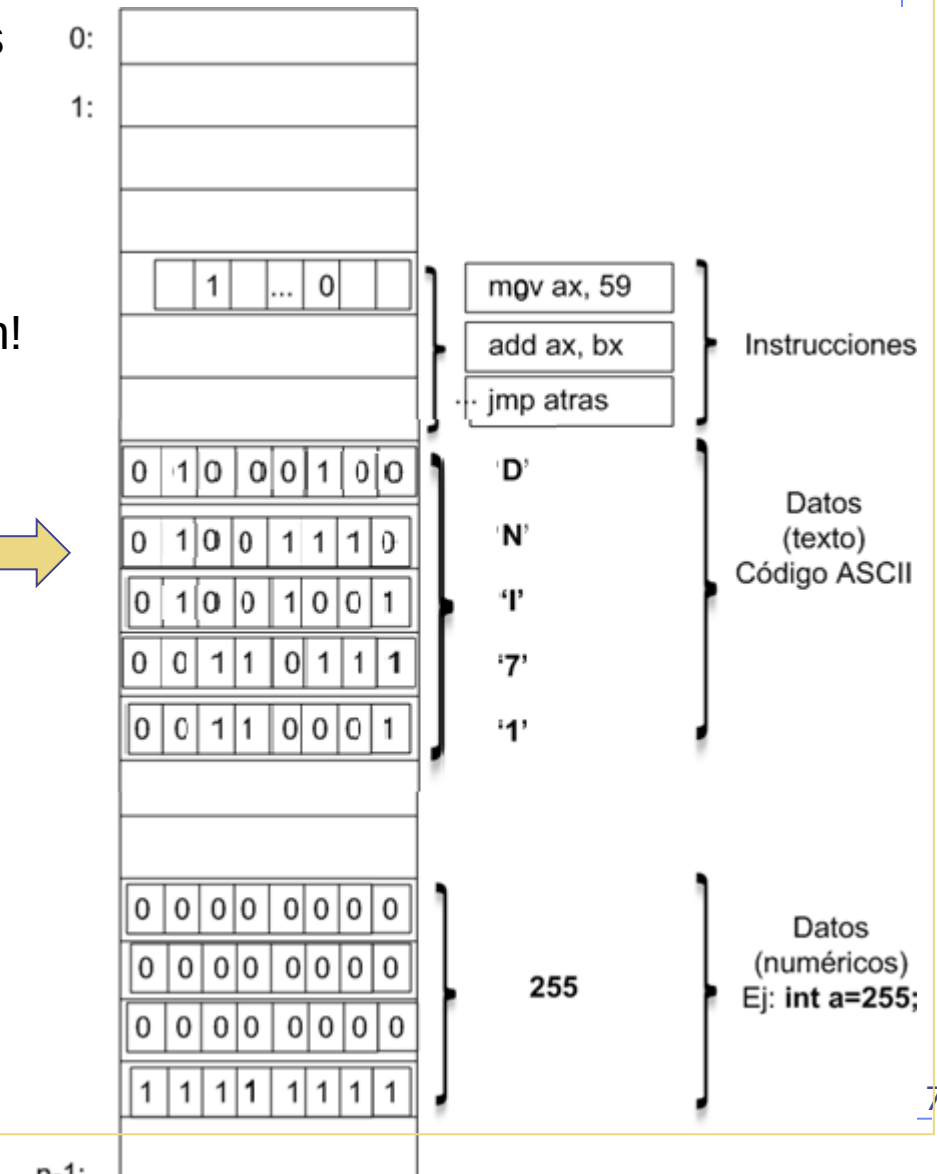
Introducción

¿Cómo suministramos la información?

Por ejemplo, cuando introducimos datos vía teclado, trabajamos con caracteres: dígitos, letras, signos de puntuación…, pero el ordenador trabaja con 0's y 1's. ¡tendremos que **codificar** la información!



Codificación



Introducción

La codificación de la información

La **codificación** es una transformación que representa los elementos de un conjunto mediante los de otro, de tal forma que a cada elemento del primer conjunto le corresponda un elemento distinto del segundo.

- Ejemplo:
  - código de provincia en las matrículas de los coches;
  - NIA
  - DNI

Introducción

La codificación de la información

La existencia de dos conjuntos diferentes de caracteres, uno externo para el usuario y otro interno para el ordenador, hace necesario **codificar** los caracteres de un conjunto con los del otro. Son los llamados **códigos de E/S** (por ejemplo el **código ACSII**).

Por otro lado, a nivel interno y para mejorar la eficiencia, la existencia de datos numéricos de muy diferentes características (binarios, enteros, coma flotante), motiva la utilización de múltiples representaciones, dependiendo del número bits utilizados, si los valores tienen signo, sin son enteros o fraccionarios, etc.

Introducción

La codificación de la información

Un aspecto fundamental en la codificación es el número de valores diferentes que podemos codificar a partir de una cantidad fija de $\boldsymbol{n}$ bits.

¿Cuántos $\boldsymbol{m}$ valores diferentes con $\boldsymbol{n}$ bits?

$\boldsymbol{m} = 2^{\boldsymbol{n}}$

Ejemplo: Con $\mathbf{10}$ bits podemos representar $\mathbf{1024} = 2^{\mathbf{10}}$ valores diferentes

Si necesito codificar $\boldsymbol{m}$ valores, ¿cuál es el número mínimo $\boldsymbol{n}$ de bits necesarios?

$\boldsymbol{n} = \log_2 \boldsymbol{m}$

Dado que $\boldsymbol{n}$ puede ser un valor no entero, tomaríamos el entero más próximo por exceso.

Ejemplo: Tenemos un almacén con $\mathbf{17524}$ contenedores y queremos identificarlos mediante una pegatina con un código binario. ¿Cuántos bits necesitamos?

$\log_2 \mathbf{17524} \approx \mathbf{14.097}$ Por tanto, la solución es $\boldsymbol{n} = \mathbf{15}$.

Sistemas de numeración

Un **sistema de numeración** es una colección de símbolos utilizado para representar *cantidades*, junto al conjunto de reglas que permiten construir números válidos.

Sistemas de numeración posicional

El número de símbolos (cifras) permitidos en un sistema de numeración posicional se conoce como **base** del sistema de numeración.

Por ejemplo en el sistema decimal la base es *b=10*.

El conjunto de 10 símbolos es: *{0,1,2,3,4,5,6,7,8,9}*

Un número se expresará mediante una secuencia de cifras, contribuyendo cada una de ellas con un valor dependiente de:

- El **valor** que representa esa cifra
- La **posición** que ocupe dentro del número

El **sistema de numeración romano** es un ejemplo de sistema de numeración **no posicional**.

Sistemas de numeración

Para construir un número N en la base b con:

n cifras enteras

k cifras fraccionarias

puede usarse con carácter general la siguiente fórmula:

$$N = \begin{cases} d_{n-1} \dots d_1 d_0 . d_{-1} \dots d_{-k} = \sum_{i=-k}^{n-1} d_i b^i \\ N = d_{n-1} b^{n-1} + \cdots + d_1 b + d_0 + d_{-1} b^{-1} + \dots + d_{-k} b^{-k} \end{cases}$$

Por ejemplo, el número N=739.3 en base 10 ($739.3|_{10}$) tiene como valor:

$$7x10^2 + 3x10^1 + 9x10^0 + 3x10^{-1}$$

El polinomio que permite calcular el valor del número se denomina *polinomio equivalente.*

Sistemas de numeración

En informática los sistemas de numeración usuales son:

Binario

Base 2 ($2=2^1$).

Símbolos {0,1}

Octal

Base 8 ($8=2^3$).

Símbolos {0,1,2,3,4,5,6,7}

Hexadecimal

Base 16 ($16=2^4$).

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

y por supuesto el **decimal**.

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde un sistema en base *b* a decimal

La conversión desde un sistema posicional de base ***b*** cualquiera al sistema decimal (base 10) se realiza **calculando el valor del polinomio equivalente.**

Código	Número	Polinomio equivalente	Valor en decimal
Binario	110.01	$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$	6.25
Octal	56.2	$5 \cdot 8^1 + 6 \cdot 8^0 + 2 \cdot 8^{-1}$	46.25
Hexadecimal	3F.D	$3 \cdot 16^1 + 15 \cdot 16^0 + 13 \cdot 16^{-1}$	63.8125

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b*

Se transforma por separado la parte entera y la parte fraccionaria del número:

- Parte entera: divisiones sucesivas entre *b.*
- Parte fraccionaria: productos sucesivos por *b.*

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base b: parte entera

Haremos divisiones sucesivas del número N por la base b hasta alcanzar un cociente C_{n-1} que sea menor que la base b.

$$
\begin{array}{l|l}
N & b \\
\hline
r_0 & C_1 \quad \Big| \quad b \\
\hline
 & r_1 \quad C_2 \\
 & \ddots \\
 & \ddots \\
 & C_{n-2} \quad \Big| \quad b \\
\hline
 & r_{n-2} \quad C_{n-1}
\end{array}
$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b* : parte entera

$$\begin{array}{r|l} N & b \\ \hline r_0 & C_1 \end{array} \qquad \begin{array}{r|l} C_1 & b \\ \hline r_1 & C_2 \end{array}$$

$$\ddots$$

$$\ddots$$

$$\begin{array}{r|l} C_{n-2} & b \\ \hline r_{n-2} & C_{n-1} \end{array}$$

$$\begin{array}{r|l} N & b \\ \hline r_0 & C_1 \end{array} \rightarrow N = b \cdot C_1 + r_0 \qquad \begin{array}{r|l} C_1 & b \\ \hline r_1 & C_2 \end{array} \rightarrow C_1 = b \cdot C_2 + r_1$$

Sustituyendo C_1 en la 1ª ecuación:

$$N = b \cdot (b \cdot C_2 + r_1) + r_0 = b^2 \cdot C_2 + b \cdot r_1 + r_0$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base b : parte entera

$$\begin{array}{r|l} N & b \\ \hline r_0 & C_1 \end{array} \qquad \begin{array}{r|l} C_1 & b \\ \hline r_1 & C_2 \end{array}$$

$$\ddots$$

$$\ddots$$

$$\begin{array}{r|l} C_{n-2} & b \\ \hline r_{n-2} & C_{n-1} \end{array}$$

$$\begin{array}{r|l} N & b \\ \hline r_0 & C_1 \end{array} \rightarrow N = b \cdot C_1 + r_0 \qquad \begin{array}{r|l} C_{i-1} & b \\ \hline r_{i-1} & C_i \end{array} \rightarrow C_{i-1} = b \cdot C_i + r_{i-1}$$

Si repetimos el proceso de sustitución de los cocientes C_i obtenemos:

$$N = C_{n-1}b^{n-1} + r_{n-2}b^{n-2} + \cdots + r_1 b + r_0$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b*: parte entera

$$\begin{array}{r|l} N & b \\ \hline r_0 & C_1 \quad \Big| \quad b \\ \hline & r_1 \quad C_2 \\ & \ddots \\ & \ddots \\ & C_{n-2} \quad \Big| \quad b \\ \hline & r_{n-2} \quad C_{n-1} \end{array}$$

Si repetimos el proceso de sustitución de los cocientes C_i obtenemos:

$$N = C_{n-1}b^{n-1} + r_{n-2}b^{n-2} + \cdots + r_1 b + r_0$$

¡Hemos obtenido el **polinomio equivalente**!. El número en base b será:

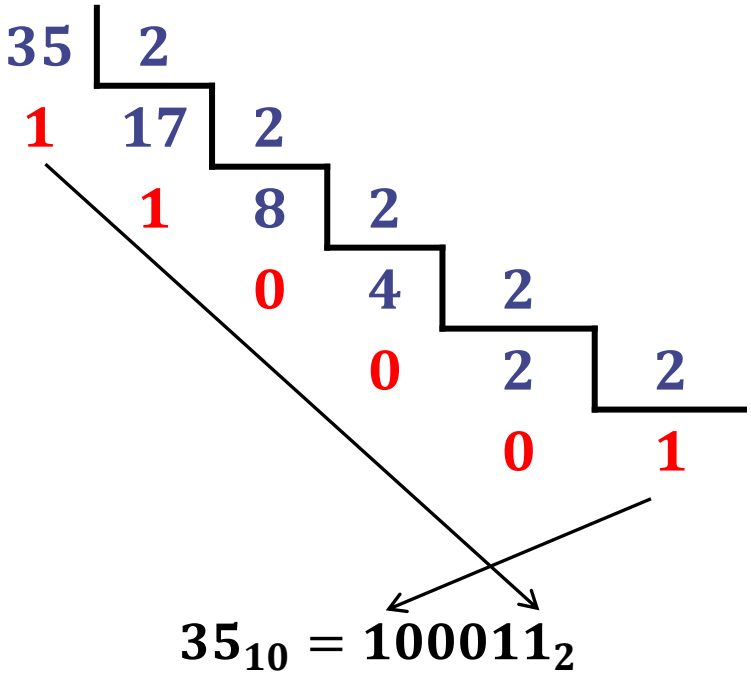
$$N_b = C_{n-1}r_{n-2} \ldots r_1 r_0$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b*: parte entera

Ejemplo: Conversión del número 35_{10} a base 2

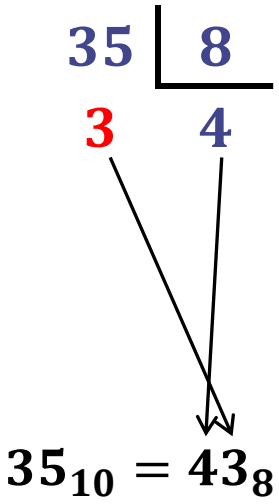


Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b*: parte entera

Ejemplo: Conversión del número 35_{10} a base 8

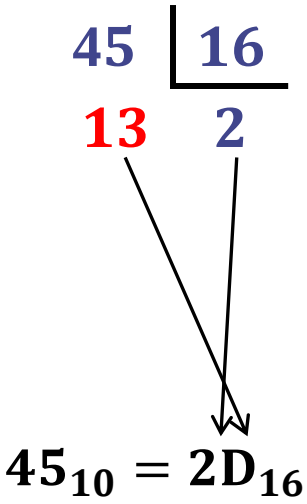


Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a un sistema en base *b*: parte entera

Ejemplo: Conversión del número 45_{10} a base 16



Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a base *b*: parte fraccionaria

La parte fraccionaria de un número es aquella que representa una cantidad inferior a la de la unidad.

Un número fraccionario N_b en base *b* será:

$$N_b =. a_{-1}\, a_{-2} \ldots a_{-j+1} a_{-j}$$

Por tanto, la parte fraccionaria N_{10} en base 10 será:

$$N_{10} = a_{-1}b^{-1} + a_{-2}b^{-2} + \cdots + a_{-j+1}b^{-j+1}a_{-j}b^{-j}$$

Obviamente conocemos la parte fraccionaria N_{10} y queremos obtener los coeficientes fraccionarios a_{-i}.

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a base *b*: parte fraccionaria

Veamos que mediante **multiplicaciones sucesivas** por el valor de la base b, podemos obtener esos coeficientes:

$$b \cdot N_{10} = b \cdot \left(a_{-1}b^{-1} + a_{-2}b^{-2} + \cdots + a_{-j+1}b^{-j+1}a_{-j}b^{-j}\right)$$
$$= \boldsymbol{a_{-1}} + \left(a_{-2}b^{-1} + \cdots + a_{-j+1}b^{-j+2}a_{-j}b^{-j+1}\right) = \boldsymbol{a_{-1}}.x_{-1}x_{-2}\ldots$$

Por tanto, en la primera multiplicación obtenemos un número cuya parte entera es precisamente el primer coeficiente buscado.

¿Cómo obtener el 2º coeficiente? Quedándonos con la parte fraccionaria del valor obtenido en la etapa anterior y volviendo a multiplicar por b.

$$\boldsymbol{a_{-1}}.x_{-1}x_{-2}\ldots - \boldsymbol{a_{-1}} = b \cdot N_{10} - \boldsymbol{a_{-1}} = \left(a_{-2}b^{-1} + \cdots + a_{-j+1}b^{-j+2}a_{-j}b^{-j+1}\right)$$

$$b \cdot \left(a_{-2}b^{-1} + a_{-3}b^{-2} + \cdots + a_{-j+1}b^{-j+2}a_{-j}b^{-j+1}\right)$$
$$= \boldsymbol{a_{-2}} + \left(a_{-3}b^{-1} + \cdots + a_{-j+1}b^{-j+3}a_{-j}b^{-j+2}\right) = \boldsymbol{a_{-2}}.y_{-1}y_{-2}\ldots$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión desde decimal a base *b*: parte fraccionaria

En este caso se tiene que definir un **criterio de parada** dado que la conversión puede no tener una solución finita. Opciones:

- número máximo de cifras fraccionarias
- error de truncamiento por debajo de una tolerancia dada

El **error de truncamiento** es la diferencia en valor absoluto entre el valor real y el obtenido.

Ejemplo: Convertir a binario 0.79_{10}

0.79	x	2	**1**.58	→	$a_{-1} = \mathbf{1}$	0.58
0.58	x	2	**1**.16	→	$a_{-2} = \mathbf{1}$	0.16
0.16	x	2	**0**.32	→	$a_{-3} = \mathbf{0}$	0.32
0.32	x	2	**0**.64	→	$a_{-4} = \mathbf{0}$	0.64
0.64	x	2	**1**.28	→	$a_{-5} = \mathbf{1}$	0.28
0.28	x	2	**…**	→	…	…

$$\mathbf{0.79_{10} = 0.11001\ldots_2}$$

El error de truncamiento ε_t es:

$$\varepsilon_t = \left|0.79 - (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-5})\right| = |0.79 - 0.78125| = 0.00875$$

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión entre binario, octal y hexadecimal

La conversión es muy simple dado que $8 = 2^3$ y $16 = 2^4$.

Desde base 2:

- Se forman grupos desde el punto fraccionario de 3 (4) cifras consecutivas. Cada grupo corresponde a un símbolo en base octal (hexadecimal). Si quedan grupos de 3 (4) sin completar, se rellenan con 0's: por la izquierda si es la parte entera y por la derecha si es la parte fraccionaria.

Desde base octal (hexadecimal):

- Se transforma cada cifra a su correspondiente valor de 3 (4) cifras en binario, manteniendo el orden a izquierda y derecha del punto fraccionario.

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

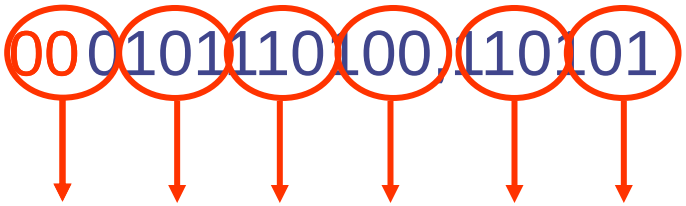
HEXADECIMAL	BINARIO
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Sistemas de numeración

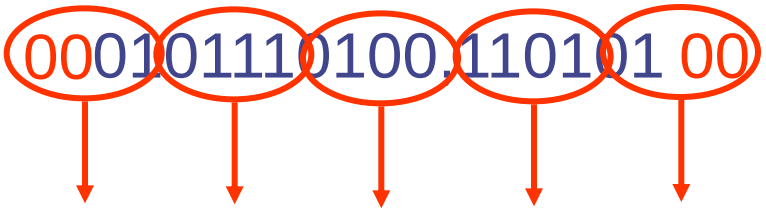
Conversión entre sistemas de numeración

Conversión entre binario, octal y hexadecimal

Ejemplo: Convertir el número 0101110100.110101_2



En base 8 (octal)



En base 16 (hexadecimal)

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

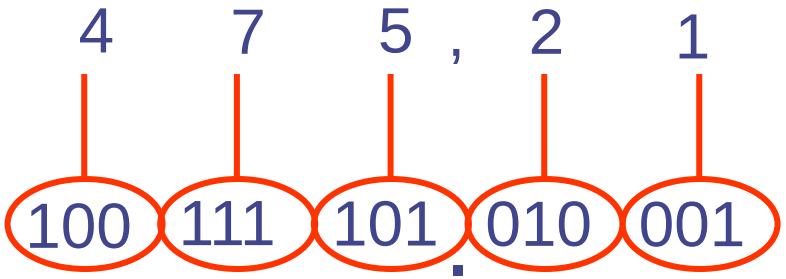
HEXADECIMAL	BINARIO
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Sistemas de numeración

Conversión entre sistemas de numeración

Conversión entre binario, octal y hexadecimal

Ejemplo: Convertir el número 475.21_8 a binario



En base 2 (binario)

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

La conversión de octal a hexadecimal o viceversa se hace de forma rápida convirtiendo previamente a binario.

Octal → Binario → Hexadecimal

Hexadecimal → Binario → Octal

Representación de números enteros

La representación interna, aunque similar, va a depender de cada lenguaje de programación, compilador y arquitectura del ordenador. Nos centraremos en C++.

El lenguaje C++ permite trabajar con enteros almacenados en memoria ocupando 1(8), 2(16), 4(32) y 8(64) bytes(bits).

Como hemos visto, el número n de bits determina el número de posibles valores representables.

Además, C++ permite trabajar utilizando valores con signo y sin signo.

Para la representación de enteros con signo tenemos que dedicar un bit de la codificación para determinar si el valor es positivo o negativo.

Así, el **Java** no contempla el uso de enteros sin signo. **Python** o **Lisp** permiten trabajar con enteros con un número *ilimitado* de bits.

Representación de números enteros

Enteros sin signo

Muchos problemas reales involucran el uso de valores positivos (números naturales) más el 0 (número de alumnos en clase, posición de un ítem en una secuencia ordenada, etc.)

La representación interna coincide con el sistema numérico posicional descrito anteriormente.

El rango de representación es $[0, 2^n - 1]$

Decimal sin signo	Representación binario sin signo para n=8 bits							
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
…	…	…	…	…	…	…	…	…
254	1	1	1	1	1	1	1	0
255	1	1	1	1	1	1	1	1

Representación de números enteros

Enteros sin signo

En C++ una declaración como

`unsigned x;` (o la equivalente **`unsigned int x;`**)

determina que la variable **`x`** se almacenará como un entero sin signo y ocupará en memoria 4(32) bytes(bits).

Por tanto, el rango de la representación es:

$$[0, 2^{32} - 1] = [0,4294967295]$$

A los efectos de programar un algoritmo, los enteros con signo pueden utilizarse sin necesitar usar los enteros sin signo. La única diferencia es que el rango de valores es la mitad al necesitar un bit para el signo.

Realizar un programa que mezcle representaciones con y sin signo, o que realice operaciones tales como restas entre valores sin signo, requiere ser muy cuidadoso para no cometer graves errores.
Por ello, en general, durante el curso **no utilizaremos variables sin signo en nuestros programas**.

Representación de números enteros

Enteros: representación signo magnitud

La representación signo magnitud de un entero consiste en utilizar el **bit más significativo** (el situado más a la izquierda) para codificar el signo:

- 0 para los positivos
- 1 para los negativos

Los $\boldsymbol{n-1}$ bits restantes, el **significando**, se usan para codificar la magnitud (valor absoluto) del entero siguiendo el sistema posicional.

El rango de representación es $[-(2^{n-1}-1), 2^{n-1}-1]$

Representación de números enteros

Enteros: representación signo magnitud

Esta representación, escasamente utilizada en computación para enteros y expuesta aquí por motivos pedagógicos, tiene 2 problemas:

- el 0 tiene 2 codificaciones
- las operaciones aritméticas son complejas de realizar: requieren circuitería electrónica diferente para sumas y restas.

Decimal con signo	Representación binario con signo para n=8 bits							
	Signo	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-127	1	1	1	1	1	1	1	1
-126	1	1	1	1	1	1	1	0
...	...	...	...	...	...	...	...	...
-1	1	0	0	0	0	0	0	1
-0	1	0	0	0	0	0	0	0
+0	0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...
+126	0	1	1	1	1	1	1	0
+127	0	1	1	1	1	1	1	1

Representación de números enteros

Enteros: representación signo magnitud

Ejemplo:

Representación del número -47_{10} (decimal) con 8 bits

El signo del número -47_{10} es negativo, luego el bit de signo es **1**.

$|-47_{10}|=47_{10}=32_{10}+8_{10}+4_{10}+2_{10}+1_{10}=\mathbf{1}x2^5+\mathbf{0}x2^4+\mathbf{1}x2^3+\mathbf{1}x2^2+\mathbf{1}x2^1+\mathbf{1}x2^0$

En binario con 7 bits es: $\mathbf{0101111}_2$ → rellenamos de **0**'s a la izquierda hasta tener los 7 bits.

El número -47_{10} en binario con formato de signo y magnitud es: $\mathbf{10101111}_2$.

Representación de números enteros

Enteros: representación en complemento a 2

El **complemento a la base** $\boldsymbol{b}$ de un número positivo $\boldsymbol{X}$ codificado con $\boldsymbol{n}$ posiciones es:

$$\text{ComplementoBase}(\boldsymbol{X}) = \boldsymbol{b^n - X}$$

En binario, $\boldsymbol{b = 2}$, de ahí **complemento a 2**. Es una representación muy extendida para trabajar con enteros negativos y es la que usa C++:

- El valor 0 y los **positivos** se representan como en **signo magnitud**.
- Los **negativos** se representan con el **complemento a 2** del valor absoluto

El **bit más significativo** nos indica el signo:

- 0 para los positivos
- 1 para los negativos

pero ahora **este bit de signo tiene valor**: en la representación en complemento a la base, el **polinomio equivalente** de un número con $\boldsymbol{n}$ posiciones pasa a ser:

$$N = \boldsymbol{-d_{n-1} b^{n-1}} + d_{n-2} b^{n-2} + \cdots + d_1 b + d_0$$

Representación de números enteros

Enteros: representación en complemento a 2

El rango de representación es $[-2^{n-1}, 2^{n-1} - 1]$.

El rango es asimétrico: podemos representar un valor negativo más respecto a los positivos.

La representación en complemento a 2 ya fue sugerida por J. V. Neumann en su célebre borrador de 1945 *First Draft of a Report on the EDVAC*.

Decimal con signo	Representación binario complemento a 2 para n=4 bits					
	Signo	Bit 2	Bit 1	Bit 0	Cálculo polinomio equivalente	$b^n - X$
-8	1	0	0	0	$-\mathbf{1} \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$	$2^4 - 8 = 8$
-7	1	0	0	1	$-\mathbf{1} \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	$2^4 - 7 = 9$
...	...	...	...	...	... ...	
-1	1	1	1	1	$-\mathbf{1} \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$	$2^4 - 1 = 15$
0	0	0	0	0	$+\mathbf{0} \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$	
+1	0	0	0	1	$+\mathbf{0} \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	
...	...	...	...	...	... ...	
+6	0	1	1	0	$+\mathbf{0} \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$	
+7	0	1	1	1	$+\mathbf{0} \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$	

Representación de números enteros

Enteros: representación en complemento a 2

Existe una regla muy fácil (que los circuitos electrónicos aprovechan) para obtener el complemento a 2 de un número binario positivo:

1. Nos desplazamos de derecha a izquierda bit a bit hasta encontrar el primer 1. Lo dejamos inalterado.
2. A partir de este 1, cambiamos 0's por 1's y 1's por 0's.

Otra regla simple consiste en:

1. Cambiar 0's por 1's y 1's por 0`s y
2. Al resultado obtenido sumarle 1.

Existen otra representación similar, el **complemento a la base menos 1** (**complemento a 1** en binario).

Fue utilizada en algunas series de computadoras descendientes de la ENIAC, como la serie UNIVAC.

En este caso, el complemento a 1 se obtiene simplemente intercambiando 0's y 1's. No la estudiaremos en el curso por su escasa utilización actual.

Representación de números enteros

Enteros: representación en complemento a 2

Ejemplos

Representar -13_{10} en binario complemento a 2 con 6 bits

Método 1

Obtenemos 13_{10} con 6 bits: $13_{10} = 8_{10}+4_{10}+1_{10}=001101_2$

De derecha a izquierda buscamos el primer 1: $00110\mathbf{1}_2$

A partir del primer 1, cambiamos 0's por 1's y 1's por 0's: $\mathbf{110011}_2$

Método 2

Por definición, $\text{ComplementoBase}(X) = b^n - X = 2^6\text{-}13=64\text{-}13=51$

Obtenemos 51_{10} con 6 bits: $51_{10}=32_{10}+16_{10}+2_{10}+1_{10}=\mathbf{110011}_2$

Representación de números enteros

Enteros: representación en complemento a 2

Ejemplos

Representar -13_{10} en binario complemento a 2 con 10 bits

Método 1

Obtenemos 13_{10} con 10 bits: $13_{10} = 8_{10}+4_{10}+1_{10}=0000001101_2$

De derecha a izquierda buscamos el primer 1: $000000110\mathbf{1}_2$

A partir del primer 1, cambiamos 0's por 1's y 1's por 0's: $\mathbf{1111100111}_2$

Método 2

Por definición, $\text{ComplementoBase}(X) = b^n - X$ = 2^{10}-13=1024-13=1011

Obtenemos 1011_{10} con 10 bits:

$$1011_{10}=512_{10}+256_{10}+128_{10}+64_{10}+32_{10}+16_{10}+2_{10}+1_{10}=\mathbf{1111110011}_2$$

Representación de números enteros

Enteros: representación en C++

En C++, una declaración como

```
int x;
```

reservará espacio en memoria para la variable entera **`x`** usando complemento a 2 y ocupando 4(32) bytes(bits).

Así, la línea de código

```
x=23;
```

supondrá que en memoria se almacenará la secuencia

```
00000000 00000000 00000000 00010111
```

mientras que la línea de código

```
x=-23;
```

supondrá que en memoria se almacenará la secuencia

```
11111111 11111111 11111111 11101001
```

Representación de números enteros

Enteros: representación en C++

El lenguaje C++ (y C) **no especifica el tamaño en bytes** de los tipos de datos. Sólo estandariza un número mínimo de bytes.

Es sin duda un aspecto *desconcertante* del lenguaje, pues compromete la portabilidad de los programas entre plataformas.

Así, una variable **`int`** puede ocupar 2 bytes en una arquitectura (en 1980 no existían ordenadores de 32 y/o 64 bits) y 4 en otra (lo habitual hoy en día, aunque no por mucho tiempo).

¿Podemos trabajar en C++ con enteros de tal forma que ocupen en memoria menos o más bytes?

Sí. Para ello existen otros tipos de datos para enteros.

Por ejemplo:

- **`short x;`**
  hará que la variable entera **`x`** se almacene en memoria con 2 bytes si el tamaño en bytes de los **`int`** es de 4 bytes.
- **`long long x;`**
  hará que la variable entera **`x`** se almacene en memoria con 8 bytes si el tamaño en bytes de los **`int`** es de 4 bytes.

Representación de números enteros

Enteros: representación en C++

Desde el estándar del lenguaje C, C99, y ahora, desde el estándar C++ 11 podemos usar tipos de datos enteros de tamaño predefinido, lo que garantiza la portabilidad. Así, en **`int64_t x;`** la variable entera **`x`** se almacenará con representación complemento a 2 con 8 bytes con independencia de la plataforma.

Nombre	Tamaño	Rango
`int8_t`	1 byte con signo	-128 to 127
`uint8_t`	1 byte sin signo	0 to 255
`int16_t`	2 byte con signo	-32,768 to 32,767
`uint16_t`	2 byte sin signo	0 to 65,535
`int32_t`	4 byte con signo	-2,147,483,648 to 2,147,483,647
`uint32_t`	4 byte sin signo	0 to 4,294,967,295
`int64_t`	8 byte con signo	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`uint64_t`	8 byte sin signo	0 to 18,446,744,073,709,551,615

En este curso nos limitaremos a usar siempre **`int`** para referirnos a valores enteros. Un tipo de dato que nos será útil más adelante es **`size_t`**. Este tipo de dato es un entero sin signo que nos garantiza almacenar el máximo valor posible en ese dispositivo.

Aritmética de enteros

El álgebra de Boole

George Boole (1815 – 1864) observó que, codificando los valores lógicos VERDAD y FALSO con los valores binarios 1 y 0, podía formularse un álgebra que capturaba los principios básicos del razonamiento lógico.

Claude Shannon (1916 – 2001), padre de la Teoría de la Información, mostró en su tesis doctoral en 1937 que el álgebra de Boole podía aplicarse al análisis y diseño de relés electromecánicos, es decir, sistemas digitales.

Aritmética de enteros

El álgebra de Boole

Las operaciones básicas del álgebra de Boole pueden representarse con **tablas de verdad**.

~ *NOT*	
0	1
1	0

& AND		
0	**0**	0
0	**1**	0
1	**0**	0
1	**1**	1

| OR		
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	1

^ XOR		
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	0

Una **tabla de verdad** despliega el valor de verdad de una proposición para cada combinación de valores de verdad que se pueda asignar a sus componentes.

Por ejemplo:

Dadas dos proposiciones P y Q,

la proposición compuesta P AND Q es cierta si P es cierta y Q es cierta.

la proposición compuesta P OR Q es cierta si P es cierta o Q es cierta.

A nivel de bits, dados dos bits p y q,

$p\&q = 1$ si $p = 1$ y $q = 1$

$p|q = 1$ si $p = 1$ o $q = 1$

Las 4 operaciones básicas del álgebra de Boole pueden aplicarse también a secuencias de bits

```
  1 0 1 0        1 0 1 0        1 0 1 0          1 1 0 0
& 1 1 0 0      | 1 1 0 0      ^ 1 1 0 0      ~
-------        -------        -------          -------
  1 0 0 0        1 1 1 0        0 1 1 0          0 0 1 1
```

Aritmética de enteros

El álgebra de Boole

C++ posee los operadores a nivel de bits **&**, **|**, **^** y **~**.
Ejemplo: si ejecutamos el siguiente programa

```
#include <iostream>
using namespace std;
int main()
{
    int x=10;
    cout <<"x="<<x<<endl;
    x=~x; // Aplicamos el operador negación NOT
    cout <<"~x="<<x<<endl;
}
```

vemos que la salida por consola es:

```
x=10
~x=-11
```

La representación de 10_{10} es 00000000 00000000 00000000 00001010

Por tanto, $\sim 10_{10}$ es 11111111 11111111 11111111 11110101

Al tener un 1 como bit más significativo, **~x** es un número negativo y, por tanto, representado en complemento a 2.
Descomplementando, tenemos 00000000 00000000 00000000 00001011=11_{10}

Aritmética de enteros

El álgebra de Boole

Ejemplo: si ejecutamos el siguiente programa

```
#include <iostream>
using namespace std;
int main()
{
    int x=10;
    cout <<"x="<<x<<endl;
    x=~x+1; //Complemento a 2==operador NOT +1
    cout <<"~x+1="<<x<<endl;
}
```

vemos que la salida por consola es:

```
x=10
~x+1=-10
```

Es decir, el complemento a 2 de un número podemos obtenerle negando bit a bit un número binario y sumando 1 al resultado.

No trabajaremos a nivel de bit en este curso, pero estos ejemplos nos permiten observar que C++ permite programar a un nivel cercano al hardware.

Aritmética de enteros

La suma binaria

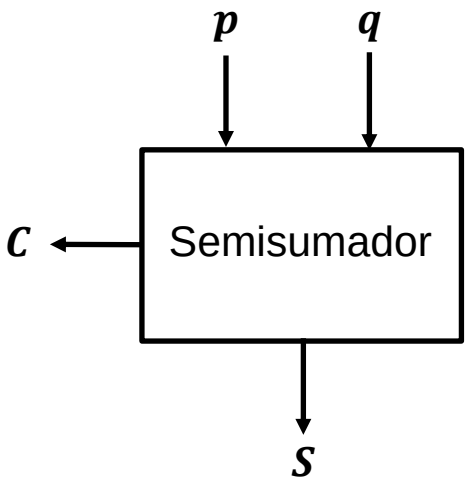
Las reglas básicas para sumar 2 bits son:

0+0=0 0+1=1 1+0=1 1+1=10

Al igual que ocurre en la aritmética decimal (ej. 7+8=**1**5), se produce **acarreo** en el caso 1+1=**1**0.

La suma aislada de dos bits, conocida como **semisumador binario**, podemos describirla con una tabla de verdad con dos bits de entrada, p y q, y dos bits de salida, suma S y acarreo C.

SEMISUMADOR BINARIO			
p	q	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Nótese que las operaciones del semisumador son fácilmente reconocibles:

$S = p \text{ ^ } q$ XOR

$C = p \,\&\, q$ AND

Aritmética de enteros

La suma binaria

Supongamos que queremos sumar 2 números binarios de 5 bits cada uno, por ejemplo 00110+00111, o lo que es lo mismo $6_{10}+7_{10}$

1 1	Acarreos
0 0 1 1 0	6_{10}
0 0 1 1 1	7_{10}
0 1 1 0 1	13_{10}

Al igual que ocurre en la aritmética decimal (ej. 87+78=**16**5), al sumar dos dígitos también hay que tener en cuenta el acarreo procedente de la suma de los dígitos anteriores.

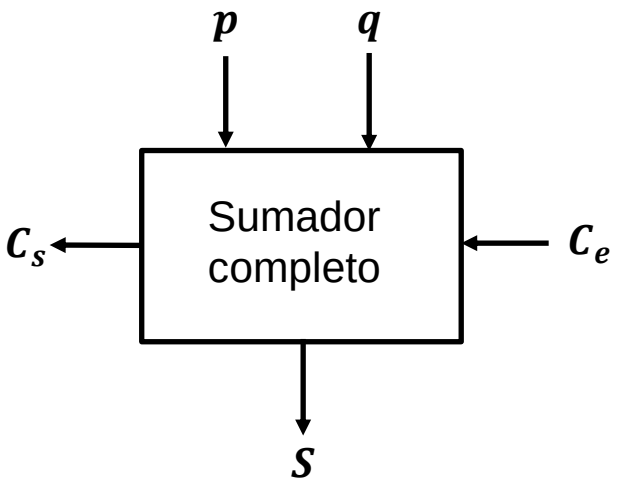
Por ello, para describir la suma de dos bits en un número binario se utiliza el **sumador completo**.

Aritmética de enteros

La suma binaria

El **sumador completo binario** podemos describirlo con una tabla de verdad con tres bits de entrada, p, q y C_e y dos bits de salida S y C_s.

C_e es el valor de acarreo procedente de la suma de los bits anteriores y C_s es el acarreo que se inyecta a la siguiente suma de bits.

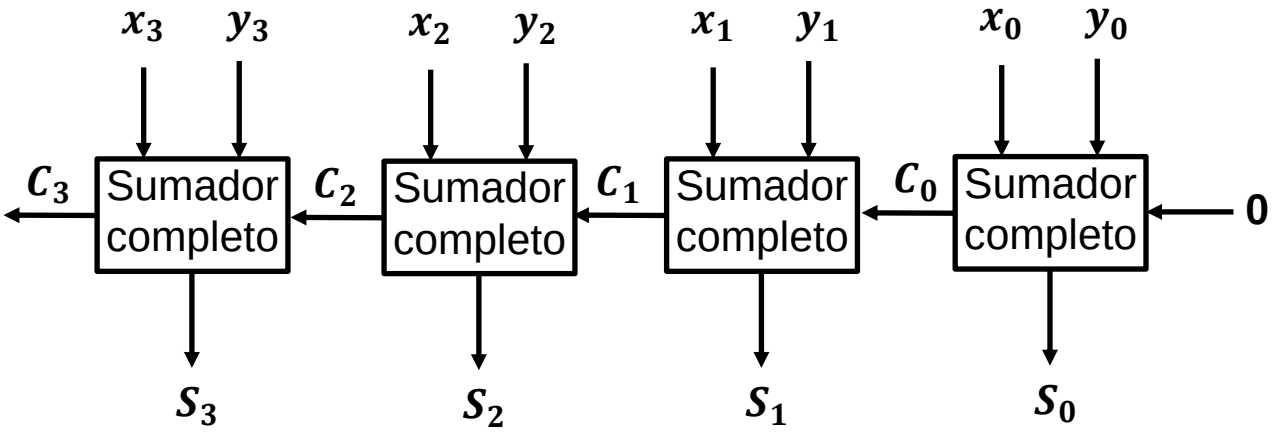


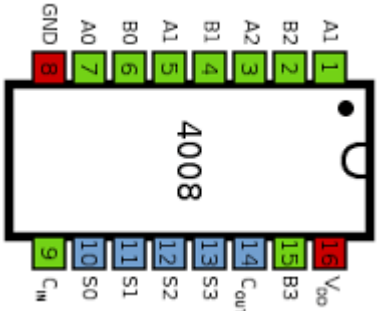
SUMADOR COMPLETO BINARIO				
p	q	C_e	S	C_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Aritmética de enteros

La suma binaria

Supongamos que necesitamos sumar dos números binarios $\boldsymbol{x}$ e $\boldsymbol{y}$ de 4 bits. Bastaría disponer de 4 sumadores completos colocados en serie para realizar la operación en paralelo.





Un ejemplo comercial es el chip 4008

Combinando sumadores completos podemos sumar números con el número de bits deseado.

Este tipo de circuitos los estudiaréis más a fondo en la asignatura Fundamentos de Electrónica.

Aritmética de enteros

La suma binaria

¿qué pasa si sumamos un número positivo con uno negativo?
¿y si los dos operandos son negativos?

La representación en complemento a 2 realiza las operaciones correctamente.

Veámoslo con 2 ejemplos:
Queremos sumar 6_{10} y -7_{10}. El resultado debe ser obviamente -1_{10}

	Acarreos
0 0 1 1 0	6_{10}
1 1 0 0 1	-7_{10}
1 1 1 1 1	-1_{10}

Queremos sumar -6_{10} y -7_{10}. El resultado debe ser obviamente -13_{10}

	Acarreos
1 1	
1 1 0 1 0	-6_{10}
1 1 0 0 1	-7_{10}
1 0 0 1 1	-13_{10}

El último acarreo no se tiene en cuenta pues no hay un sumador a continuación al cual inyectarlo.

Aritmética de enteros

La resta binaria

Hemos hablado de sumas, ¿qué pasa con restas como en **`x=y-z;`**?

Cuando delante de una variable aparece el operador **–**, internamente:

- el valor de la variable afectada (**`z`** en el ejemplo) se modifica calculando su complemento a 2
- el operador **–** pasa a ser **+**

Como hemos visto en un ejemplo anterior: **`-z` ←→ `~z+1`**

```
#include <iostream>
using namespace std;
int main()
{
    int x,y=10;
    int z=7;
    x=y-z;
    cout <<" y-z="<<x<<endl;
    x=y+~z+1; // Resta == suma del complemento a 2
    cout <<" y+~z+1="<<x<<endl;
}
```

```
y-z=3
y+~z+1=3
```

Aritmética de enteros

Multiplicaciones y divisiones con enteros

Estas operaciones involucran procedimientos más complejos y no las estudiaremos.

En esencia, estas operaciones se construyen mediante algoritmos que realizan sumas parciales y desplazamientos de bits.

0 1 1 0	6_{10}
x 0 0 1 1	3_{10}
0 0 1 1 0	
0 0 1 1 0	→ Desplazamiento de 1 bit a la izquierda
0 1 0 0 1 0	18_{10}

C++ dispone de operadores de desplazamiento de bits: **<<** y **>>**.

Por ejemplo:

`x<<2` desplaza a la izquierda el valor de **`x`**, rellenando con dos 0's por la derecha. Equivale a multiplicar por 4 el valor de **`x`**.

`x>>3` desplaza a la derecha el valor de **`x`**, rellenando con tres 0's por izquierda. Equivale a dividir por 8 el valor de **`x`** siempre que **`x`** sea un valor positivo. ¿por qué no equivale a una división cuando **`x`** es negativo?

No estudiaremos los operadores de desplazamiento en este curso.

Representación de números reales

Un número real consta de una parte entera y otra fraccionaria, más el signo.

Existen dos opciones de representación: **coma fija** o **coma flotante**.

Coma fija

Se asigna una cantidad fija de dígitos para la parte entera y una cantidad fija para la parte fraccionaria.

Ejemplo: Obviando por el momento el signo, si disponemos de 8 bits y reservamos 5 bits para la parte fija y 3 para la fraccionaria, el número 21.75_{10} sería 10101.110

Al usar la notación en coma fija, queda muy limitado el número de cantidades a representar y todas ellas tienen la misma resolución.

Ejemplo: 8 bits (5 parte entera y 3 parte fraccionaria) sin tener en cuenta el signo

No podremos representar números enteros mayores o iguales que 32 (2^5)

Ni números más pequeños que 0.125 (2^{-3}).

La resolución fija entre dos valores consecutivos es 0.125 (2^{-3}).

Representación de números reales

Coma flotante

Un número real r en coma flotante se representa según la **notación científica**:

$$r = m \cdot b^e$$

donde:

- m es la **mantisa**, un coeficiente formado por un número real con una sola cifra entera seguido por una coma (o punto) y de varias cifras fraccionarias.
- b es la **base** del sistema de numeración (10 en el caso decimal y 2 en el binario).
- e es el **exponente** al que se eleva la base.

Se consigue, manteniendo el número de cifras respecto a la coma fija:

- representar un rango mayor de números (dependiendo del exponente)
- con una mayor o menor precisión (según el número de cifras en la mantisa).

Representación de números reales

Coma flotante: norma IEEE 754

Hasta la década de los 80, cada fabricante usaba su propia representación, lo que dificultaba el desarrollo de algoritmos eficientes de cálculo, procesadores estándar, etc.

Por ello, la asociación IEEE desarrolló el estándar **IEEE 754**, que contempla, entre otros, dos posibles formatos:

Simple precisión: 32 bits

- **1** bit de signo
- **8** bits para el exponente (representado en exceso **127**)
- **23** bits para la mantisa

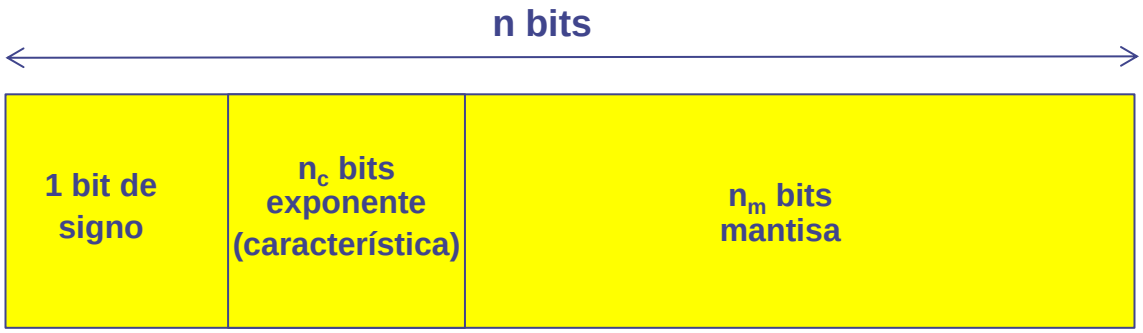
Doble precisión: 64 bits.

- **1** bit de signo
- **11** bits para el exponente (representado en exceso **1023**)
- **52** bits para la mantisa

Representación de números reales

Coma flotante: norma IEEE 754

El orden de almacenamiento es: signo, seguido de exponente, seguido de mantisa:



El orden no es baladí: permite que los algoritmos de comparación entre números enteros sean también válidos para esta representación de números reales.

Se emplea en la bibliografía la palabra ***característica*** para distinguir el *valor* real del **exponente** de su *representación*, la cual veremos veremos enseguida.

Representación de números reales

Coma flotante: norma IEEE 754

Números negativos

Como en la representación de enteros se usa un **bit de signo**

- Positivos : Bit **0**
- Negativos: Bit **1**

Representación de números reales

Coma flotante: norma IEEE 754

Representación de los exponentes: la característica

En el caso de la representación del **exponente** nos encontramos con el hecho de que éste puede ser positivo ($3.45 \cdot 10^{3}$) o negativo ($2.7 \cdot 10^{-6}$) y, por tanto, además de un bit de signo para determinar si el número es positivo o negativo, necesitamos un segundo bit de signo para el exponente.

Este segundo bit de signo es un grave inconveniente pues su uso invalida el poder comparar de forma sencilla dos valores reales entre sí. Afortunadamente, podemos ahorrarnos ese segundo bit de signo para el exponente usando una **representación en exceso.**

Representar los exponentes en complemento a 2 no aporta ninguna ventaja:

- los exponentes no se suman al sumar dos números
- se pierde la propiedad de orden comentada arriba

Representación de números reales

Coma flotante: norma IEEE 754

Representación de los exponentes: la característica

La **representación en exceso k** de un número x es tan simple como sumar al número un número positivo k, $x_{en\_exceso} = x + k$.

La norma IEEE 754 define el rango de valores para los exponentes de la siguiente forma:

- Simple precisión (8 bits): rango=[-127,128]

  →$2^{8} = 256$ posibles valores

- Doble precisión (11 bits): rango=[-1023,1024]

  →$2^{11} = 2048$ posibles valores

¿Qué pasa si representamos en exceso los exponentes usando $k = 127$ para simple precisión y $k = 1023$ para doble precisión?

En la representación en exceso los nuevos rangos son [0,255] y [0,2047]:

Representación de números reales

Coma flotante: norma IEEE 754

Representación de los exponentes: la característica

La **representación en exceso** tiene la importante propiedad de que dos números representados en exceso conservan sus propiedades de orden (las tareas de ordenación son muy habituales en los algoritmos).

$$x_{en\_exceso} \leq y_{en\_exceso} \Leftrightarrow x \leq y$$

No hay beneficio sin servidumbre: si sumamos dos números representados en exceso, debemos corregir el resultado *restando un exceso*.

$$x_{en\_exceso} + y_{en\_exceso} = x + k + y + k = (x + y)_{en\_exceso} + \boldsymbol{k}$$

Representación de números reales

Coma flotante: norma IEEE 754

Representación de la mantisa

Se utiliza la notación científica, normalizando del modo $1.m$: el primer uno (bit más significativo) del número se sitúa a la izquierda de la coma.

Uno de los retos que debe lograrse para representar los números reales es conseguir poder representar el mayor rango de ellos y con la mejor precisión posible.

Una solución obvia es dedicar más bits (32 frente a 64 en el caso de simple o doble precisión) y también **ahorrarnos bits superfluos**.

- Puesto que siempre tendremos un único 1 a la izquierda del punto fraccionario, no hace falta representarlo (*uno implícito*).

La mantisa m representa en binario la magnitud del número.

- Aquí tampoc tiene sentido usar la representación complemento a 2 dado que, en general, las mantisas de dos números en notación científica no pueden sumarse directamente.

Representación de números reales

Coma flotante: norma IEEE 754

Ejemplo: representar $13.125|_{10}$ en simple precisión

1) Transformamos a binario la parte entera y la parte fraccionaria

   $13.125|_{10} = 1101.001|_2$

2) Determinamos el signo

   El número es positivo, luego el bit de signo es **0**.

3) Representamos el número según la notación científica

   $1101.001 = 1.101001 \cdot 2^3$

4) Identificamos la mantisa (desechamos el 1 implícito) y el exponente

   $1.\mathbf{101001} \cdot 2^{\mathbf{3}}$

5) Rellenamos la mantisa de 0's por la derecha hasta completar los 23 bits que marca la norma

   10100100000000000000000

6) Añadimos el exceso **127** al exponente y lo representamos en binario con 8 bits

   $(\mathbf{3} + \mathbf{127})_{10} = 130_{10} = (128 + 2)_{10} = (2^7 + 2^1)_{10} = \mathbf{10000010_2}$

7) Formamos el número

   010000010**10100100000000000000000**

Representación de números reales

Coma flotante: norma IEEE 754

Ejemplo: A partir del valor 01000010111011010000000000000000 obtener su valor decimal.

1) Agrupamos los 1+8+23 bits para identificar signo, exponente y mantisa

   0 10000101 11011010000000000000000

2) Determinamos el signo

   El número es positivo, pues el bit de signo es **0**.

3) Calculamos el exponente

   $\mathbf{10000101} = 2^7 + 2^2 + 2^0 = 133$

   Eliminando el exceso, $exponente = 133 - 127 = \mathbf{6}$

4) Calculamos la mantisa añadiendo el bit implícito y desechando 0's superfluos

   11011010000000000000000 → **1.1101101**

5) Formamos el número en notación científica $m \cdot 2^e$

   $\mathbf{1.1101101 \cdot 2^6}$

6) Por comodidad de cálculo, eliminamos el término $\mathbf{2^e}$ desplazando la coma $\boldsymbol{e}$ posiciones (a la derecha si el signo es positivo y a la izquierda si es negativo)

   $\mathbf{1.1101101 \cdot 2^6 = 1110110.1}$

7) Transformamos a base 10

   $\mathbf{1110110.1_2} = (\mathbf{2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^{-1}})_{\mathbf{10}} = \mathbf{118.5_{10}}$

Representación de números reales

Coma flotante: norma IEEE 754

Procedimiento de cálculo general

Un número positivo x podemos expresarle como:

$$x = m \cdot 2^e$$

Tomando logaritmos en base 2:

$$\log_2 x = \log_2(m \cdot 2^e) = \log_2 m + \mathrm{e} \cdot \log_2 2 = \log_2 m + e$$

Puesto que la mantisa normalizada es un valor positivo de la forma $1, \ldots$, es decir, un número mayor o igual que 1 y menor que 2, $\log_2 m$ es un valor mayor o igual que 0 y menor que 1, es decir, $0, \ldots$

Por tanto, si *despreciamos* $\log_2 m$, el exponente e será siempre el entero más próximo **por defecto** al valor obtenido en el cálculo de $\log_2 x$.

Ejemplo: Si $\log_2 x$=23.465, entonces $e = 23$

Si $\log_2 x$=-23.465, entonces $e = -24$

Una vez calculado el exponente:

$$m = \frac{x}{2^e}$$

Representación de números reales

Coma flotante: norma IEEE 754

Procedimiento de cálculo general

Ejemplo: Representar $x = 1359 \cdot 10^{26}$ en simple precisión IEEE 754

1) El número es positivo, luego el bit de signo es **0**.
2) Calculamos $\log_2 x$

   $\log_2(1359 \cdot 10^{26}) =$ **96.77846020783882**
3) Obtenemos el exponente tomando el entero por defecto más próximo

   $e = entero\_por\_defecto(96.77846020783882) =$ **96**

   El exponente en exceso es 96+127=**223** cuya representación en binario con 8 bits es **11011111**
4) Calculamos la mantisa

   $m = \frac{x}{2^e} = \frac{1359 \cdot 10^{26}}{2^{96}} =$ **1.715299129486084** ...

   Obviando el bit implícito y aplicando el método de multiplicaciones sucesivas se obtiene la mantisa: **10110111000111011101100**
5) Formamos el número

   01101111101101110001110111011100

Representación de números reales

Coma flotante: norma IEEE 754

Procedimiento de cálculo general

Ejemplo: Representar $x = -0.027 \cdot 10^{-27}$ en simple precisión IEEE 754

1) El número es negativo, luego el bit de signo es **1**.
2) Calculamos $\log_2 x$

   $\log_2(0.027 \cdot 10^{-27}) = \mathbf{-94.90295534445741}$
3) Obtenemos el exponente tomando el entero por defecto más próximo

   $e = entero\_por\_defecto(-94.90295534445741) = \mathbf{-95}$

   El exponente en exceso es -95+127=**32** cuya representación en binario con 8 bits es **00100000**
4) Calculamos la mantisa

   $m = \frac{x}{2^e} = \frac{-0.027 \cdot 10^{-27}}{2^{-95}} = \mathbf{1.}\mathbf{069801973342896}$ ...

   Obviando el bit implícito y aplicando el método de multiplicaciones sucesivas se obtiene la mantisa: **00010001110100000000001**
5) Formamos el número

   100100000**00010001110100000000001**

Representación de números reales

Coma flotante: norma IEEE 754

Casos especiales

La norma contempla situaciones especiales cuando el exponente está formado completamente por 0's (números muy pequeños en valor absoluto) o por 1's (números muy grandes en valor absoluto).

Exponente con todo 1's

- *Mantisa con todo "0" y signo "0" : +∞*
- *Mantisa con todo "0" y signo "1" : -∞*
- *Mantisa distinta de "0": NaN (**N**ot **a** **N**umber: indeterminado)*

Exponente con todo 0's

- *Mantisa con todo "0" y signo "0" : +0*
- *Mantisa con todo "0" y signo "1" : -0*
- *Mantisa distinta de "0": Números desnormalizados*

Representación de números reales

Coma flotante: norma IEEE 754

Casos especiales: números desnormalizados simple precisión

Imaginemos que queremos representar en la norma IEEE 754 el número decimal $3 \cdot 2^{-130}|_{10}$ en simple precisión.

Siguiendo las pautas que hemos indicado para calcular el exponente $\log_2(3 \cdot 2^{-130}) = -128.415$ ....→ $e = -129$

El exponente -129, esta fuera del rango [-127,128] que marca la norma.

Pero tenemos una alternativa:

- A un exponente con todo 0's y mantisa no todos 0's se le asigna el valor -126.
- Relajemos la condición de que la mantisa tenga la forma $1, \ldots$ de tal forma que en estos casos particulares tenga la forma $0, \ldots$

$$3 \cdot 2^{-130}|_{10} = 11|_2 \cdot 2^{-130} = 0.0011 \cdot 2^{-126}$$

¿Por qué -126?

Dado que una mantisa y exponente con todo 0's se asume como valor 0, el menor valor absoluto representable normalizado en simple precisión es $1.000 \ldots 000|_2 \cdot 2^{\mathbf{-126}}$.

El siguiente menor valor absoluto ya desnormalizado sería $0.111 \ldots 111|_2 \cdot 2^{\mathbf{-126}}$

El menor valor absoluto representable desnormalizado sería $2^{\mathbf{-23}} \cdot 2^{\mathbf{-126}}$

Representación de números reales

Coma flotante: norma IEEE 754

Rangos en simple precisión

Mayor valor en valor absoluto

Exponente: $11111110|_2=254|_{10}-127|_{10}=127|_{10}$

Mantisa: $1111..1111|_2=1.111\ldots111|_2=(2-2^{-23})|_{10}=2x(1-2^{-24})|_{10}$

Por tanto, mayor valor en valor absoluto $2x(1-2^{-24})x2^{127} \approx 3.4x10^{38}$

Menor valor normalizado en valor absoluto

Exponente: $00000001|_2=1|_{10}-127|_{10}=-126|_{10}$

Mantisa: $0000\ldots0000|_2=1.000\ldots000|_2=1.0|_{10}$

Por tanto, menor valor normalizado en valor absoluto $2^{-126} \approx 1.18x10^{-38}$

Menor valor desnormalizado en valor absoluto

Exponente: $00000000|_2 \rightarrow$ por convención$=-126|_{10}$

Mantisa: $0000\ldots0001|_2=0.000\ldots001|_2=2^{-23}|_{10}$

Por tanto, menor valor normalizado en valor absoluto $2^{-23}\ 2^{-126} \approx 1.4x10^{-45}$

Estas tablas no hay que aprenderlas. Simplemente sirven para entender la norma y practicar. Para doble precisión, con desarrollos análogos, puede consultarse la bibliografía.

Representación de números reales

Coma flotante: norma IEEE 754

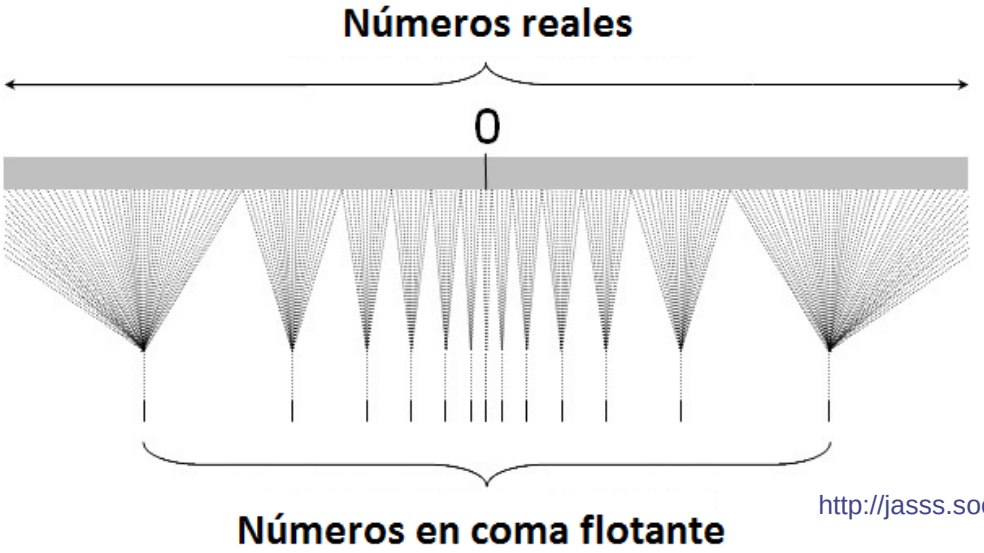
Número de bits de la mantisa

La influencia de disponer de un bit a mayores en la mantisa supone que la distancia entre 2 números consecutivos representables se divide por 2.

> Según esto el término “doble precisión” no es riguroso. La precisión que ofrece es mucho mayor.

Distribución en la recta real

Los números no están uniformemente distribuidos sobre la recta real, sino que están más próximos cerca del origen (exponentes más pequeños) y más separados a medida que nos alejamos de él (exponentes más grandes).



http://jasss.soc.surrey.ac.uk/9/4/4.html

Representación de números reales

Coma flotante: norma IEEE 754

Precisión

Veamos qué pasa representando el valor $0.1_{|10}$ en simple precisión.

1) Transformamos a binario la parte entera y la parte fraccionaria

   $0.1|_{10} = 0.000110011\mathbf{0011}\ldots|_2$

2) Determinamos el signo

   El número es positivo, luego el bit de signo es **0**.

3) Representamos el número según la notación científica

   $0.000110011\ldots = 1.100110011\ldots \cdot 2^{-4}$

4) Identificamos la mantisa (desechamos el 1 implícito) y el exponente

   $1.\mathbf{10011001100110011001101} \cdot 2^{\mathbf{-4}}$

5) Añadimos el exceso **127** al exponente y lo representamos en binario con 8 bits

   $(\mathbf{-4} + \mathbf{127})_{10} = 123_{10} = (64 + 32 + 16 + 8 + 2 + 1)_{10} = \mathbf{01111011_2}$

7) Formamos el número

   00111101100110011001100110011001101

Es 1 el último bit en lugar de 0 por que se redondea al valor más próximo

Hagamos ahora el proceso inverso para comprobar realmente qué número estamos representando.

Representación de números reales

Coma flotante: norma IEEE 754

Precisión

$0.1_{|10}$ → $\mathbf{00111101100110011001100110011001101}_{|2-IEEE\ 754}$

$\mathbf{00111101100110011001100110011001101}_{|2-IEEE\ 754}$ → ¿? $_{|10}$

1) Determinamos el signo

El número es positivo, pues el bit de signo es **0**.

2) Calculamos el exponente

$\mathbf{01111011} = 123$

Eliminando el exceso, $exponente = 123 - 127 = \mathbf{-4}$

3) Calculamos la mantisa añadiendo el bit implícito y desechando 0's superfluos

$\mathbf{10011001100110011001101}$ → $\mathbf{1.10011001100110011001101}$

5) Formamos el número en notación científica $m \cdot 2^e$

$\mathbf{1.10011001100110011001101 \cdot 2^{-4}}$

6) Por comodidad de cálculo, eliminamos el término $\mathbf{2^e}$ desplazando la coma $\boldsymbol{e}$ posiciones (a la derecha si el signo es positivo y a la izquierda si es negativo)

$\mathbf{1.10011001100110011001101 \cdot 2^{-4}} = \mathbf{0.00011001100110011001101101}$

7) Transformamos a base 10 $\mathbf{0.00011001100110011001101_2} =$

$(\mathbf{2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-24} + 2^{-25} + 2^{-27}})_{\mathbf{10}} =$

$= \mathbf{0.100000014901161\ ..._{10}}$ → **¡La representación no es exacta!**

Representación de números reales

Coma flotante: norma IEEE 754

Precisión

Supongamos que en un programa C++ tenemos 3 variables en coma flotante como en el siguiente ejemplo:

```
double x=0.5;
double y=0.1;
double z=0.4;
```

La representación del valor `0.5` es exacta, pero la de `0.1` y `0.4` no. Como consecuencia de esto, `x` es distinto de `y+z`, y, por tanto `x-(y+z)` es distinto de `0`.

Hay que ser muy cuidadosos en nuestros programas cuando el desempeño de éste dependa de condiciones de este tipo. Las consecuencias pueden ser catastróficas.

The Patriot Missile Failure

Representación de números reales

Aritmética en coma flotante

Los procedimientos son más complejos que en la aritmética de enteros y, como es lógico, surgen casos especiales.

Resumiendo:

- Para **sumas** y **restas** tenemos que ajustar al mismo valor los exponentes de los operandos. Esto los programas software o los circuitos electrónicos lo hacen fácilmente pues:
  - multiplicar por 2 (sumar 1 a uno de los exponentes) es desplazar un bit a la izquierda
  - dividir por 2 (restar 1 a uno de los exponentes) desplazar un bit a la derecha.
  - Una vez hecho esto, podemos sumar o restar las mantisas en aritmética de enteros y normalizar el resultado final si es necesario.

- Para **multiplicaciones** y **divisiones** tenemos que sumar o restar los exponentes en aritmética de enteros (y restar el exceso que se produce). Las mantisas se multiplican o dividen en aritmética de enteros y se normaliza el resultado final si es necesario.

Representación de números reales

Reales: representación en C++

En C++ una declaración como

```
float x;
```

reservará espacio en memoria para la variable real **`x`** usando la norma IEEE 754 simple precisión y, por tanto, usando 4 bytes.

La declaración

```
double x;
```

reservará espacio en memoria para la variable real **`x`** usando la norma IEEE 754 doble precisión y, por tanto, usando 8 bytes.

Como hemos comentado, C++ permite diferentes tamaños de almacenamiento para los tipos de datos fundamentales, pero 4 y 8 bytes respectivamente es lo habitual.

Se recomienda usar siempre **`double`** cuando declaremos variables reales en nuestros programas:

- A pesar de usar doble capacidad de almacenamiento, hoy en día gran parte del hardware hace las operaciones más rápidas en precisión doble.
- Como ingenieros, necesitaremos programar algoritmos en los que la precisión puede ser fundamental.

Desbordamiento de la representación

Se produce un **desbordamiento** (***overflow***) cuando tras operar entre dos números, el resultado sale del rango de representación.

Supongamos que disponemos de 4 bits para representar enteros en complemento a 2. Sumemos 6_{10} y 5_{10}.

1	Acarreo
0 1 1 0	6_{10}
0 1 0 1	5_{10}
1 0 1 1	-5_{10}

Vemos que el resultado es totalmente erróneo debido a que para poder representar el verdadero resultado 11_{10} necesitamos al menos 5 bits.

En el caso de las operaciones en coma flotante, el desbordamiento lo veremos al encontrar un exponente con todos sus valores a 1.

Desbordamiento de la representación

En C++ debemos vigilar el desbordamiento, ya sea al operar entre variables o al introducir los datos. Se muestra un ejemplo con enteros.

```
#include <iostream>
using namespace std;
int main()
{
    int x=2147483647; //Limite superior para 4 bytes
    int y=1;
    cout<<x+y<<endl; //Suma es el límite inferior del rango
    x=4294967296;
    cout<<x<<endl; //El valor almacenado es 0
    //Si introducimos un valor fuera de rango, C++ lo trunca
    //al valor superior o inferior del rango
    cout <<"Introduzca un valor superior o inferior al rango:";
    cin>>x;
    cout<<x<<endl;
}
```

Salida:

```
-2147483648
0
Introduzca un valor superior o inferior al rango:3456765478
2147483647
```

Desbordamiento de la representación

En el caso de números reales, si hay desbordamiento, C++ nos mostrará por pantalla la salida `inf`.

```
#include <iostream>
using namespace std;
int main()
{
    cout<<endl;
    float x=2e38; //Limite superior para 4 bytes
    //Producto superior al límite superior del rango
    cout<<2*x<<endl;
    //Si introducimos un valor fuera de rango, C++ lo trunca
    //al valor superior o inferior del rango
    cout<<"Introduzca un valor superior o inferior al rango:";
    cin>>x;
    cout<<x<<endl;
}
```

Salida:

```
inf
Introduzca un valor superior o inferior al rango:45e40
3.40282e+038
```

Representación de caracteres

La representación de caracteres se basa en asociar un código entero a cada carácter.

Aunque hay diferentes codificaciones propuestas, las más importantes son la codificación **ASCII** y la **UNICODE**. Ambas son estándares.

La codificación **ASCII** permite codificar 256 caracteres, por lo que necesita **1 byte** (8 bits) para lograrlo.

La codificación **UNICODE** permite codificar 65536 caracteres, para lo que emplea **2 bytes** (16 bits). UNICODE facilita la transmisión y visualización de textos de múltiples lenguajes (incluso lenguas muertas) y en diferentes disciplinas técnicas. El término Unicode refleja sus objetivos de universalidad, eficiencia y unicidad.

Representación de caracteres

Código ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)

Inicialmente constaba de 128 caracteres. Luego fue extendido para incluir caracteres con tildes y otros símbolos.

Caracteres de control ASCII

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

Caracteres ASCII imprimibles

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

ASCII extendido

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	▀	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

http://www.elcodigoascii.com.ar

Representación de caracteres

Código ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)

Nótese que los caracteres numéricos (0, 1, 2, … ,9) ocupan posiciones consecutivas con códigos desde el 48 al 57. Esto facilita la conversión entre caracteres y números enteros y viceversa, por ejemplo, en las operaciones de entrada/salida.

Carácter 0 → 48 → 0011 0000 → Número 0
Carácter 1 → 49 → 0011 0001 → Número 1
Carácter 2 → 50 → 0011 0010 → Número 2
…
Carácter 9 → 57 → 0011 1001 → Número 9

Las mayúsculas y las minúsculas se diferencian únicamente en un bit: internamente, los programas podrán pasar de minúsculas a mayúsculas y viceversa de forma sencilla.

Carácter A → 65 → 01000001 → 01100001 → 97 → Carácter a
Carácter B → 66 → 01000010 → 01100010 → 98 → Carácter b
…
Carácter Z → 90 → 01011010 → 01111010 → 122 → Carácter z

El hecho de que las posiciones sean consecutivas permiten su ordenamiento numérico o lexicográfico de forma sencilla.

Representación de caracteres

Caracteres: representación en C++

En C++ una declaración como

```
char c;
```

reservará espacio en memoria para la variable **c**, que será un carácter codificado en ASCII, usando 1 byte para su almacenamiento.
Nótese la existencia de los caracteres no imprimibles, como el salto de línea, que usaremos mucho en C++. Para lograr un salto de línea, C++ dispone de la secuencia **'\n'**. Esto permite distinguir entre caracteres numéricos y valores numéricos cuando formamos **cadenas de caracteres** (secuencia de caracteres entre comillas dobles). Las estudiaremos más a fondo más adelante.

```
#include <iostream>
using namespace std;
int main()
{
   char c=10;              //Valor entero ASCII del retorno de línea
   cout <<"Hola"<<c;       //Forma correcta pero no se usa
   cout <<"Hola\n";        //Forma preferida en C++
   cout <<"Hola"<<endl;    //Forma alternativa para el retorno de línea
   cout <<"Hola10";        //Error: imprimiremos Hola10
}
```

Salida:

```
Hola
Hola
Hola
Hola10
```

Conversión entre tipos de datos

C++ es un lenguaje **estáticamente tipado**, lo que significa que los tipos asignados a las variables se comprueban en tiempo de compilación.

> Así, el compilador verifica que las operaciones que el programador ha escrito en el código fuente están soportadas por el lenguaje. Si existe alguna incompatibilidad, el compilador avisará con un mensaje de error.

Por ello, **siempre deben declararse los tipos de las variables antes de ser utilizadas**.

No obstante, los lenguajes estáticamente tipados sí que permiten mezclar tipos de datos en las expresiones a pesar de que sus representaciones internas sean diferentes.

Para trabajar de forma cómoda con estas situaciones, los lenguajes permiten que el programador adopte dos tipos de decisión:

- **Conversión implícita**: El compilador se encarga de forma automática de adaptar las representaciones para poder efectuar los cálculos de forma correcta.
- **Conversión explícita**: El programador escribe explícitamente en el código fuente cómo quiere que se haga la conversión entre las representaciones.

Conversión entre tipos de datos

Conversión implícita en C++

Asignaciones

En las asignaciones se evalúa la expresión a la derecha del operador de asignación **=** y el resultado se convierte implícitamente al tipo de dato situado a la izquierda del operador de asignación.

Cualquier asignación de un tipo de dato no booleano a **`bool`** da un resultado falso si el valor es 0. En caso contrario, el resultado de la asignación es cierto.

```
bool b=48.1; //b es cierto
```

En cualquier asignación de un tipo real a un tipo entero se trunca al entero resultante de eliminar la parte fraccionaria.

```
int r=48.1; //r se almacenara como entero con valor 48

int s=-28.1; //r se almacenara como entero con valor -28
```

En cualquier asignación de un tipo entero a un tipo real éste se convertirá según la norma IEEE 754. Puede conllevar la pérdida de precisión en números *grandes* para los **`float`**, no así para los **`double`**, que tienen suficientes bits (una razón más para usar siempre **`double`**).

```
float x=19999999; //x se almacenara como real 2.0e7

double y=19999999; //y se almacenara como real 19999999
```

Conversión entre tipos de datos

Conversión implícita en C++

Operaciones

Cuando aparece una expresión con distintos tipos de datos mezclados, C++ promueve todos los tipos a un único tipo, el que tiene mas **jerarquía** de entre los presentes. De esta forma, la operación (aritmética o lógica) se realiza según la representación del tipo de mayor jerarquía.

Para los tipos de datos que vamos a manejar en la asignatura, la jerarquía de menor a mayor en C++ es:

```
bool → int
char → int → float → double


int x=1;
double y=19999998;
//x+y es una expresión que se evalúa como double y se
//almacena en un registro interno. Al ejecutarse el
//programa, su valor será 19999999.
float w=x+y;  //w se almacena con valor float  20000000
double z=x+y; //z se almacena con valor double 19999999
```

Conversión entre tipos de datos

Conversión explícita en C++

En ocasiones el programador necesita forzar la conversión.

Imaginemos un problema en el cual deseamos calcular la pendiente media de un puerto de montaña. Para la longitud del puerto y para la altitud se han elegido variables enteras que representan metros.

La fórmula será: $pendiente = \frac{altitud}{longitud}$

Un programa ejemplo podría ser el siguiente:

```
#include <iostream>
using namespace std;
int main()
{
    int longitud=18342;
    int altitud=1123; //Respecto a la base del puerto

    double pendiente=altitud/longitud;

    cout<<"La pendiente es "<<pendiente;
}
```

Salida:

```
La pendiente es 0
```

Conversión entre tipos de datos

Conversión explícita en C++

En el ejemplo observamos un valor no deseado pero lógico.

Las expresiones a la derecha de una asignación = se evalúan antes de realizarse la asignación, almacenándose en un registro interno. En el ejemplo, la operación **`altitud/longitud`** es una expresión que involucra 2 variables enteras y puesto que el resultado *real* es 0.xxxx se trunca a 0. El valor 0 se asigna a la variable **`double pendiente`** que toma el valor 0.

Un programa ejemplo para obtener el valor correcto sería:

```
#include <iostream>
using namespace std;
int main()
{
    int longitud=18342;
    int altitud=1123; //Respecto a la base del puerto

    double pendiente=static_cast<double>(altitud)/longitud;

    cout<<pendiente;
}
```

Salida:

```
La pendiente es 0.0612256
```

Conversión entre tipos de datos

Conversión explícita en C++

La conversión forzada de tipos **`static_cast<tipo_de_dato>(expresion)`** convierte la representación interna de **`expresion`** a la representación marcada por **`tipo_de_dato.`**

En nuestro ejemplo, forzamos a que la representación interna de la variable **`altitud`** sea **`double.`** De esta forma, la expresión **`altitud/longitud`** se convertirá implícitamente a **`double`** y obtendremos el resultado correcto.

Podréis encontrar código con conversiones forzadas tales como:

```
double pendiente=double(altitud)/longitud;
double pendiente=(double)altitud/longitud;//Estilo C
```

Aunque para el ejemplo que nos ocupa todas las opciones son válidas y compilarán perfectamente, se recomienda usar **`static_cast.`**

Por otro lado, en la medida de lo posible, debemos evitar el uso de las conversiones forzadas. En muchas ocasiones podemos evitar su uso. En el ejemplo hubiese bastado con declarar:

```
double longitud=18342;
double altitud=1123;
```

Conversión entre tipos de datos

Las conversiones también conllevan graves riesgos. Un ejemplo clásico es el ocurrido con el cohete Ariane, que se produjo por una conversión entre un número en coma flotante de 64 bits (que era mayor de 32768) a un número entero de 16 bits.

ariane 5 explosion.mp4

Bibliografía

- S. B. Lippman, J. Lajoie, B. E. Moo. C++ Primer. Addison-Wesley. 2012.
- R. E. Bryant, D. R. O'Hallaron Computer Systems. A Programmer's Perspective. Pearson 2010.
- https://es.wikipedia.org/wiki/Sistema_de_numeración
- https://es.wikipedia.org/wiki/UNIVAC_I
- https://en.wikipedia.org/wiki/Two's_complement
- http://www.learncpp.com/cpp-tutorial/24a-fixed-width-integers/
- http://www.h-schmidt.net/FloatConverter/IEEE754.html
- https://en.wikipedia.org/wiki/IEEE_754-1985
- http://www.unicode.org/
- http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html