REPRESENTACIÓN INTERNA DE DATOS

1 Introducción

En el próximo capítulo estudiaremos la forma en que se representan los distintos tipos de objetos con que trabajamos en computación. En los lenguajes de alto nivel manejamos distintos tipos de datos: caracteres, strings, números enteros, números reales. Para trabajar con ellos, en general sólo nos interesa saber que son, como se opera con cada uno y esto se estudia en los cursos de programación. En esta materia nos ocuparemos de otro aspecto: como están implementados, a nivel interno, los distintos tipos de datos.

La unidad elemental de información que se usa en computación es un objeto que toma solo 2 valores posibles (0, 1): el BIT (dígito binario)

Los distintos tipos de datos se construyen en base a estructuras de bits, los cuales serán en general arrays de n elementos que reciben el nombre de **palabra** de largo n. En el caso particular de n = 4 se denomina el array se denomina **nibble**, en el caso de n = 8 el array se denomina **byte**.

Estudiaremos entonces, la representación interna de los datos como la expresión de los distintos tipos en función de estructuras de bits.

Por lo anterior resulta que los distintos tipos de datos se representan a través de **códigos binarios**. Es decir existe un proceso de codificación de los objetos de información en función de otros (las estructuras de bits) con los que se trabajará en realidad.

2 Tipo Carácter

2.1 Introducción

Comenzaremos viendo cómo se representan los caracteres mediante códigos binarios. Entendemos por caracteres los símbolos que se utilizan en el lenguaje natural escrito: letras, números, símbolos de puntuación, símbolos especiales, etc.

La forma en que se codifican los caracteres ha ido evolucionando con el tiempo y han existido, y existen actualmente, múltiples formas distintas de hacerlo. Los sistemas de codificación de caracteres muchas veces dependen de los idiomas y por tanto difieren según las geografías donde se apliquen y su popularidad depende en parte de la importancia geopolítica y económica de dichas regiones.

Durante los 80 y los 90 la forma más difundida de representar estos símbolos es la establecida por el denominado código **ISO-8859-1** para los idiomas de Europa Occidental. Este código fue diseñado por la International Standard Organization en forma original como **ISO/IEC 8859-1** (que tenía menos caracteres definidos), adoptado en forma oficial por la IANA (Internet Assigned Numbers Authority = organismo que regula actualmente la actividad en internet) y fue utilizado por los sistemas operativos UNIX (o derivados). Una variante del ISO/IEC 8859-1 es el **Windows-1252**, utilizado por el sistema operativo Windows.

Actualmente la tendencia es hacia la utilización del UCS (Universal Character Set), que es la parte de especificación de caracteres del proyecto Unicode, el cuál se diseño para ser un estándar universal de cómo representar texto en múltiples idiomas y de múltiples tipos (ej: fórmulas matemáticas) en las computadoras. El Unicode fue adoptado por la ISO bajo la denominación ISO 10646. Existen distintos sistemas de codificación del Unicode. Los más difundidos son el UTF-8, el UTF-16 y el UTF-32. El UTF-8 es un sistema de codificación de largo variable, compatible con el **ASCII** (American Standard Code for Information Interchange), origen también del ISO-8859 y muchos otros sistemas de codificación de caracteres.

2.2 Representación ASCII

El **ASCII** es un código de 7 bits que especifica la representación de las letras y símbolos especiales usados en el idioma Inglés (más exactamente Inglés norteamericano) además de los números, y es muy similar al alfabeto número 5 del **CCITT** (Comité Consultivo Internacional sobre Telefonía y Telecomunicaciones), actualmente **ITU** (International Telecomunication Union), organismo que, entre otras funciones, establece propuestas de estandarización en materia de comunicaciones.

De los 128 valores posibles del código (7 bits) 10 se utilizan para los dígitos decimales (del 30h al 39h), 26 para las letras mayúsculas (del 41h al 5Ah), 26 para letras minúsculas (del 61h al 7Ah), 34 para símbolos especiales (espacio, !, #,$,%,/,&,+,-,*, etc.) y los 32 primeros se denominan genéricamente "caracteres de control" y se utilizan básicamente en la comunicación de datos y con fines de dar formato a los textos en impresoras y pantallas de vídeo.

Algunos de estos caracteres de control son:

00h $\Rightarrow$ NUL (Null) Es la ausencia de información, se utiliza como carácter de relleno.

02h $\Rightarrow$ STX (Start of Text) Muchos protocolos de comunicación lo utilizan para indicar el comienzo de un texto.

03h $\Rightarrow$ ETX (End of Text) Idem para fin de texto.

06h $\Rightarrow$ ACK (Acknowledge) Se utiliza en comunicaciones para contestar afirmativamente la recepción correcta de un mensaje.

15h $\Rightarrow$ NAK (Negative acknowledge) Idem para recepción incorrecta.

0Ah $\Rightarrow$ LF (Line Feed) Indica pasar a la siguiente línea (en una impresora o pantalla).

0Ch $\Rightarrow$ FF (Form Feed) Indica pasar a página siguiente.

0Dh $\Rightarrow$ CR (Carriage Return) Indica volver a la primera posición dentro de la línea.

11h $\Rightarrow$ DC1 (Data Control 1) Indica dispositivo libre (disponible).

12h $\Rightarrow$ DC2 (Data Control 2) Indica dispositivo ocupado (no disponible).

Supongamos que enviamos caracteres a una impresora y ésta no está en condiciones de recibir más (ha llenado su "buffer") entonces la impresora envía el carácter DC2 al computador (para indicarle que no envíe más caracteres), cuando queda en condiciones de recibir nuevos caracteres, envía el carácter DC1. Cuando se utilizan caracteres con este fin, se dice que se hace uso de un protocolo XON/XOFF (Transmit on/Transmit off).

1Bh $\Rightarrow$ ESC (Escape) Indica el comienzo de una "secuencia de escape". Los caracteres que vienen a continuación tienen un significado especial. Estas secuencias se usan típicamente para enviar comandos a las impresoras y/o terminales de visualización. Por ejemplo, para posicionar el cursor en la pantalla, cambiar el tipo de letra en una impresora.

La forma más habitual de representar el código ASCII, y en general todos los sistemas de codificación de caracteres, es a través de una matriz cuyas columnas están asociadas a los 3 bits más significativos del código, y sus filas a los 4 bits menos significativos tal como se muestra en la siguiente tabla:

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

Tabla 2 - Código ASCII

Como ya hemos establecido, el ASCII codifica los caracteres utilizados para escribir textos en idioma Inglés y los signos de puntuación y símbolos especiales propios de dicha lengua. ¿Qué pasa entonces con otras lenguas como el español, en particular?. Cómo se puede observar, la Ñ, por ejemplo, no está codificada, tampoco las letras acentuadas, así como signos de puntuación característicos de nuestro idioma como "¿" y "¡". La situación en este campo fue confusa hasta la última década del pasado siglo y existieron distintas propuestas de codificación aceptadas. Se propusieron dos mecanismos básicos para atacar el problema: uno fue modificar el ASCII de 7 bits adaptándolo a cada lengua; el otro fue utilizar un código de 8 bits, que en sus primeros 128 valores coincidiera con el ASCII y los restantes utilizarlos para representar los caracteres propios de un conjunto relativamente grande de idiomas.

Un ejemplo del mecanismo de modificación del ASCII es el que fue utilizado por EPSON en sus impresoras y que se convirtió en un "estándar de facto" en el mundo de las computadoras personales al ser aceptado y soportado por la casi totalidad de los otros fabricantes de impresoras. El mecanismo consiste en sustituir caracteres (en general símbolos especiales) por los caracteres que le "faltan" al ASCII para adaptarse a cada lengua en particular. Así tenemos un "ASCII español", un "ASCII francés", un "ASCII inglés (de Inglaterra)", etc.

2.3 Representación IBM437

Un ejemplo de código de 8 bits es el que utilizó IBM en sus primeras computadoras personales, el cual puede considerarse también un "estándar de facto". Utiliza los 128 valores más altos para letras y símbolos de otros idiomas y para caracteres gráficos (que permiten dibujar recuadros y cosas similares). Posee también algunas versiones adaptadas a distintas regiones del mundo. La utilizada para la mayoría de los idiomas de Occidente es la **IBM437** (o CP437, por Code Page 437). En la tabla 3 se muestran los caracteres "agregados" al ASCII (incompleta) para esta variante.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	Ç 00C7	ü 00FC	é 00E9	â 00E2	ä 00E4	à 00E0	å 00E5	ç 00E7	ê 00EA	ë 00EB	è 00E8	ï 00EF	î 00EE	ì 00EC	Ä 00C4	Å 00C5
90	É 00C9	æ 00E6	Æ 00C6	ô 00F4	ö 00F6	ò 00F2	û 00FB	ù 00F9	ÿ 00FF	Ö 00D6	Ü 00DC	¢ 00A2	£ 00A3	¥ 00A5	₧ 20A7	ƒ 0192
A0	á 00E1	í 00ED	ó 00F3	ú 00FA	ñ 00F1	Ñ 00D1	ª 00AA	º 00BA	¿ 00BF	⌐ 2310	¬ 00AC	½ 00BD	¼ 00BC	¡ 00A1	« 00AB	» 00BB
B0	░ 2591	▒ 2592	▓ 2593	│ 2502	┤ 2524	╡ 2561	╢ 2562	╖ 2556	╕ 2555	╣ 2563	║ 2551	╗ 2557	╝ 255D	╜ 255C	╛ 255B	┐ 2510
C0	└ 2514	┴ 2534	┬ 252C	├ 251C	─ 2500	┼ 253C	╞ 255E	╟ 255F	╚ 255A	╔ 2554	╩ 2569	╦ 2566	╠ 2560	═ 2550	╬ 256C	╧ 2567
D0	╨ 2568	╤ 2564	╥ 2565	╙ 2559	╘ 2558	╒ 2552	╓ 2553	╫ 256B	╪ 256A	┘ 2518	┌ 250C	█ 2588	▄ 2584	▌ 258C	▐ 2590	▀ 2580
E0	α 03B1	ß 00DF	Γ 0393	π 03C0	Σ 03A3	σ 03C3	µ 00B5	τ 03C4	Φ 03A6	Θ 0398	Ω 03A9	δ 03B4	∞ 221E	φ 03C6	ε 03B5	∩ 2229
F0	≡ 2261	± 00B1	≥ 2265	≤ 2264	⌠ 2320	⌡ 2321	÷ 00F7	≈ 2248	° 00B0	∙ 2219	· 00B7	√ 221A	ⁿ 207F	² 00B2	■ 25A0	NBSP 00A0

Tabla 3 - Código IBM437 de 8 bits (con código Unicode asociado).

2.4 Representación ISO-8859-1

El código **ISO-8859-1** está representado en la tabla 4, incluyendo los caracteres agregados por la codificación **Windows-1252** (también conocida como Windows Latin-1).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80			‚	*ƒ*	„	…	†	‡	ˆ	‰	Š	‹	Œ			
90		‘	’	“	”	•	–	—	˜	™	š	›	œ			Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Tabla 4 - ISO-8859-1 (en las filas 80 y 90 están los caracteres agregados por Windows-1252).

2.5 Representación ISO 10646

Finalmente mencionaremos los códigos de 16 bits que dan soporte a los idiomas de Oriente (Chino, Japonés, Coreano). De estos códigos el más aceptado es el **ISO 10646** el cual coincide con la parte de representación de caracteres del Proyecto Unicode. El **Unicode** pretende codificar texto (no solamente caracteres) y toda clase de símbolos dando soporte a áreas tales como las matemáticas, la lingüística y los lenguajes de programación, entre otras.

3 Tipo String

El tipo STRING es una sucesión de caracteres. Existen varias formas de representación interna. Lo principal a saber es dónde termina la sucesión.

- Una primera manera es emplear largo fijo. Esta representación es demasiado rígida.

- Una segunda manera es reservar un código especial para fin de string. Este código especial no podrá formar parte del string. Por ejemplo, el lenguaje "C" utiliza el NULL para fin de string. Cuando se utiliza combinado con codificación la codificación de los caracteres en ASCII (o similar), esta representación recibe el nombre de ASCIIZ.

- Una tercera manera consiste en convertir los strings en un registro donde el primer campo tiene el largo y el segundo tiene el contenido. El único inconveniente es que la estructura es más compleja. En BASIC se pueden encontrar ejemplos de esta representación.

4 Tipo Natural (entero sin signo)

4.1 Representación Binaria

Los enteros sin signo (siempre positivos, incluyendo el 0) poseen la representación más simple: su código binario coincide con su expresión en base 2 restringida a un número fijo de bits. Se utilizan para contadores, direcciones, punteros y para derivar otros tipos.

Las operaciones elementales de este tipo son las cuatro usuales para los números enteros (+ - * /).

Por ejemplo en la suma de 2 enteros sin signo, se aplica el algoritmo usual para los números binarios. Veamos un par de casos considerando representaciones de 8 bits

Ejemplo 1:

```
      000110000  ⇒  carry final = 0
  25   00011001
+ 74   01001010
 ----  ---------
  99   01100011
```

Ejemplo 2:

```
      111110000  ⇒  carry final = 1
  25   00011001
+234   11101010
 ----  ---------
 259   00000011  ⇒  la representación no es correcta
```

Los bits de carry (acarreo) de las operaciones anteriores (000110000 y 111110000) se presentan en la primera línea de la operación. Notar que si el ultimo bit de acarreo (es el denominado bit de acarreo o *carry* de la operación) es 1, el código binario resultante no representa al resultado de la operación. En ese caso (carry = 1) ha sucedido un *overflow*, que significa que nos hemos salido del rango de la representación.

El problema principal en esta representación, que es común a todas las basadas en un tamaño fijo de palabra, es que estamos restringidos en el tamaño y las operaciones pueden, por tanto, dar overflow (en este caso indicado por el carry en 1).

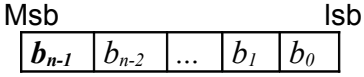
Los tamaños usuales para esta representación son: el byte (0 a 255), la palabra de 2 bytes (16 bits, 0 a (2^{16})-1), la palabra de 4 bytes (32 bits, 0 a (2^{32})-1) y, actualmente, la palabra de 8 bytes (64 bits, 0 a (2^{64})-1). En general el rango de la representación para n bits será:

$$0 \leqslant N \leqslant 2^n - 1$$

5 Tipo Entero (con signo)

5.1 Representación Valor Absoluto y Signo

Si tenemos n bits para representar el número, tomamos uno para el signo y el resto representa el valor absoluto del número en binario (expresión del numero en base 2).



b_{n-1} = *Signo.*
$b_{n-2} \ldots b_1 b_0$ = *Valor absoluto*

- Si b_{n-1} = 1 entonces el número es negativo.
- Si b_{n-1} = 0 entonces es positivo.

Ejemplo: (en 4 bits)

$0110 \Rightarrow 6$
$1110 \Rightarrow -6$

Para n bits el rango del número representado es:

$$-(2^{n-1} - 1) \leq N \leq 2^{n-1} - 1$$

En esta representación tenemos dos formas de representar el cero, 1000 y 0000 (para n=4). Esto puede verse como un inconveniente. Además las operaciones no trabajan directamente con la representación sino que deben interpretarse en base a los signos relativos.

5.2 Representación Complemento a Uno

Los números positivos se representan en binario, y los números negativos se representan como el valor absoluto complementado bit a bit.

Para n bits el rango del número representado es:

$$-(2^{n-1} - 1) \leq N \leq 2^{n-1} - 1$$

$n = 8$

	msb	Lsb
$5 \Rightarrow$	0000	0101
$-5 \Rightarrow$	1111	1010

Ejemplo: ($n = 4$)

-7	*1000*
-6	*1001*
-5	*1010*
-4	*1011*
-3	*1100*
-2	*1101*
-1	*1110*
0	*0000*
0	*1111*
1	*0001*
2	*0010*
3	*0011*
4	*0100*
5	*0101*
6	*0110*
7	*0111*

El orden en binario (interpretando los códigos como si fueran números en base 2, aunque no lo son) no corresponde al orden de los números que representa. Otra desventaja, es que hay 2 representaciones distintas para el cero (*0000 y 1111*).

5.3 Representación Desplazamiento

La representación por desplazamiento supone un corrimiento de los valores a representar según un valor *d* (llamado desplazamiento), y posteriormente se la aplica el módulo para que se pueda almacenar en el tamaño de la representación deseada. Para el desplazamiento, se supone que el valor codificado (resultado de la operación *N + d*), es un número que para n bits es un valor entre *0* y 2^n-1, por lo que permite representar valores desde *–d* a 2^n-d-1.En general para representar 2^n números diferentes, se asigna a *d* el valor 2^{n-1}, o $2^{n-1}-1$ aplicando un módulo de 2^n.

$N \Rightarrow (N + d)$ *representado con n bits*

Ejemplo: $n=4$, $d= 8$

$-8 \Rightarrow 0000$
$-7 \Rightarrow 0001$
....
$-1 \Rightarrow 0111$
$0 \Rightarrow 1000$
$1 \Rightarrow 1001$
....
$7 \Rightarrow 1111$

La propiedad más importante de esta representación es que los códigos conservan el orden de los números, con o sin signo. En particular, toda representación de un número negativo es menor que cualquiera de un número positivo. Otra ventaja, es que hay una sola representación para el cero.

El principal inconveniente de esta representación es que los algoritmos de las operaciones usuales son más complejos

5.4 Representación Complemento a Dos

Los números positivos se representan directamente en binario y para conseguir el código de los negativos, se complementa el valor absoluto y se los incrementa en uno.

$$N \Rightarrow N_2 \qquad si\ N \geq 0$$

$$N \Rightarrow not\ (N_2) + 1 = \overline{N} \qquad si\ N < 0$$

Por ejemplo, para números de 8 bits se tiene:

Si $70 = 01000110$, entonces para lograr -70 hago el complemento a uno (negación bit a bit) de esta configuración y luego le sumo uno, luego:

$$\begin{array}{l} 01000110\ (70) \\ 10111001\ (\text{complemento a 1}) \\ \qquad +1 \\ \hline 10111010\ (-70) \end{array}$$

Las propiedades más importantes de esta representación son:

- Mantiene la suma (la suma con o sin signo es la misma operación, es decir que el algoritmo es el mismo). Es decir la suma de las representaciones da la representación de la suma de los números representados, sean estos positivos o negativos.

- Es coherente la representación del cero:

$$\begin{array}{l} 00000000 \rightarrow 0 \\ 11111111 \rightarrow \text{not } 0 \\ \qquad +1 \\ \hline 00000000 \end{array}$$

  repr(0) = repr(-0)

- Se pierde la relación de orden. El algoritmo de comparación de A con B depende de los signos de A y de B.

- La resta se hace sumando el negativo del sustraendo, con lo que también se mantiene.

  A - B = A + (-B) = A + not B + 1

Al igual que en toda representación de largo fijo las operaciones pueden generar *overflow.* Pero a diferencia del caso de la representación binaria (entero sin signo) en este caso el bit de carry no indica por sí solo esta condición. Veamos algunos ejemplos ilustrativos de los distintos casos posibles:

Ejemplo 1:

```
            000110000  ⇒  carry final = 0
   25        00011001
+  74        01001010
  ----      ---------

   99        01100011  ⇒  representación correcta (overflow = 0)
```

Ejemplo 2:

```
            111110000  ⇒  carry final = 1
   25        00011001
+ -22        11101010
  ----      ---------

    3        00000011  ⇒  representación correcta (overflow = 0)
```

Ejemplo 3:

```
            011100000  ⇒  carry final = 0
   25        00011001
+ 114        01110010
  ----      ---------

  139        10001011  ⇒  -117 representación incorrecta (overflow = 1)
```

Ejemplo 4:

```
            100010100  ⇒  carry final = 1
 -120        10001000
+ -22        11101010
  ----      ---------

 -142        01110010  ⇒  114 representación incorrecta (overflow = 1)
```

En esta representación para saber si hubo *overflow* al final de la operación hay que verificar la existencia de acarreo en los dos bits más significativos. Dejamos a cargo del lector determinar la expresión lógica del *overflow* en función de dichos acarreos.

Para el caso de la multiplicación sucede algo paradójico: si bien la representación mantiene también dicha operación, la forma usual de implementar la operación no lo hace.

Veamos primero con ejemplos la primer parte de la afirmación:

Ejemplo 1:

```
   25       00011001
x   3       00000011
 ____     __________

   15       00011001
   6       00011001
 ____     __________

   75       01001011   ⇒ representación correcta
```

Ejemplo 2:

```
  -22       11101010
x   3       00000011
 ____     __________

    6       11101010
   6       11101010
 ____     __________

  -66       10111110   ⇒ representación correcta
```

El problema es que la multiplicación binaria se implementa normalmente dando el resultado en el doble de bits que los operandos. Esto es: si los operandos son de ***n*** bits, el resultado de la multiplicación se calcula en ***2n*** bits. Esto es porque de otra manera rápidamente nos iríamos de representación aunque multiplicáramos operandos pequeños.

Al extender el resultado de la operación a 2n bits, la propiedad de mantener la multiplicación deja de ser cumplida por la representación complemento a 2. De allí es que se diga habitualmente que la representación complemento a 2 mantiene la suma pero no la multiplicación (aunque estrictamente no sea correcta la afirmación).

De hecho la propiedad del mantenimiento de ambas operaciones (suma y multiplicación) se desprende del hecho que en el fondo la aritmética de la representación de complemento a 2 tiene un vínculo muy estrecho con la aritmética del módulo.

Veamos este punto. Primero observemos que para n bits, el complemento a 1 de un numero cumple con:

$$N + \overline{N} = 2^n - 1$$

Por tanto:

$$N + \overline{N} + 1 = 2^n$$

o sea:

$$N + \overline{N} = 2^n \Rightarrow -N = -1 \times 2^n + \overline{N}$$

dicho de otra manera el complemento a 2 de un número es lo que le falta para llegar a 2^n (por ejemplo en 8 bits, el complemento a 2 de 22 es 234, el complemento a 2 de 117 es 139 y así). La segunda parte es una forma equivalente de plantear la primera, de forma de evidenciar lo que viene a continuación.

Consideremos ahora la representación de un número N cualquiera. Si N es positivo se cumple siempre que:

$N = 0 \times 2^n + N$

y si fuera negativo se cumplirá que:

$-N = -1 \times 2^n + \overline{N}$

Entonces en ambos casos la representación del número coincide con el resto de la división entera del numero entre 2n. Es una propiedad conocida que la aritmética del resto (aritmética de módulo) mantiene las operaciones de suma y multiplicación, siempre que se tenga la precaución de tomar el módulo (resto de la división entera) de la operación realizada entre los módulos. Esto equivale a quedarse con los n bits menos significativos del resultado. Como vimos esto se hace en la práctica con la suma, pero no se hace con la multiplicación (ya que se extiende a 2n bits), por lo que la propiedad solo se aplica a la suma en el caso práctico.

5.5 Representación Decimal

Se usan para sumar cantidades de muchos dígitos donde no se puede perder precisión. En aplicaciones comerciales se tiene un caso típico. Pueden ser del largo que se quiera y se representan internamente en forma similar a un string formado por los dígitos de su expresión en base 10, incluyendo el signo de "-" al comienzo si es negativo.

La codificación típica es utilizando ASCII para los dígitos, signo y punto (o cualquier otro sistema de codificación de caracteres compatible con él):

0 -> 00110000
1 -> 00110001
.....
9 -> 00111001

Observamos que en la representación en 8 bits para dígitos, se usan 10 de los 256 códigos, es decir es muy ineficiente.

Los algoritmos para las operaciones son similares a los empleados en las operaciones decimales hechas a mano. Para sumar, por ejemplo, primero se deben alinear los strings de modo que los finales estén enfrentados. Se deben rellenar hacia la izquierda el mas corto con "0" para que los dos strings tengan el mismo largo. Luego se recorre de derecha a izquierda dígito a dígito. Se debe obtener el valor del dígito de cada sumando a partir de su representación como carácter. Si la codificación es ASCII se puede hacer una operación de "máscara" (AND bit a bit) con 0x0F, ó restar el carácter '0' (si el lenguaje lo permite) ó utilizar la función "ordinal" o similar que algunos lenguajes poseen (que recibe el carácter como argumento y devuelve el dígito que representa como número entero).

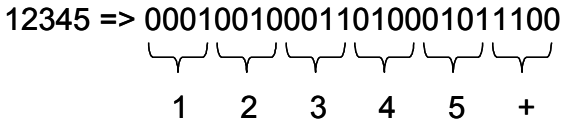
Luego que se obtiene la pareja de dígitos se realiza la operación (teniendo en cuenta los signos de los sumandos). Si el resultado es mayor que nueve (para el caso de suma) se resta 10 y se acarrea 1 para la suma de los próximos dígitos. Si el resultado es menor que cero (para la resta), se suma 10 y se acarrea un -1 para la resta de la siguiente pareja de dígitos.

Al finalizar el proceso se deben quitar los 0 a la izquierda del número para terminar de darle formato al string que representará al resultado de la operación.

5.6 Representación Decimal Empaquetado (BCD)

En la representación Decimal se emplean 8 bits para cada lugar, cuando solamente necesita 4 bits. Por esta razón se define la representación Decimal Empaquetado, en los cuales se codifica con 4 bits cada dígito, ocupando así un solo "nibble" ("nibble" es el conjunto de 4 bits). También se les llama Packed BCD (Binary Coded Decimal).

Vemos un ejemplo de codificación:



El nibble más a la derecha se utiliza como marca de fin e indica el signo del número. El código 1100 (C hexadecimal) indica el fin de un positivo, mientras que el 1101 (D) indica el fin de un negativo. A veces también se utiliza el código 1111 (F) para indicar el fin (sería sin signo, es decir positivo).

Para operar con estas representaciones es necesario alinear los números (para que los dígitos del mismo peso queden enfrentados) y luego se suma dígito a dígito teniendo en cuenta el signo. Para obtener cada dígito recurro a técnicas de "máscara" (AND con 0x0F) para obtener el dígito del nibble inferior y de desplazamiento (SHIFT) a la derecha (4 bits) para obtener el dígito del nibble superior. En el resto el algoritmo es como el descripto para el caso de la representación Decimal. Luego que se tienen los dos dígitos de un byte del resultado se arma el byte desplazando el dígito de la parte alta del byte a la izquierda (4 bits) y haciendo un OR con el dígito de la parte baja del byte.

6 Tipo Fraccionario

6.1 Representación Decimal con parte fraccional

Es una extensión de la representación Decimal, donde además se representa el punto decimal con el carácter ".".

En este caso para sumar primero se deben alinear los strings de modo que los puntos decimales estén enfrentados. Se deben rellenar hacia la derecha y/o izquierda con "0" para que los dos strings tengan el mismo largo. Luego el algoritmo es análogo al ya descripto para la representación Decimal.

6.2 Representación Punto Fijo

Para representar números con parte fraccional se puede utilizar un número determinado de bits (ej: 16, 32, 64), asignando un cierto número *fijo* de bits para representar la parte entera del número y el resto para la parte fraccional. La parte entera siempre ocupa los bits más significativos del código.

La representación se obtiene representando en Complemento a 2 el número resultante de multiplicar N por 2^f, siendo f el número de bits dedicados a la parte fraccional.

Para sumar y restar se opera directamente con las representaciones y el resultado es consistente. Para la multiplicación se requiere corregir el resultado dividiendo entre 2^f. Para la división se debe multiplicar el resultado por dicho número.

6.3 Representación Punto Flotante

En las distintas arquitecturas de computadoras se han utilizado diversas representaciones para expresar números reales, todas ellas basadas en la siguiente notación :

$$N = (-1)^s . b^e . M$$

donde: s es el signo
e es el exponente
M es la mantisa
b es la base de representación

Las bases utilizadas han sido, normalmente, 10 y 2. Dado que esta representación es ambigua (existen varias representaciones para un mismo numero) se utiliza una versión más restringida que se llama **normalizada**. Los números normalizados son aquellos en que el bit más significativo de la mantisa es distinto de cero o, lo que es equivalente, son aquellos en que la mantisa sea máxima.

6.3.1 Estándar IEEE 754 de punto flotante

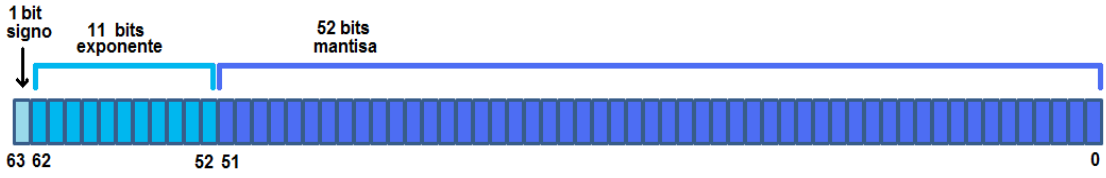
Para que los números representados en punto flotante fueran posibles intercambiar entre distintas arquitecturas se establece el estándar IEEE 754 que define el formato y las operaciones con estos. El estándar IEEE 754 usa la base 2 (b = 2) y su mantisa normalizada es de la forma 1,F. La representación utiliza el signo, el exponente (codificado usando representación desplazamiento) y la parte fraccional de la mantisa (lo que está después de la “coma binaria”). En definitiva el número se representa por la terna de códigos binarios S (codificación del signo),E (codificación del exponente en desplazamiento),F (codificación de la parte fraccional de la mantisa en binario).

El estándar define tres formatos (en función de la cantidad de bits utilizados):

	S (bits)	E (bits)	F (bits)	Total (bytes)
Simple Precisión	1	8	23	4
Doble Precisión	1	11	52	8
Precisión Extendida	1	15	64	10

La “Precisión Extendida” se usa para resultados intermedios de operaciones, pero no para almacenamiento permanente.

Los números se almacenan de la siguiente forma :



En el curso usaremos también la "Media Precisión" (no se usa en la práctica) que tiene:

	S (bits)	E (bits)	F (bits)	Total (bytes)
Media Precisión	1	5	1	2

Los números normalizados son de la forma : $(-1)^s$ 1,F 2^e, donde el bit más significativo de la mantisa es un 1. Como todos los números normalizados tienen un uno en el bit más significativo el estándar define una representación que omite este bit (para optimizar). Esta consiste en: un 1 implícito, una coma implícita y luego la parte "fraccional" de la mantisa.

Por lo tanto la representación a utilizar es de la forma :

$N' = (-1)^s . 2^{e+127} . (1, F)$ para números de simple precisión
$N' = (-1)^s . 2^{e+1023} . (1, F)$ para números de doble precisión

El conjunto de valores posibles pueden ser divididos en los siguientes:

- cero
- números normalizados
- números desnormalizados
- infinitos
- NaN (¬E, no es un número, como por ejemplo, la raíz cuadrada de un número negativo)

Las clases se distinguen principalmente por el valor del campo “E” (exponente), siendo modificada ésta por el campo “F” (fracción). Cada clase se representa con los siguientes rangos de valores:

	E (Exponente)	F (Fracción)
Normalizados	0000….0 < Exp < 1111….1	Cualquier combinación
Desnormalizados	0000……..0	$\neq 0$
Cero	0000……..0	0
Infinito	1111.....1	0
Not a Number	1111.....1	$\neq 0$

Los números desnormalizados sirven para operar con números menores que el menor número normalizado representable. Estos números asumen un 0 implícito en vez del 1 implícito de los números normalizados. Por lo tanto cuando tenemos un número en notación punto flotante desnormalizado estamos representando el número (ejemplo para simple precisión):

$(-1)^s . 2^{e+127} . (0, F)$ donde el exponente coincide con el mas negativo normalizado

6.3.2 Operaciones en Representación Punto Flotante

Las operaciones (suma, resta, multiplicación) se realizan mediante algoritmos especializados para la representación.

Para la suma se debe:

- alinear los exponentes (llevando el menor al mayor), adecuando concordantemente la mantisa (teniendo en cuenta el 1 omitido)
- sumar las mantisas teniendo en cuenta los signos
- normalizar, acomodando la mantisa y el exponente, si corresponde

Para la multiplicación se debe:

- determinar el signo en base a los signos de los operandos
- sumar los exponentes
- multiplicar las mantisas
- normalizar, acomodando la mantisa y el exponente, si corresponde

6.3.3 Normalización y Pérdida de Información

Como vimos los números normalizados son de la forma: 1,F. donde el bit más significativo de la mantisa es un 1. Todos los números deben ser representados en su forma normalizada (siempre que se pueda) y los resultados finales de operaciones se deben normalizar.

El método de normalización utilizado es mover la coma a la parte más significativa de la cifra, es decir, variando el peso aritmético de los dígitos que lo componen.

Ejemplo: Representación de un decimal a binario en punto flotante (precisión simple):

Se codificará el número decimal -118,625 usando el sistema de la IEEE 754.

Dado que es un número negativo, el signo es "1".

Primero, hay que escribir el número (sin signo) usando su expresión en base 2. El resultado es 1110110,101. Luego se mueve la coma “binaria” a la izquierda, dejando sólo un dígito a su izquierda.

$$1110110{,}101 = 1{,}110110101 \times 2^6.$$

Esto es un número en punto flotante normalizado.

La parte fraccional de la mantisa es la parte a la derecha de la coma binaria, rellenada con ceros a la derecha hasta obtener los 23 bits. Es decir

$$F = 11011010100000000000000.$$

El exponente es 6, que debe ser representado en Desplazamiento. Para el formato IEEE 754 de 32 bits, el desplazamiento es 127, así es que 6 + 127 = 133. En binario, esto se escribe como 10000101.

$$E = 10000101$$

El número queda entonces:

S	E	F
1	10000101	11011010100000000000000

6.3.4 Perdida de Precisión

Una característica de esta representación es que el conjunto de números representados no es continuo, pudiendo existir "huecos" importantes. Esto lleva que en una suma puede suceder que el resultado coincida con el mayor de los operandos. Es fácil de ver si se piensa qué pasa si el numero de bits a correr hacia la derecha (para alinear los exponentes) hace que el bit más significativo sea menos significante que el bit menos significativo de la otra mantisa: al sumar y tomar los bits mas significativos sólo se estarán considerando los bits de la mantisa del número mayor.

7 Tipo Moneda

Para representar cantidades monetarias se suele utilizar una representación del tipo "punto fijo", con un numero suficiente de dígitos luego de punto fraccional de forma de darle cierta precisión a las operaciones financieras hasta el nivel de los "centavos".

Una forma alternativa de verlo es pensar que lo que se almacena es la parte entera de la cantidad multiplicada por una potencia de 10. Por ejemplo el BASIC de Microsoft define el tipo "currency" como un entero de 64 bits que resulta de multiplicar la cantidad original por 10.000.

Es importante tener en cuenta que representaciones con precisión variable, como el Punto Flotante, no son buenas representaciones para este tipo de información.