Programación I

Organización de Computadoras

Segunda Parte

Temas de la segunda parte

- Representación de datos ←
- Números enteros
- Operaciones aritméticas
- Punto fijo
- Punto flotante

Representación de datos

Las computadoras almacenan datos e instrucciones en memoria.

Para ello utilizan el sistema binario.

Razones :

- El dispositivo se encuentra en uno de dos estados posibles (0 ó 1)
- Identificar el estado es más fácil si sólo hay dos

Representación de datos

- Ejemplo :
    - lámpara encendida o apagada
    - lámpara encendida con 10 intensidades distintas

- Es más fácil conocer el “estado” de la lámpara en el primer caso (encendida o apagada), que determinar alguna de las 10 intensidades distintas

- Como contrapartida, no es humanamente intuitivo dado que estamos acostumbrados a cuantificar las cosas en decimal.

Representación de datos

Las computadoras manejan varios tipos básicos de datos binarios

- Números enteros sin/con signo
  - -50; 0; 3; 85
- Números reales con signo
  - -34,5; 78,33; 3,14
- Caracteres (codificación o *encoding*)
  - ASCII
  - ISO 8859-X
  - UTF-8

Representación de datos. Datos alfanuméricos.

Representación alfanumérica.

El conjunto de caracteres abarca:

- Letras (‘A’ … ‘Z’ y ‘a’ … ‘z’)
- Dígitos decimales (‘0’, ..., ‘9’) ← OJO: 0 <> ‘0’
- Signos de puntuación (:, ;, ,, .)
- Caracteres especiales (#, $, %, !, SP, etc.)
- Ordenes de control (CR, LF, ESC, NUL, etc.)

Representación de datos. Datos alfanuméricos.

La representación alfanumérica simplemente requiere **un código binario para cada caracter**:

A mayor número de bits por carácter, mayor cantidad de caracteres distintos se podrán representar.

- Con 3 bits podré representar 8 (2^3) caracteres distintos
- Con 4 bits podré representar 16 (2^4) caracteres distintos
- Con 7 bits podré representar 128 (2^7) caracteres distintos
- Con 8 bits podré representar 256 (2^8) caracteres distintos

Representación de datos. Codificaciones.

Algunos ejemplos de codificación:

- **FIELDATA**
    - Código de 6 bits → Total 64 combinaciones
    - 26 letras mayúsculas + 10 dígitos + 28 caracteres especiales
- **ASCII**: American Standard Code for Information Interchange
    - FIELDATA + minúsculas + CONTROL
    - Código de 7 bits → Total 128 combinaciones
- **ASCII Extendido**
    - ASCII + multinacional + semigráficos + matemáticos
    - Código de 8 bits → Total 256 combinaciones
- **EBCDIC**: Extended BCD Interchange Code
    - Similar al ASCII pero de IBM
    - Código de 8 bits → Total 256 combinaciones
- **UTF-8**
    - Codificación de longitud variable

Representación de datos. ASCII Extendido.

ASCII control characters

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)

ASCII printable characters

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	_			

theASCIIcode.com.ar

Extended ASCII characters

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	ö	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	

Representación de datos. UTF-8

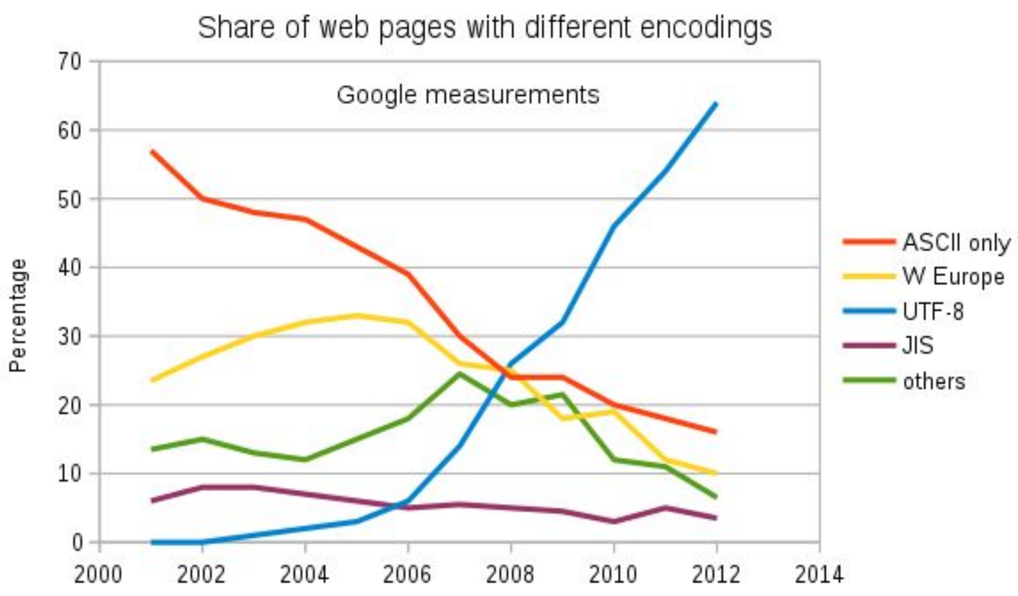
UTF-8

Permite codificación de longitud variable (hasta 4 bytes), lo que permite representar más de 1.000.000 de caracteres distintos.

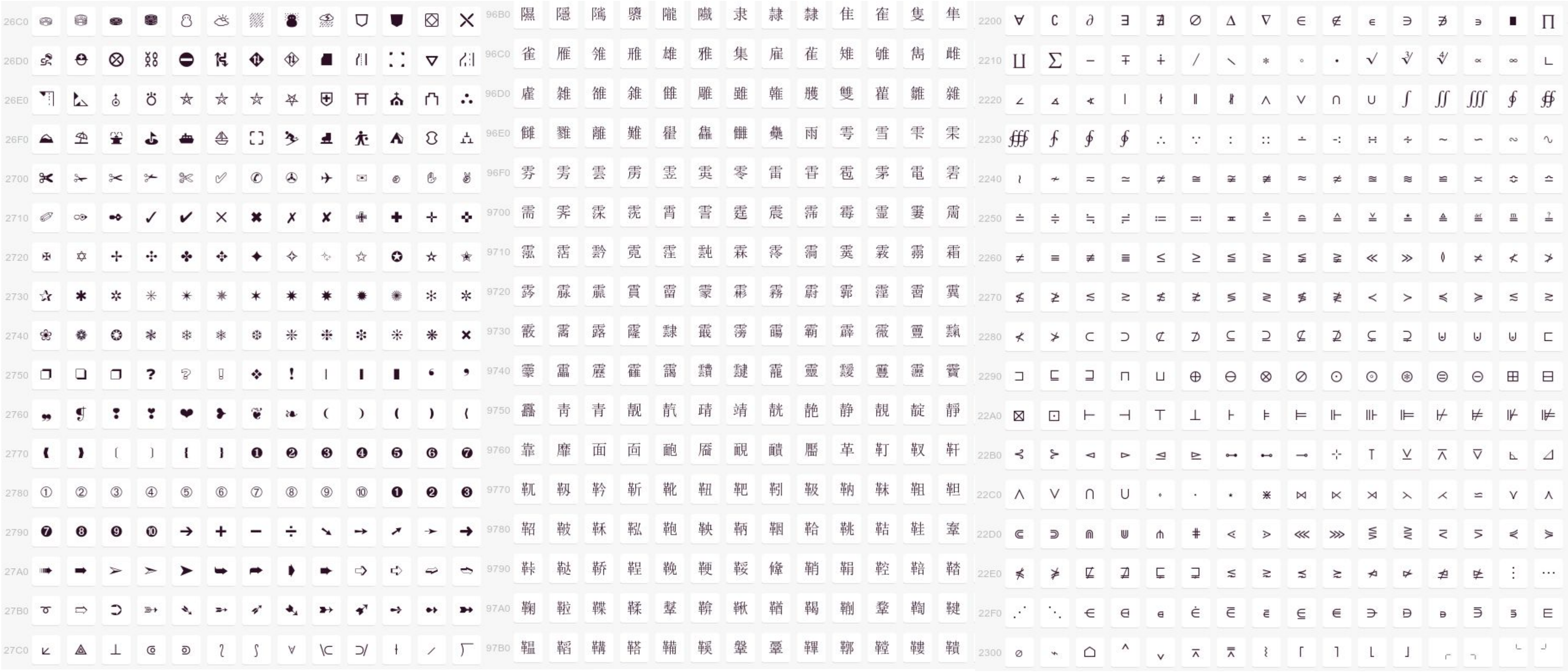
Fue concebido considerando la retrocompatibilidad con ASCII.

Es actualmente el formato de codificación más utilizado.

Solo hay codificada una porción del total posible de caracteres: Unicode v11 (Junio 2018): 137.374 caracteres.



Representación de datos. UTF-8



http://www.unicode.org https://unicode-table.com

Representación de datos. Números.

La representación alfanumérica es mayormente sencilla de comprender. Sin embargo...

¿Cómo representar/almacenar valores decimales en binario?

¿Cómo representar/almacenar un signo negativo, una coma decimal?

¿Qué valor decimal representa la siguiente secuencia de dígitos binarios?

1001

Representación de datos. Números.

¿**1001** qué valor en decimal representa?

a) 9
b) 4,5
c) 2,25
d) -1
e) -0,5
f) -0,25
g) -6
h) -7
i) 1

Representación de datos. Números.

¿**1001** qué valor en decimal representa?

a) 9 ← BSS
b) 4,5 ← BSS, 1 bit para fracción
c) 2,25 ← BSS, 2 bits para fracción
d) -1 ← BCS
e) -0,5 ← BCS, 1 bit para fracción
f) -0,25 ← BCS, 2 bits para fracción
g) -6 ← Ca1
h) -7 ← Ca2
i) 1 ← Ex2

Necesitamos saber el sistema de representación utilizado para poder determinar qué valor se encuentra almacenado en la secuencia de dígitos binarios especificada.

Temas de la segunda parte

- Representación de datos
- Números enteros ←
- Operaciones aritméticas
- Punto fijo
- Punto flotante

Representación de números enteros

- Sin signo
  - Binario sin signo (**BSS**)
- Módulo y signo
  - Binario con signo (**BCS**)
- Complemento a la base reducida
  - Complemento a uno (**Ca1**)
- Complemento a la base
  - Complemento a dos (**Ca2**)
- Exceso
  - Exceso base 2 (**Ex2**)

Sistemas posicionales

Teorema Fundamental de la Numeración

$$N° = \sum_{i=-m}^{n} (dígito)_i \times (base)^i$$

$$... + x_4 \times B^4 + x_3 \times B^3 + x_2 \times B^2 + x_1 \times B^1 + x_0 \times B^0 + x_{-1} \times B^{-1} + x_{-2} \times B^{-2} + ...$$

N° es el valor decimal de una cantidad expresada en base **B** a lo largo de las posiciones **i**-ésimas comprendidas entre **-m** y **n**

Sistemas posicionales

Sistema decimal: Base 10

Digitos {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

8574	= 8000	+ 500	+ 70	+ 4	
	$= 8 \times 10^3$	$+ 5 \times 10^2$	$+ 7 \times 10^1$	$+ 4 \times 10^0$	
3,1416	= 3	+ 0,1	+ 0,04	+ 0,001	+ 0,0006
	$= 3 \times 10^0$	$+ 1 \times 10^{-1}$	$+ 4 \times 10^{-2}$	$+ 1 \times 10^{-3}$	$+ 6 \times 10^{-4}$

Sistemas posicionales

Sistema binario: Base 2

Digitos {0, 1}

$\mathbf{1001{,}1_2}$	=	1×2^3	$+\ 0 \times 2^2$	$+\ 0 \times 2^1$	$+\ 1 \times 2^0$	$+\ 1 \times 2^{-1}$
	=	8	+ 0	+ 0	+ 1	+ 0,5
	=	$9{,}5_{10}$				

Sistemas posicionales

Sistema hexadecimal: Base 16

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

10,11,12,13,14,15

$$\mathbf{2CA,8_{16}} = 2 \times 16^2 + C \times 16^1 + A \times 16^0 + 8 \times 16^{-1}$$

$$= 512 + 192 + 10 + 0,5$$

$$= 714,5_{10}$$

BCH. Hexadecimal Codificado en Binario

Dígito hexadecimal	Código BCH
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

Abstracción para simplificar la lectura, dado que 4 dígitos binarios se agrupan en un solo dígito hexadecimal.

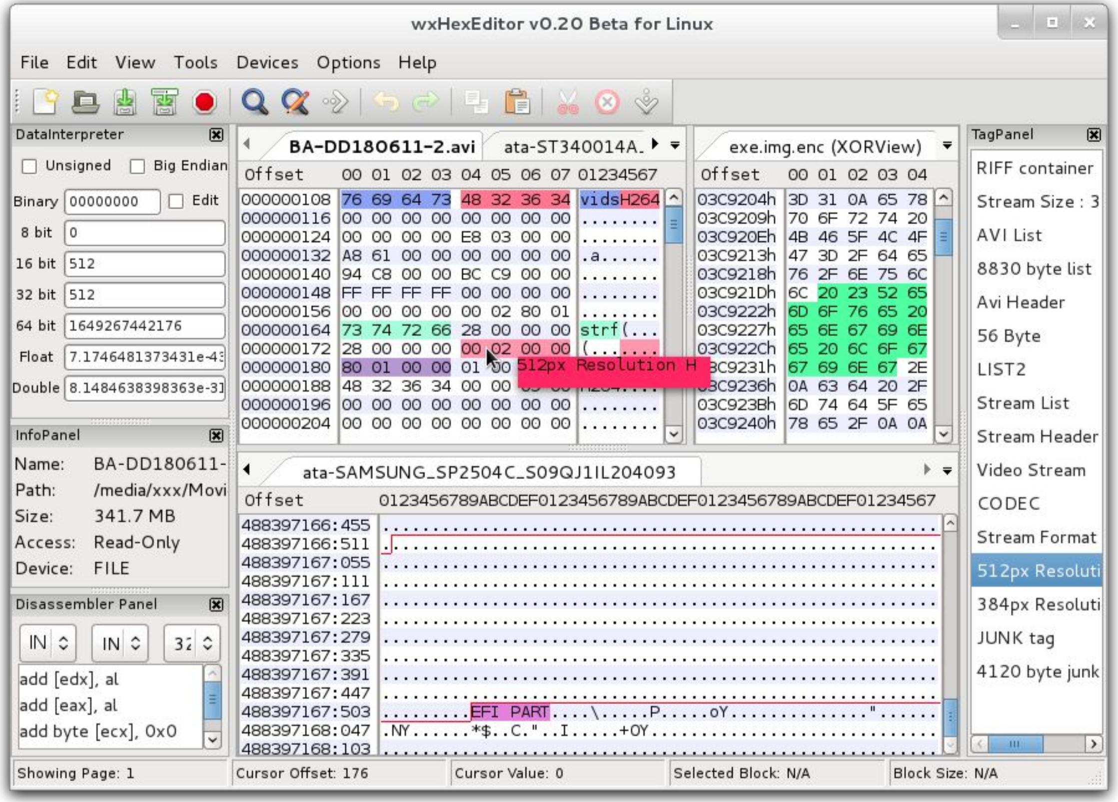
Por ejemplo, la cadena $\mathbf{AB9D_{16}}$ es más sencillo de leer e interpretar que $\mathbf{1010101110011101_2}$

BCH. Hexadecimal Codificado en Binario

Dígito hexadecimal	Código BCH
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

BCH. Hexadecimal Codificado en Binario

Dígito hexadecimal	Código BCH
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



BCD. Decimal Codificado en Binario

Dígito decimal	Código BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

De las 16 combinaciones con 4 bits, se utilizan 10 para representar los dígitos del 1 al 9. Sobran 6 combinaciones de 4 bits para utilizarlos según el ámbito de aplicación (sincronismo, operadores aritméticos, etc.)

Sistema de representación:

- Los dígitos decimales se convierten uno a uno en binario
- Para representar un dígito decimal se requerirán 4 bits
- Se asocia cada dígito con su valor en binario puro

Dos ámbitos de aplicación:

- <u>E/S y periféricos</u>, los números se codifican usando **8 bits por dígito**. Se dice que el número está desempaquetado.
- <u>En cálculo</u>, se reservan **4 bits por dígito**. Se dice que el número está empaquetado.

BCD. Decimal Codificado en Binario

Dígito decimal	Código BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

De las 16 combinaciones con 4 bits, se utilizan 10 para representar los dígitos del 1 al 9. Sobran 6 combinaciones de 4 bits para utilizarlos según el ámbito de aplicación (sincronismo, operadores aritméticos, etc.)

E/S y Periféricos

Por cada dígito se usan 8 bits, 4 para el binario puro y 4 se completan con “1”

834 = 11111000 11110011 11110100 = F8 F3 F4

Cálculo

Ciertas combinaciones se usan para operadores

C_{16} = 1100 representa al signo +

D_{16} = 1101 representa al signo -

BCD tiene usos específicos y NO fue concebido para maximizar la capacidad de representación en función del número total de combinaciones.

Enteros sin signo. Binario Sin Signo (BSS).

Si el número tiene **n** bits, es posible representar $\mathbf{2^n}$ números distintos.

El **rango** comprende desde **0** hasta $\mathbf{2^n – 1}$

Ejemplo con **3** bits: $\mathbf{2^3}$ = **8** números distintos. Rango: **0 ..** $\mathbf{2^3}$**-1** = **0 .. 7**

000_2	0
001_2	1
010_2	2
011_2	3
100_2	4
101_2	5
110_2	6
111_2	7

Enteros sin signo. Binario Sin Signo (BSS).

Ejemplo con **8** bits: $\mathbf{2^8}$ = **256** números distintos. Rango: $\mathbf{0 .. 2^8-1}$ = **0 .. 255**

00000000_2	0
00000001_2	1
00000010_2	2
...	...
01111111_2	127
10000000_2	128
10000001_2	129
...	...
11111110_2	254
11111111_2	255

Enteros sin signo. Binario Sin Signo (BSS).

En BSS es posible representar la misma cantidad de números distintos que combinaciones posibles de bits.

Ejemplo: Con 8 bits se pueden realizar 256 combinaciones.
En BSS con 8 bits se pueden representar 256 números distintos.

No todos los sistemas maximizan la capacidad de representación en función del número de combinaciones posibles.

Enteros con signo. Binario Con Signo (BCS).

¿Cómo representar valores negativos en una computadora?
¿Cómo almacenar el signo - (menos)?
Debemos “robar” un bit a la magnitud para poder almacenar el signo

Bit 7 (signo)	Bit 6 (magnitud)	Bit 5 (magnitud)	Bit 4 (magnitud)	Bit 3 (magnitud)	Bit 2 (magnitud)	Bit 1 (magnitud)	Bit 0 (magnitud)

- Un **0** en el bit de signo indica que el número es **positivo**
- Un **1** en el bit de signo indica que el número es **negativo**

Los bits **0** a **n-2** representan el valor absoluto en binario.

Ejemplos: $\mathbf{1010_2 = -2}$ $\mathbf{0010_2 = 2}$ $\mathbf{1111_2 = -7}$ $\mathbf{0111_2 = 7}$

El rango: $\mathbf{-(2^{n-1}-1)\ ..\ (2^{n-1}-1)}$ con 2 ceros: **0000** y **1000**

Enteros con signo. Binario Con Signo (BCS).

Ejemplo con **3** bits: $\mathbf{2^3-1}$ = **7** números distintos. Rango: -**($\mathbf{2^2}$-1) .. $\mathbf{2^2}$-1** = **-3 .. 3** (intervalo **<u>simétrico</u>**)

000_2	0
001_2	1
010_2	2
011_2	3
100_2	-0
101_2	-1
110_2	-2
111_2	-3

Enteros con signo. Binario Con Signo (BCS).

Ejemplo con **8** bits: $\mathbf{2^8-1}$ = **255** números distintos. Rango: -**($\mathbf{2^7}$-1) .. $\mathbf{2^7}$-1** = **-127 .. 127** (intervalo **<u>simétrico</u>**)

00000000_2	0
00000001_2	1
00000010_2	2
...	...
01111111_2	127
10000000_2	-0
10000001_2	-1
...	...
11111110_2	-126
11111111_2	-127

Enteros con signo. Binario Con Signo (BCS).

En BCS no es posible representar la misma cantidad de números distintos que combinaciones posibles de bits.

Ejemplo: Con 8 bits se pueden realizar 256 combinaciones.
En BCS con 8 bits se pueden representar 255 números distintos.

BCS **no** maximiza la capacidad de representación en función del número de combinaciones posibles, debido al doble cero.

Enteros con signo. Binario Con Signo (BCS).

Utilizado en ciertas (pocas) computadoras de vieja data

IBM 7090

Enteros con signo. Complemento a 1 (Ca1).

Complemento: El complemento a un número **N** de un número **A** es igual a la cantidad que le falta a **A** para ser **N** (A menor que N)

“ Complemento a N de A = N - A ”

Complemento a 26 de 3 = 23

Complemento a 12 de 4 = 8

Complemento a 7 de 7 = 0

¿Cómo aplicar este concepto para definir números negativos?

Enteros con signo. Complemento a 1 (Ca1).

Definición: Si un número binario está expresado en Ca1:

- Si el bit más significativo es 0, el número es positivo y se interpreta como BSS.
- Si el bit más significativo es 1, el número es negativo y se interpretará el Ca1 del valor.

Ca1 → ¿Qué valor de N utilizar?

Para cualquier número con **n** cantidad de dígitos binarios, **N** se define como: $\mathbf{N = 2^n - 1}$ (complemento a la base reducida o **Ca1**)

- Si n = 1 → $\mathbf{N = 2^1 - 1 = 1}$
- Si n = 2 → $\mathbf{N = 2^2 - 1 = 3}$
- Si n = 3 → $\mathbf{N = 2^3 - 1 = 7}$
- Si n = 4 → $\mathbf{N = 2^4 - 1 = 15}$

Enteros con signo. Complemento a 1 (Ca1).

- Si n = 1 → $\mathbf{N = 2^1 - 1 = 1}$
- Si n = 2 → $\mathbf{N = 2^2 - 1 = 3}$
- Si n = 3 → $\mathbf{N = 2^3 - 1 = 7}$
- Si n = 4 → $\mathbf{N = 2^4 - 1 = 15}$

Ejemplo acotado a 3 bits:

- 010 (2 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **2**
- 110 (6 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca1 de 6: 7 - 6 = 1
  - Valor: **-1**

Enteros con signo. Complemento a 1 (Ca1).

- Si n = 1 → **$N = 2^1 - 1 = 1$**
- Si n = 2 → **$N = 2^2 - 1 = 3$**
- Si n = 3 → **$N = 2^3 - 1 = 7$**
- Si n = 4 → **$N = 2^4 - 1 = 15$**

Ejemplo acotado a 4 bits:

- 0101 (5 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **5**
- 1001 (9 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca1 de 9: 15 - 9 = 6
  - Valor: **-6**

Enteros con signo. Complemento a 1 (Ca1).

- Si n = 1 → **N = 2^1 - 1 = 1**
- Si n = 2 → **N = 2^2 - 1 = 3**
- Si n = 3 → **N = 2^3 - 1 = 7**
- Si n = 4 → **N = 2^4 - 1 = 15**

Ejemplo acotado a 4 bits:

- 0111 (7 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **7**
- 1111 (15 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca1 de 15: 15 - 15 = 0
  - Valor: **-0**

Enteros con signo. Complemento a 1 (Ca1).

Una técnica más sencilla

Leer el Ca1 de una secuencia de dígitos binarios simplemente implica “invertir” los dígitos de dicho número, y luego leerlo como BSS (adicionando previamente el -)

De esta manera no hay necesidad de “determinar” el valor de N en función de n.

Aplicamos la nueva aproximación en los ejemplos anteriores:

- Nro negativo expresado en Ca1 → Invierto bits → Leo en BSS → Agrego el signo -
- 110 → 001 → 1 → **-1**
- 1001 → 0110 → 6 → **-6**
- 1111 → 0000 → 0 → **-0**

Enteros con signo. Complemento a 1 (Ca1).

Números a representar igual que en BCS. Ejemplo con **3** bits: $\mathbf{2^3-1}$ = **7** números distintos.

000_2	0
001_2	1
010_2	2
011_2	3
100_2	-3
101_2	-2
110_2	-1
111_2	-0

El rango es el mismo que en BCS: -$\mathbf{(2^2-1)}$ **..** $\mathbf{2^2-1}$ = **-3 .. 3** (intervalo **simétrico**)

Enteros con signo. Complemento a 1 (Ca1).

Ejemplo con **8** bits: $\mathbf{2^8-1}$ = **255** números distintos. Rango: -$\mathbf{(2^7-1)}$ **..** $\mathbf{2^7-1}$ = **-127 .. 127** (intervalo **simétrico**)

00000000_2	0
00000001_2	1
00000010_2	2
...	...
01111111_2	127
10000000_2	-127
10000001_2	-126
...	...
11111110_2	1
11111111_2	-0

Enteros con signo. Complemento a 1 (Ca1).

En Ca1 no es posible representar la misma cantidad de números distintos que combinaciones posibles de bits.

Ejemplo: Con 8 bits se pueden realizar 256 combinaciones.
En Ca1 con 8 bits se pueden representar 255 números distintos.

Presenta el mismo problema del doble cero que BCS.

Al igual que el BCS, el Ca1 **no** maximiza la capacidad de representación en función del número de combinaciones posibles.

Entonces… ¿para qué sirve?

Enteros con signo. Complemento a 1 (Ca1).

Igualmente sirve como un sistema de representación de números negativos.

Varios computadores lo han utilizado.

CDC 6600

UNIVAC 1100

PDP-1

Adicionalmente sirve como introducción a Complemento a 2.

Enteros con signo. Complemento a 2 (Ca2).

Complemento a 2 (Ca2)

Por su definición, Ca2 es una mejora con respecto a Ca1 dado que no presenta el problema del doble cero.

Ca2 es un sistema de representación muy utilizado en computadoras para representar números de punto fijo negativos.

Enteros con signo. Complemento a 2 (Ca2).

Definición: Si un número binario está expresado en Ca2:

- Si el bit más significativo es 0, el número es positivo y se interpreta como BSS.
- Si el bit más significativo es 1, el número es negativo y se interpretará el Ca2 del valor.

Ca2 → ¿Qué valor de N utilizar?

Para cualquier número con **n** cantidad de dígitos binarios, **N** se define como: $\mathbf{N = 2^n}$ (complemento a la base o **Ca2**)

- Si n = 1 → $\mathbf{N = 2^1 = 2}$
- Si n = 2 → $\mathbf{N = 2^2 = 4}$
- Si n = 3 → $\mathbf{N = 2^3 = 8}$
- Si n = 4 → $\mathbf{N = 2^4 = 16}$

Enteros con signo. Complemento a 2 (Ca2).

- Si n = 1 → **N = 2^1 = 2**
- Si n = 2 → **N = 2^2 = 4**
- Si n = 3 → **N = 2^3 = 8**
- Si n = 4 → **N = 2^4 = 16**

Ejemplo acotado a 3 bits:

- 010 (2 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **2**
- 110 (6 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca2 de 6: 8 - 6 = 2
  - Valor: **-2**

Enteros con signo. Complemento a 2 (Ca2).

- Si n = 1 → $\mathbf{N = 2^1 = 2}$
- Si n = 2 → $\mathbf{N = 2^2 = 4}$
- Si n = 3 → $\mathbf{N = 2^3 = 8}$
- Si n = 4 → $\mathbf{N = 2^4 = 16}$

Ejemplo acotado a 4 bits:

- 0101 (5 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **5**
- 1001 (9 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca2 de 9: 16 - 9 = 7
  - Valor: **-7**

Enteros con signo. Complemento a 2 (Ca2).

- Si n = 1 → **N = 2^1 = 2**
- Si n = 2 → **N = 2^2 = 4**
- Si n = 3 → **N = 2^3 = 8**
- Si n = 4 → **N = 2^4 = 16**

Ejemplo acotado a 4 bits:

- 0111 (7 en decimal)
  - Inicia con 0. Se interpreta positivo.
  - Se lee como BSS. Valor: **7**
- 1111 (15 en decimal)
  - Inicia con 1. Se interpreta negativo.
  - Se lee el Ca2 de 15: 16 - 15 = 1
  - Valor: **-1** ← (En Ca1 era el -0)

Enteros con signo. Complemento a 2 (Ca2).

Una técnica más sencilla

Leer el Ca2 de una secuencia de dígitos binarios simplemente implica interpretar primeramente el Ca1 y luego restarle 1.

(Notar que - por definición de complementos a la base y a la base reducida - Ca2 tiene un delta de 1 unidad con respecto a Ca1).

Aplicamos la nueva aproximación en los ejemplos anteriores:

- Nro negativo expresado en Ca2 → Interpreto en Ca1 → Resto 1
- 110 → -1 → **-2**
- 1001 → -6 → **-7**
- 1111 → -0 → **-1** ← No más doble cero!

Enteros con signo. Complemento a 2 (Ca2).

Ejemplo con **3** bits: $\mathbf{2^3}$ = **8** números distintos (contra 7 de BCS y Ca1)

000_2	0
001_2	1
010_2	2
011_2	3
100_2	-4
101_2	-3
110_2	-2
111_2	-1

El rango es mayor al de Ca1 : **-(2^2) .. 2^2-1** = **-4 .. 3** (intervalo **<u>asimétrico</u>**)

Enteros con signo. Complemento a 2 (Ca2).

Ejemplo con **8** bits: $\mathbf{2^8}$ = **256** números distintos. Rango: -$\mathbf{(2^7)}$ **..** $\mathbf{2^7}$**-1** = **-128 .. 127** (intervalo **asimétrico**)

00000000_2	0
00000001_2	1
00000010_2	2
...	...
01111111_2	127
10000000_2	-128
10000001_2	-127
...	...
11111110_2	2
11111111_2	-1

Enteros con signo. Complemento a 2 (Ca2).

En Ca2 sí es posible representar la misma cantidad de números distintos que combinaciones posibles de bits.

Ejemplo: Con 8 bits se pueden realizar 256 combinaciones.
En Ca2 con 8 bits se pueden representar 256 números distintos.

No presenta el problema del doble cero como Ca1 y BCS.

A diferencia de BCS y Ca1, **Ca2 maximiza** la capacidad de representación en función del número de combinaciones posibles.

Enteros con signo. Exceso a 2 (Ex2).

Técnica del exceso

- La representación de un número **A** en exceso es la que corresponde a la SUMA del mismo y un valor constante E (o exceso).
    - **Exceso E de A = A + E**

- Dado un valor en exceso, el número representado se obtiene RESTANDO el valor del exceso.
    - **A = (Exceso E de A) - E**

Enteros con signo. Exceso a 2 (Ex2).

Técnica del exceso

Simplificado:

- Si quiero expresar un número en exceso, sumo el exceso.
- Si quiero interpretar un número expresado en exceso, resto el exceso.

- Ejemplo: Sea E = 5.
  - Representar 3 en exceso: 3 + 5 = **8**
  - Representar -4 en exceso: -4 + 5 = **1**
  - Interpretar 2 (expresado en exceso): 2 - 5 = **-3**
- Al contrario de BCS, Ca1 y Ca2; Ex2 **NO** sigue la regla del bit más significativo

Enteros con signo. Exceso a 2 (Ex2).

Esta técnica simplemente desplaza todos los números representables en función el exceso E definido.

En Ex2 binario, **E** está definido por $\mathbf{2^{n-1}}$, donde **n** es el número de bits.

Ejemplos:

- Si $n=3 \rightarrow E = 2^{3-1} = 4$
- Si $n=4 \rightarrow E = 2^{4-1} = 8$
- Si $n=8 \rightarrow E = 2^{8-1} = 128$

Enteros con signo. Exceso a 2 (Ex2).

- Si n=3 → $E = 2^{3-1} = 4$
- Si n=4 → $E = 2^{4-1} = 8$
- Si n=8 → $E = 2^{8-1} = 128$

Ejemplos **acotado a 4 bits**. El valor resultante luego de aplicar el exceso se escribe en BSS. La interpretación al leer un número expresado en exceso se hace en BSS.

- Expresar 1 en Exceso: 1 + 8 = 9 → $\mathbf{1001_2}$
- Expresar 3 en Exceso: 3 + 8 = 11 → $\mathbf{1011_2}$
- Expresar -2 en Exceso: -2 + 8 = 6 → $\mathbf{0110_2}$
- Interpretar $\mathbf{0000_2}$ expresado en Exceso: 0 - 8 = -8 → **-8**
- Interpretar $\mathbf{1111_2}$ expresado en Exceso: 15 - 8 = 7 → **7**
- Interpretar $\mathbf{1000_2}$ expresado en Exceso: 8 - 8 = 0 → **0**

Enteros con signo. Exceso a 2 (Ex2).

Dado que **Ex2 simplemente transporta los números representables a un intervalo diferente** de la recta numérica, es posible representar la misma cantidad de números distintos que combinaciones posibles de bits.

Ejemplo: Con 8 bits se pueden realizar 256 combinaciones.
En Ex2 con 8 bits se pueden representar 256 números distintos.

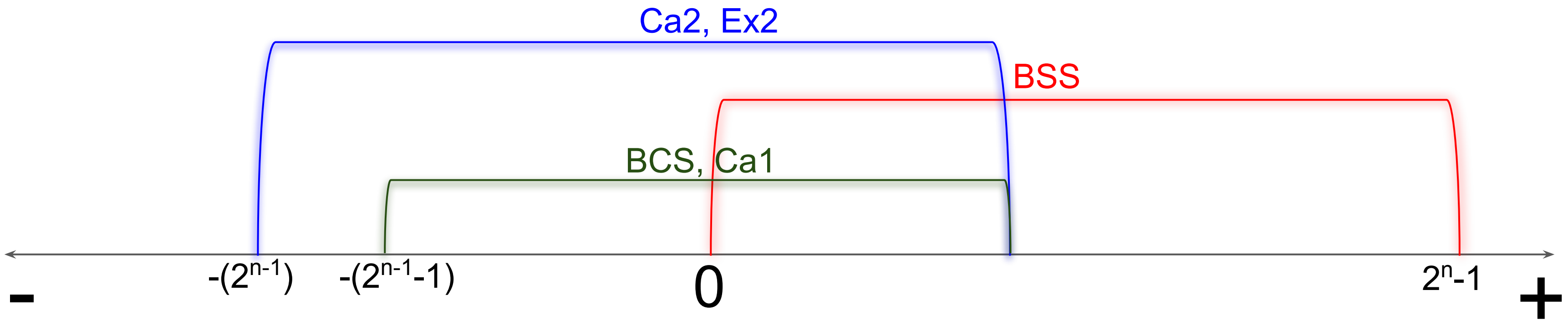
No presenta el problema del doble cero como Ca1 y BCS.

A diferencia de BCS y Ca1, **Ex2 maximiza** la capacidad de representación en función del número de combinaciones posibles.

Resumen de sistemas

Secuencia de bits	**BSS** Nros: 2^n Rango: $0..2^n-1$	**BCS** Nros: 2^n-1 Rango: $-(2^{n-1}-1)..2^{n-1}-1$	**Ca1** Nros: 2^n-1 Rango: $-(2^{n-1}-1)..2^{n-1}-1$	**Ca2** Nros: 2^n Rango: $-(2^{n-1})..2^{n-1}-1$	**Ex2** Nros: 2^n Rango: $-(2^{n-1})..2^{n-1}-1$
000	0	0	0	0	-4
001	1	1	1	1	-3
010	2	2	2	2	-2
011	3	3	3	3	-1
100	4	-0	-3	-4	0
101	5	-1	-2	-3	1
110	6	-2	-1	-2	2
111	7	-3	-0	-1	3

Resumen de sistemas

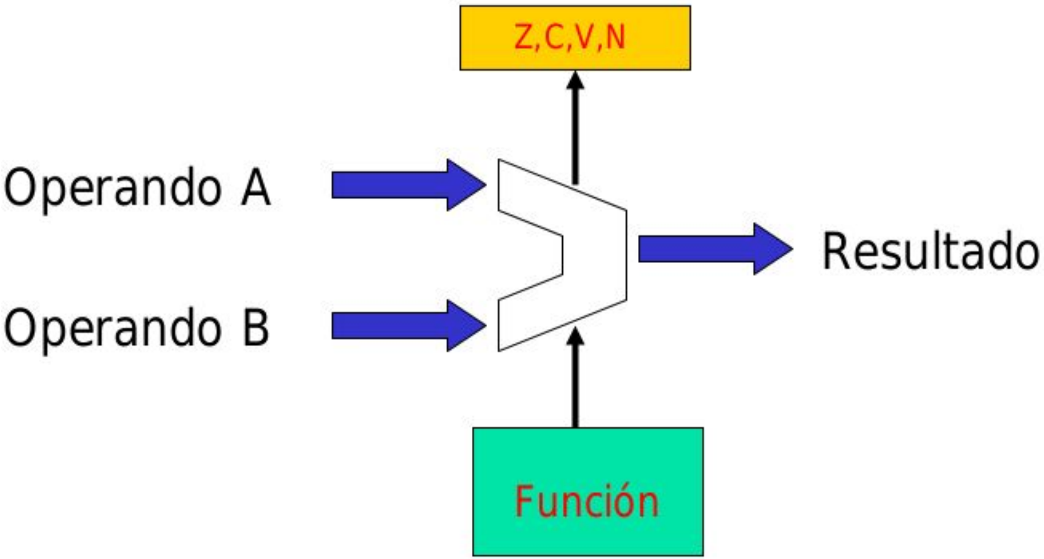


Temas de la segunda parte

- Representación de datos
- Números enteros
- Operaciones aritméticas ←
- Punto fijo
- Punto flotante

Operaciones aritméticas

- La **ALU** realiza operaciones lógicas y aritméticas.
- Recibe los operandos y la operación a realizar
- Devuelve el resultado y las bits de condición (flags)



Operaciones aritméticas.

Las sumas y restas en binario son similares que en decimal.
"Me llevo uno" y "le pido al compañero" siguen vigentes.

```
  0110          0011
+ 0100        + 0111
  ----          ----
  1010          1010

  0101          0100
- 0001        - 0001
  ----          ----
  0100          0011
```

1

0 ~~10~~ 10

0~~1~~00

\- 0001

\----

0011

Operaciones aritméticas. Flags aritméticos

¿Qué son los flags?

Son bits que el procesador establece de modo automático acorde al resultado de cada operación realizada.

Sus valores permitirán tomar decisiones como:

- Realizar o no una transferencia de control.
- Determinar relaciones entre números (mayor, menor, igual).

Operaciones aritméticas. Flags aritméticos

- **Z** (zero): vale 1 si todos los bits del resultado de una operación son cero

```
  0010                    0001
- 0010                  + 0010
  ----                    ----
  0000 → Z = 1               0011 → Z = 0
```

Es de utilidad para - por ejemplo - realizar comparaciones por igualdad.

Si quiero comparar la igualdad de 2 variables A y B; simplemente puedo restar ambos valores, y si el flag de Z es 1, entonces son iguales.

Operaciones aritméticas. Flags aritméticos

- **N** (negativo): vale 1 si el bit más significativo del resultado también es 1.

```
  0010                      0001
- 0010                    + 1010
  ----                      ----
  0000 → N = 0                  1011 → N = 1
```

Es de utilidad para - por ejemplo - realizar comparaciones por desigualdad.

Si quiero comparar la desigualdad de 2 variables A y B; simplemente puedo restar A-B, y si el flag de N es 1, entonces B es mayor que A.

Solo es de utilidad su análisis si estamos trabajando en un sistema con signo.

Operaciones aritméticas. Flags aritméticos

- **C** (carry): flag de acarreo
  - vale 1 si en una suma hay acarreo del bit más significativo
  - vale 1 si en una resta hay *'borrow'* hacia el bit más significativo

```
  1000                    0000
+ 1010                  - 1000
  ----                    ----
  0010 → C = 1               1000 → C = 1
```

Sirve para determinar si no se puede representar el resultado de la operación con el número de bits utilizado, principalmente en BSS.

¿Qué ocurre si sumamos 1?

Operaciones aritméticas. Flags aritméticos

- **V** (overflow): flag de desbordamiento: Indica una condición fuera de rango en Ca2. Casos:
  - Si sumamos 2 positivos y el resultado es negativo
  - Si sumamos 2 negativos y el resultado positivo
  - Si a un positivo le restamos un negativo el resultado es negativo
  - Si a un negativo le restamos un positivo el resultado es positivo

```
  1000
+ 1010
  ----
  0010 → V = 1
```

```
  1000
- 0001
  ----
  0111 → V = 1
```

Sirve para determinar si no se puede representar el resultado de la operación con el número de bits utilizado en sistemas con signo, ppalmente Ca2.

Operaciones aritméticas. Flags aritméticos

Tanto **C** (para BSS) como **V** (para Ca2) implican errores de desbordamiento debido a que el resultado **no puede ser representado con el número de bits** que se está utilizando.

Se requieren más bits para representar el resultado de manera correcta en el sistema dado.

Veamos a continuación algunos **ejemplos**. En todos los casos se realiza primero la operación matemática binaria a nivel de bits, se determinan los flags y **recién luego se interpretan los valores** en los sistemas de representación.

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC          Ca2    BSS

  0100        0000             4      4
+ 0010                       + 2    + 2
  ----                       ---    ---
  0110                         6      6
```

- Los dos resultados son correctos

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC         Ca2    BSS

  0101       1010              5      5
+ 0111                       + 7    + 7
  ----                       ---    ---
  1100                        -4     12
```

- Ca2 incorrecto, BSS correcto

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC          Ca2    BSS

  1101        0101            -3      13
+ 0011                       + 3     + 3
  ----                       ---     ---
  0000                         0       0
```

- Ca2 correcto, BSS incorrecto

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC          Ca2    BSS

  1001        0011            -7       9
+ 1100                       +-4     +12
  ----                       ---     ---
  0101                         5       5
```

- Ambos resultados incorrectos

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC          Ca2   BSS

  0101       1001              5     5
- 0111                       - 7   - 7
  ----                       ---   ---
  1110                        -2    14
```

- Ca2 correcto, BSS incorrecto

Operaciones aritméticas. Flags aritméticos

```
  Operación     NZVC          Ca2   BSS

  1001       0010           -7     9
- 0100                    -  4  -  4
  ----                    ---   ---
  0101                       5     5
```

- Ca2 incorrecto, BSS correcto

Operaciones aritméticas. Flags aritméticos

Recordar:

Las operaciones sobre bits son **siempre las mismas** sin importar el sistema de representación utilizado. Ejemplo: $1_2 + 1_2 = 10_2$ siempre.

La CPU recibe los operandos y operación a realizar, para luego devolver el resultado y flags **sin importar el sistema de representación utilizado**.

El análisis de uno u otro flag será de utilidad **dependiendo el tipo sistema en el que están representados los valores**.

Ejemplo: No es de utilidad analizar los flags V o N si estamos representando números en BSS.

Errores de desbordamiento “famosos”

Pac-Man Kill Screen Bug

Al llegar al nivel 255 del pacman, caos total en pantalla.

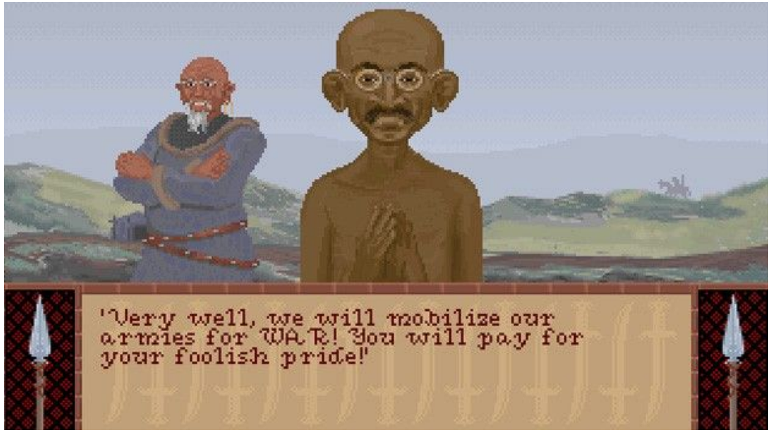
1. `LD A,(#4E13): Load 255 into A`
2. `INC A: Increment 255 to 256 -` **`this overflows to 0`**
3. `CP #08: Is 0 less than 8? Yes.`
4. `JP NC,#2C2E: Don't jump to large-number fruit handling code`
5. `LD DE,#3B08: Load the address of the cherry into DE`
6. `Start the fruit-drawing loop`

https://retrocomputing.stackexchange.com/questions/1640/why-does-the-kill-screen-glitch-occur-in-pac-man
http://www.gamasutra.com/view/feature/3938/the_pacman_dossier.php?print=1

El código no contempla la posibilidad de que al llegar al nivel 255 y sumar 1, el contador de niveles (un dato de 8 bits únicamente), desborda a 0. Esto hace que el render del nivel se comporte de manera errática.

Errores de desbordamiento “famosos”

Gandhi ultraviolento en Civilization



Gandhi fue un pacifista, pero en el juego Civilization declaraba la guerra y atacaba otras civilizaciones con bombas nucleares.

Motivo: todos los líderes del juego poseen un **nivel de agresión**. Gandhi por ser pacifista se le asignó el menor valor posible entre todos los líderes: 1.

Pero si un jugador adopta la democracia, el nivel de agresión se reduce en **2 unidades** para todos los líderes.

Esto genera un underflow no contemplado del valor de la agresión a 255!

https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Errores de desbordamiento “famosos”

Gangnam Style INT_MAX overflow

Alojaba el número de reproducciones en contador de 32 bits.

2^{32} = 4.294.967.296 números, de los cuales la mitad son positivos y la mitad negativos, con lo cual el rango comprende los valores -2.147.483.648 y 2.147.483.647

Alcanzada la reproducción 2.147.483.647+1, el contador de desborda a -2.147.483.648.

Resuelto al convertir el tipo de dato a 64 bits. Nuevo desborde al alcanzar 9.223.372.036.854.775.807 (9 trillones) de visitas.

https://arstechnica.com/information-technology/2014/12/gangnam-style-overflows-int_max-forces-youtube-to-go-64-bit/

Temas de la segunda parte

- Representación de datos
- Números enteros
- Operaciones aritméticas
- Punto fijo ←
- Punto flotante

Punto fijo

En los sistemas de punto fijo, se considera que todos los números a representar tienen exactamente la misma cantidad de dígitos y la coma fraccionaria está siempre ubicada en el mismo lugar.

Por ejemplo, en sistema decimal: **0,23**; **5,12**; **9,11**

En cada caso, los números tienen tres dígitos, y la coma está a la derecha del más significativo

Punto fijo

En sistema binario es el mismo concepto: **11,10**; **01,10**; **00,11**

Hay **4 dígitos** y la coma está entre el segundo y tercer dígito.

La diferencia principal entre la representación en el papel y su almacenamiento en computadora, es que no se guarda coma alguna, se supone que está en un lugar determinado.

Punto fijo

Rango y resolución

Rango: para un sistema de representación dado, el **rango** contempla el intervalo de valores representable entre el mayor y el menor de éstos

Resolución: para un sistema de representación dado, la **resolución** es la diferencia mínima posible entre dos números consecutivos

Punto fijo

Para el ejemplo anterior en sistema decimal:

- **Rango** es de 0,00 a 9,99 o bien **[0,00 .. 9,99]**
- **Resolución** es 0,01: 2,32 - 2,31 = **0,01** o bien 9,99 - 9,98 = **0,01**

Si desplazamos la coma dos lugares a la derecha:

- **Rango** es de 0 a 999 o bien **[0 .. 999]**
- **Resolución** es 1: 3 - 2 = **1** o bien 999 - 998 = **1**

→ Notar que hay un compromiso entre rango y resolución ←

En cualquiera de los casos hay $\mathbf{10^3}$ números distintos: El número de combinaciones entre los dígitos **es siempre la misma** más allá de la posición de la coma decimal.

Punto fijo

Binario	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

4 bits parte entera
0 bits parte fraccionaria

Rango: 0 .. 15
Resolución: 1

Punto fijo

Binario	Decimal
000,0	0
000,1	0,5
001,0	1
001,1	1,5
010,0	2
010,1	2,5
011,0	3
011,1	3,5
100,0	4
100,1	4,5
101,0	5
101,1	5,5
110,0	6
110,1	6,5
111,0	7
111,1	7,5

3 bits parte entera
1 bit parte fraccionaria

Rango: 0 .. 7,5
Resolución: 0,5

Punto fijo

Binario	Decimal
00,00	0
00,01	0,25
00,10	0,5
00,11	0,75
01,00	1
01,01	1,25
01,10	1,5
01,11	1,75
10,00	2
10,01	2,25
10,10	2,5
10,11	2,75
11,00	3
11,01	3,25
11,10	3,5
11,11	3,75

2 bits parte entera
2 bits parte fraccionaria

Rango: 0 .. 3,75
Resolución: 0,25

Punto fijo

Binario	Decimal
0,000	0
0,001	0,125
0,010	0,25
0,011	0,375
0,100	0,5
0,101	0,625
0,110	0,75
0,111	0,875
1,000	1
1,001	1,125
1,010	1,25
1,011	1,375
1,100	1,5
1,101	1,625
1,110	1,75
1,111	1,875

1 bits parte entera
3 bits parte fraccionaria

Rango: 0 .. 1,875
Resolución: 0,125

Punto fijo

Binario	Decimal
,0000	0
,0001	0,0625
,0010	0,125
,0011	0,1875
,0100	0,25
,0101	0,3125
,0110	0,375
,0111	0,4375
,1000	0,5
,1001	0,5625
,1010	0,625
,1011	0,6875
,1100	0,75
,1101	0,8125
,1110	0,875
,1111	0,9375

0 bits parte entera
4 bits parte fraccionaria

Rango: 0 .. 0,9375
Resolución: 0,0625

Punto fijo

Si todos los bits se encuentran asignados a la parte fraccionaria, el sistema nunca llegará a representar el número 1 decimal, incluso con una gran cantidad de bits.

Con 6 bits → $,111111_2$ → **0,984375**
Con 7 bits → $,1111111_2$ → **0,991875**
Con 8 bits → $,11111111_2$ → **0,99609375**
Con 9 bits → $,111111111_2$ → **0,998046875**
Con 10 bits → $,1111111111_2$ → **0,999023438**
Con 11 bits → $,11111111111_2$ → **0,999511719**

Cada bit fraccionario adicional está sumando la mitad del bit inmediato más significativo. De esta manera, el sistema representará números entre 0 y "casi 1".

Punto fijo. Representación y errores

Al convertir un número decimal a sistema binario tendremos 2 casos:

- Sin restricción en la cantidad de bits a usar:

$3{,}0078125_{10} = 11{,}0000001_{2}$

Con restricción, por ejemplo 3 bits para parte entera y 4 bits para parte fraccionaria:

- $3{,}125_{10} = 011{,}0010_{2}$

En este caso, el número de bits disponible es suficiente para representar el número.

En una computadora, la capacidad de representación de un número estará dada por el tipo de dato que es utilizado para representar a dicho número.

Punto fijo. Representación y errores

Error absoluto y relativo

En ciertos casos, no es posible representar el número exacto, debiendo representar el valor más cercano posible.

Error Absoluto (EA): es la diferencia absoluta entre el valor representado y el valor que se deseaba representar

El EA máximo será siempre menor o igual a la mitad de la resolución del sistema: en el peor de los casos el número a representar se encuentra justo en la mitad de dos números representables consecutivos dentro del sistema utilizado.

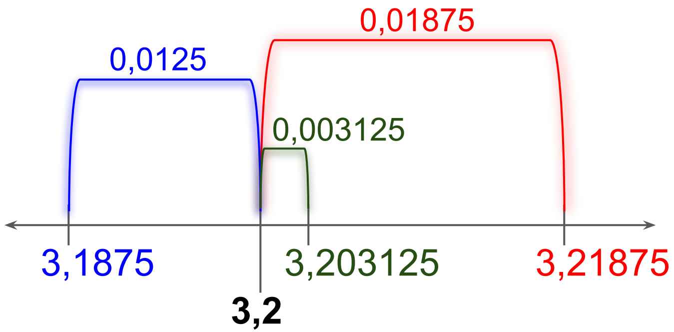
Error Relativo (ER): = EA / Número a representar

El ER es un coeficiente que denota la incidencia del error de representación sobre el número representado ¿cuán importante es el error?

Punto fijo. Representación y errores

Error absoluto (EA). Ejemplo: representar $\mathbf{3{,}2_{10}}$ con distintas restricciones en el número de bits.

- 3 bits para parte fraccionaria: $011{,}010_2 = 3{,}25_{10}$
  - Error = 3,2 – 3,25 = -0,05
- 4 bits para parte fraccionaria: $011{,}0011_2 = 3{,}1875_{10}$
  - Error = 3,2 – 3,1875 = 0,0125
- 5 bits para parte fraccionaria: $011{,}00111_2 = 3{,}21875_{10}$
  - Error = 3,2 – 3,21875 = - 0,01875
- 6 bits para parte fraccionaria: $011{,}001101_2 = 3{,}203125_{10}$
  - Error = 3,2 – 3,203125 = - 0,003125



El **EA** más pequeño es **0,003125** entonces **3,203125** es la representación más cercana a **3,2** dentro de este ejemplo. Con más bits asignados a la parte fraccionaria, sería incluso posible reducir más el error.

Punto fijo. Representación y errores

Error relativo (ER) = EA / Número a representar

Ejemplo:

- Se desea representar el 1,5 pero solo es posible representar el 1_{10}
    - EA = | 1 - 1,5 | = **0,5**
    - ER = 0,5 / 1,5 = **0,333**
    - El error representa un 33,3% con respecto del valor que se deseaba representar
- Se desea representar el $1.000.000,5_{10}$ pero solo es posible representar el $1.000.000_{10}$
    - EA = | 1.000.000 - 1.000.000,5 | = **0,5**
    - ER = 0,5 / 1.000.000,5 = **0,0000005**
    - El error representa un 0,00005% con respecto del valor que se deseaba representar

Para ambos casos el EA es el mismo, sin embargo, el ER es:

- Considerable en el primero
- Despreciable en el segundo (para la mayoría de los dominios tradicionales de la computación general).

Temas de la segunda parte

- Representación de datos
- Números enteros
- Operaciones aritméticas
- Punto fijo
- Punto flotante ←

Punto flotante

En punto fijo es posible representar un rango de valores positivos y negativos centrados en 0.

Sin embargo, punto fijo tiene limitaciones: números “muy grandes” y números “muy pequeños” en un mismo sistema:

- Tamaño de un átomo:
  - **0,00000000002 metros**
- 1 googol:
  - **10.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000**

CPU Intel i386 junto al co-procesador matemático de punto flotante i387 DX

Punto flotante

Notación científica es de utilidad en estos casos:

- Un número decimal “muy grande”: **976.000.000.000.000** se puede representar como: **$9{,}76 \times 10^{14}$**

- Un número decimal “muy pequeño”: **0,0000000000000976** se puede representar como: **$9{,}76 \times 10^{-14}$**

Punto flotante

Hemos desplazado en forma dinámica la coma decimal a una posición conveniente y usar el exponente de base 10 para definir la ubicación de la coma.

Esto permite tener un rango de números desde “muy pequeños” a “muy grandes” y pueden ser representados con pocos dígitos.

Este mismo enfoque puede ser aplicado para representar números binarios.

Punto flotante

Un número se puede representar de la forma: $\pm M \times B^{\pm E}$

Este número se puede almacenar en una palabra binaria con los dos campos **M** (mantisa) y **E** (exponente), los cuales pueden estar representados en cualquiera de los sistemas ya vistos (BSS, BCS, Ca1, Ca2, Ex2).

La mantisa **M** puede incluso ser fraccionaria.

La base **B** es implícita y no necesita almacenarse.

→ Se necesitan menos bits para almacenar **M** y **E** que el número completo. ←

Punto flotante

Un posible formato para almacenar un número en punto flotante:

Exponente (Ca2 8 bits)	Mantisa (BCS 24 bits)

La ubicación del exponente y de la mantisa, así como el número de bits asignados y los sistemas de representación utilizados son parte de la definición del sistema de punto flotante dado.

Punto flotante

Ejemplo:

- **Mantisa**
    - BSS
    - 4 bits
    - Entera
- **Exponente**
    - BSS
    - 4 bits
    - Entera

Punto flotante

Ejemplo:

- **Mantisa**
  - BSS
  - 4 bits
  - Entera
- **Exponente**
  - BSS
  - 4 bits
  - Entera

- Nro Máximo: $1111 \times 2^{1111} = 15 \times 2^{15}$
- Nro Mínimo: 0
- Rango: $[0 .. 15 \times 2^{15}] = [0 .. 491520]$
- Resolución extremo inferior: $(1 - 0) \times 2^{0} = 1$
- Resolución extremo superior: $(15 - 14) \times 2^{15} = 2^{15}$

Contrastemos con sistema punto fijo 8 bits BSS:

- Rango: [0 .. 255]
- Resolución extremo inferior: 1
- Resolución extremo superior: 1

Punto flotante

Ejemplo:

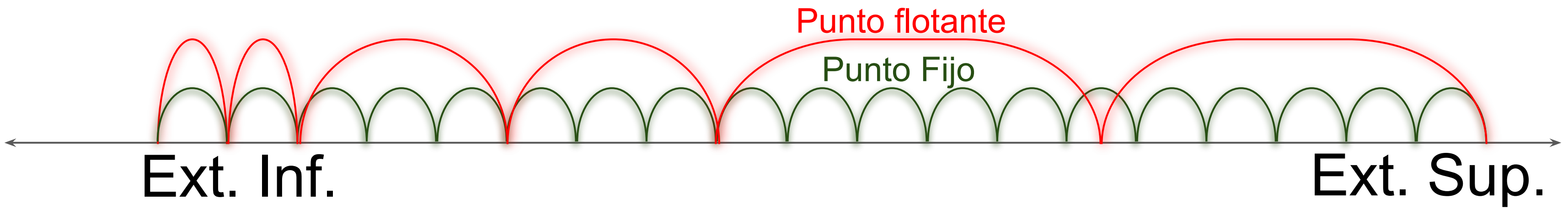
- **Mantisa**
  - BSS
  - 4 bits
  - Entera
- **Exponente**
  - BSS
  - 4 bits
  - Entera

Si comparamos ambos sistemas vemos:

- El rango en punto flotante es mayor
- La cantidad de combinaciones binarias distintas es la misma en ambos sistemas: $2^8 = 256$
- En punto flotante la resolución no es constante a lo largo del intervalo, como lo es en el ejemplo de punto fijo.
- En punto flotante "Ganamos" rango, "resignando" resolución con respecto a punto fijo.

Punto flotante

En punto flotante la posición de los valores no es uniforme a lo largo de la recta de números representables (como lo es en punto fijo), con lo cual existe una resolución distinta en cada uno de los extremos.



Punto flotante

Ejemplo:

- **Mantisa**
  - Ca2
  - 4 bits
  - Entera
- **Exponente**
  - Ca2
  - 4 bits
  - Entera

Punto flotante

Ejemplo:

- **Mantisa**
  - Ca2
  - 4 bits
  - Entera
- **Exponente**
  - Ca2
  - 4 bits
  - Entera

- Nro Máximo: $0111 \times 2^{0111} = 7 \times 2^7$
- Nro Mínimo: $1000 \times 2^{0111} = -8 \times 2^7$
- Rango: $[-8 \times 2^7 .. 7 \times 2^7] = [-1024 .. 896]$
- Resolución extremo inf (pos): $(1 \times 2^{-8} - 0) = 0{,}0390625$
- Resolución extremo sup (pos): $(7 - 6) \times 2^7 = 2^7 = 128$

Punto flotante

Ejemplo:

- **Mantisa**
  - BCS
  - 24 bits
  - Fracc.
- **Exponente**
  - Ca2
  - 8 bits
  - Entera

FORMATO DEL EJEMPLO	
Exponente	Mantisa Fraccionaria
SEEEEEEE S = bit de signo E = bits magnitud	½¼⅛ SMMMMMMMMMMMMMMMMMMMMMMM S = bit de signo M = bits de magnitud (fraccionarios) Primer bit M tiene una magnitud de ½ Segundo bit M tiene una magnitud de ¼ Tercer bit M tiene magnitud de ⅛

Punto flotante

Ejemplo:

- **Mantisa**
  - BCS
  - 24 bits
  - Fracc.
- **Exponente**
  - Ca2
  - 8 bits
  - Entera

El rango es simétrico dado que la mantisa es BCS.

- **Nro máximo positivo**: $011111111111111111111111 \times 2^{01111111}$
  - $= (2^{-1} + 2^{-2} + 2^{-3} + \ldots\ 2^{-21} + 2^{-22} + 2^{-23}) \times 2^{127}$
  - $= (1-2^{-23}) \times 2^{127} = 0{,}999999881 \times 1{,}701411835 \times 10^{38}$
  - = 170.141.163.300.000.000.000.000.000.000.000.000.000
- **Nro mínimo positivo**: $000000000000000000000001 \times 2^{10000000}$
  - $= 2^{-23} \times 2^{-128} = 2^{-151}$
  - = 0,00000000000000000000000000000000000000000350..
- **Nro máximo negativo**: $100000000000000000000001 \times 2^{10000000}$
  - $= -2^{-23} \times 2^{-128} = 2^{-151}$
  - = -0,00000000000000000000000000000000000000000350..
- **Nro mínimo negativo**: $111111111111111111111111 \times 2^{01111111}$
  - $= -(2^{-1} + 2^{-2} + 2^{-3} + \ldots\ 2^{-21} + 2^{-22} + 2^{-23}) \times 2^{127}$
  - $= -(1-2^{-23}) \times 2^{127} = -0{,}999999881 \times 1{,}701411835 \times 10^{38}$
  - = -170.141.163.300.000.000.000.000.000.000.000.000.000

Punto flotante. Normalización

Veamos el siguiente problema:

$40 \times 10^0 = 4 \times 10^1 = 0{,}4 \times 10^2 = 400 \times 10^{-1} = 40$ en todos los casos

Existen distintos valores de mantisa y exponente para representar un mismo número.

Lo mismo sucede en base 2.

Con el objetivo de tener un único par de valores de mantisa y exponente para un número, se introduce la normalización.

Punto flotante. Normalización

La normalización binaria es el proceso de mover el punto fraccionario de manera tal que el bit más significativo después del punto (sin considerar el bit de signo) sea 1. Esto es:

$$\begin{array}{cccccccccccc} & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & & & & & & & \\ S & 1 & M & M & M & M & M & \ldots & M & M & M \end{array}$$

Donde S es el bit de signo (si corresponde) y M son los restantes bits de la mantisa.

Esto implica además que la mantisa normalizada tendrá siempre una magnitud de al menos ½.

Punto flotante. Normalización

Ejemplo de normalización. Mantisa fraccionaria 4 bits BSS. Exponente BSS 3 bits.

$0010 \times 2^{010} = \frac{1}{8} \times 2^{2} = 0,5$ ← No está normalizada!

Debemos correr los bits de la mantisa 2 posiciones hacia la izquierda, pero además **debemos compensar con el exponente** para mantener el mismo valor del número (0,5). Para ésto, **se decrementa** en 2 unidades el valor del exponente.

$1000 \times 2^{000} = \frac{1}{2} \times 2^{0} = 0,5$ ← Sí está normalizada

Recordar:

- Al correr hacia la izquierda los bits de la mantisa, estoy multiplicando por 2
- Al decrementar en una unidad el valor del exponente, estoy dividiendo por 2

Punto flotante. Bit implícito.

Todo número en punto flotante normalizado comienza con 1 (representando ½).
¿es necesario almacenar el 1?

Si no lo almacenamos, ganamos un bit para representar números con mayor precisión:

$$\begin{array}{cccccccccccc} & & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & & & & & & & \\ S & \cancel{1} & M & M & M & M & M & \ldots & M & M & M & M \end{array}$$

Donde M es el nuevo bit que podemos adicionar a la magnitud de la mantisa.

Notar que el bit más significativo adiciona a la magnitud ¼ en lugar de ½

Al igual que las mantisas fraccionarias normalizadas, las mantisas fraccionarias normalizadas con bit implícito **siempre tienen una magnitud mínima de 0,5**.

Punto flotante. Bit implícito.

Mantisa fraccionaria **normalizada** BCS 4 bits, exponente BSS 2 bits. ¿nro máximo?

$$\tfrac{1}{2}\,\tfrac{1}{4}\,\tfrac{1}{8}$$

$$0\ \mathbf{1}\ 1\ 1 \times 2^{11} = 0\mathbf{1}11\text{x}2^{11} = (\mathbf{2^{-1}} + 2^{-2} + 2^{-3}) \times 2^{3} = 0{,}875 \times 2^{3} = 7$$

Mantisa fraccionaria **normalizada** con bit implícito BCS 4 bits, exponente BSS 2 bits. ¿nro máximo?

$$\tfrac{1}{2}\,\tfrac{1}{4}\,\tfrac{1}{8}\,{}^{1}/_{16}$$

$$0\ \cancel{1}\ 1\ 1\ 1 \times 2^{11} = 0111\text{x}2^{11}\ (2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}) \times 2^{3} = 0{,}9375 \times 2^{3} = 7{,}5$$

¿Cómo representar el 0 en este sistema?

NO SE PUEDE

Punto flotante. Errores de representación.

Punto flotante presentará errores de representación para ciertos números “difíciles” de representar en un espacio acotado de bits (por ejemplo, números recurrentes):

```
usuario@pc-usuario: ~
usuario@pc-usuario:~$ python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+2
3
>>> .1+.2
0.30000000000000004
>>>
```

$0{,}1_{10} = 0.0001100\overline{1100}\ldots$ $0{,}2_{10} = 0.0011001\overline{10011}\ldots$

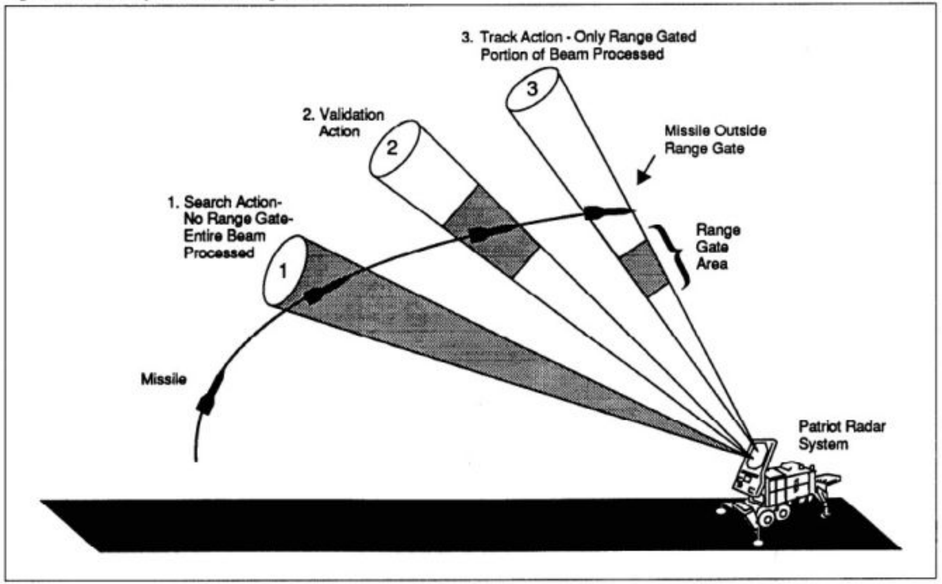
https://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/
http://effbot.org/pyfaq/why-are-floating-point-calculations-so-inaccurate.htm
https://stackoverflow.com/questions/588004/is-floating-point-math-broken

Punto flotante. Errores de representación. Ejemplo

1991 Patriot missile failure due to magnification of roundoff error

R-17 (SS-1C Scud-B)

Figure 5: Incorrectly Calculated Range Gate



Patriot Radar System

MIM-104 Patriot Missile

Punto flotante. Errores de representación. Ejemplo

1991 Patriot missile failure due to magnification of roundoff error

- Software desarrollado en **assembler 20 años antes**. **Modificado varias veces** para la guerra del Golfo Pérsico (operation *Desert Storm*) para poder interceptar misiles balísticos de alta velocidad.
- Registro de valores de tiempo transcurrido desde el instante de inicio del sistema **en décimas (0.1d) de segundo**, acotado a **24 bits**.
- Problema de representación:
  - `0.1d =                   0.000110011001100110011001100`**`1100`**`... b`
  - `Truncado a 24 bits =  0.000110011001100110011001100b`
  - `Error =                  0.000000000000000000000000000`**`1100`**`... b`
- Introducción de **subrutina** para convertir mediciones de tiempo en punto flotante **con mayor precisión**
- La invocación a esta subrutina era **necesaria en varias partes del código**, pero dichas invocaciones no se realizaron en todas las ubicaciones donde se requería.
- Luego **100 horas de *up-time*** del sistema Patriot: **0.3433** segundos de error (tiempo en el que un Scud atraviesa 500 metros aproximadamente).
- Saldo: 28 soldados fallecidos

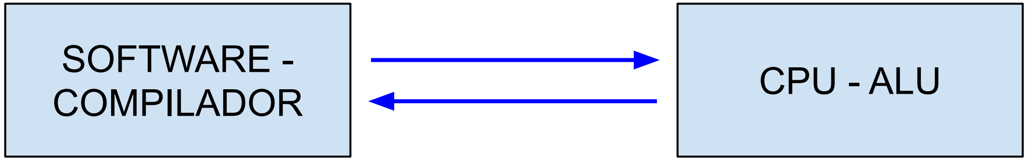
http://www-users.math.umn.edu/~arnold/disasters/Patriot-dharan-skeel-siam.pdf

IEEE 754

Standard establecido por la IEEE (Institute of Electrical and Electronics Engineers) en el año 1985 para cómputo en punto flotante.

Ampliamente utilizado en varios dominios de la computación (aunque no en todos, dadas las limitaciones de precisión en punto flotante).

Tanto software como hardware “hablan el mismo idioma” en lo que refiere a representación de números en punto flotante.



IEEE 754

Ejemplo en Java:

float **a** = 3.14; → 32-bit IEEE-754 (simple precision)
double **b** = 6.022e23; → 64-bit IEEE-754 (double precision)

Las variables **a** y **b** serán representadas por una secuencia de bits según el standard.

Esta secuencia de bits es “entendida” entonces tanto por el software como por el hardware.

IEEE 754

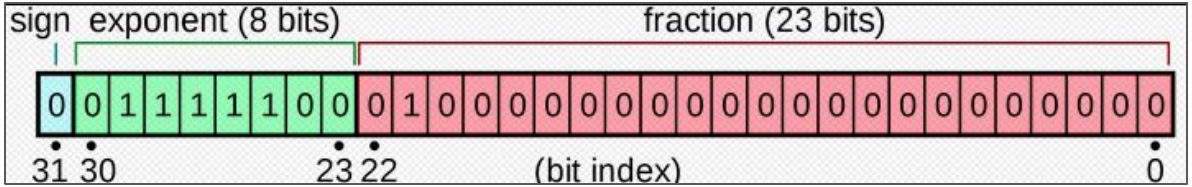
El standard define:

- Formatos numéricos
  - De qué manera estructurar la secuencia de bits a fin de representar números
- Operaciones
  - De tipo aritméticas, trigonométricas, etc.
- Reglas de redondeo
  - A fin de aproximar al valor más cercano representable mediante redondeos o truncamientos.
- Manejo de excepciones
  - Ante la presencia de situaciones inesperadas como división por cero

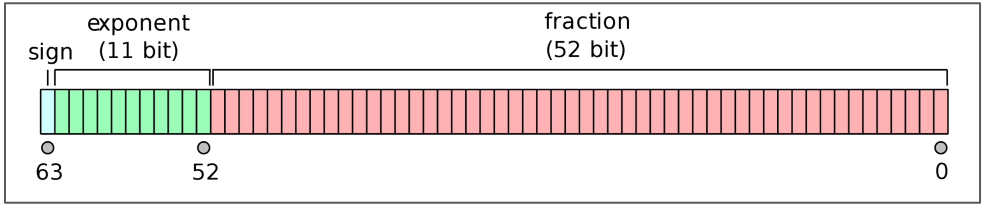
IEEE 754

Entre otros, define los siguientes formatos:

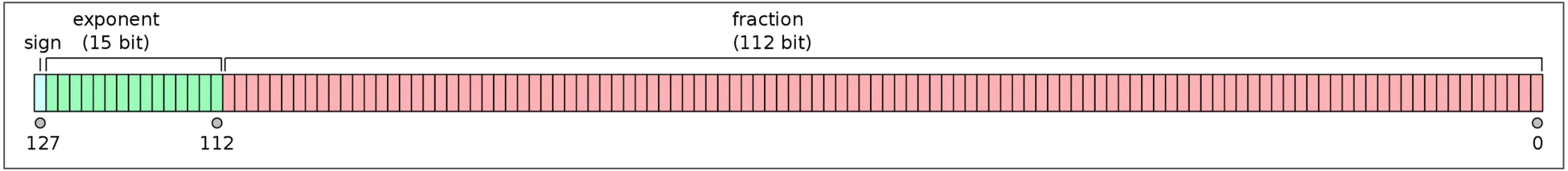
Simple precisión (32 bits)



Doble precisión (64 bits)



Cuádruple precisión (128 bits)



IEEE 754

Formato numérico (introducción):

- Mantisa:
  - BCS
  - Fraccionaria normalizada con la coma después del primer bit que es siempre uno (1,)
- Exponente:
  - Representado en exceso sesgado: $2^{n-1}-1$
- Casos especiales
  - $E = 111..111_2$ $M <> 0$ → NaN
  - $E = 111..111_2$ $M = 0$ → Infinito
  - $E = 0$ $M = 0$ → Cero
  - $E = 0$ $M <> 0$ → Denormalizado (0,)

IEEE 754

Máximo valor representable simple precisión:

$$2^{128} \approx 3{,}40\times10^{38} \approx$$

34.000.000.000.000.000.000.000.000.000.000.000.000

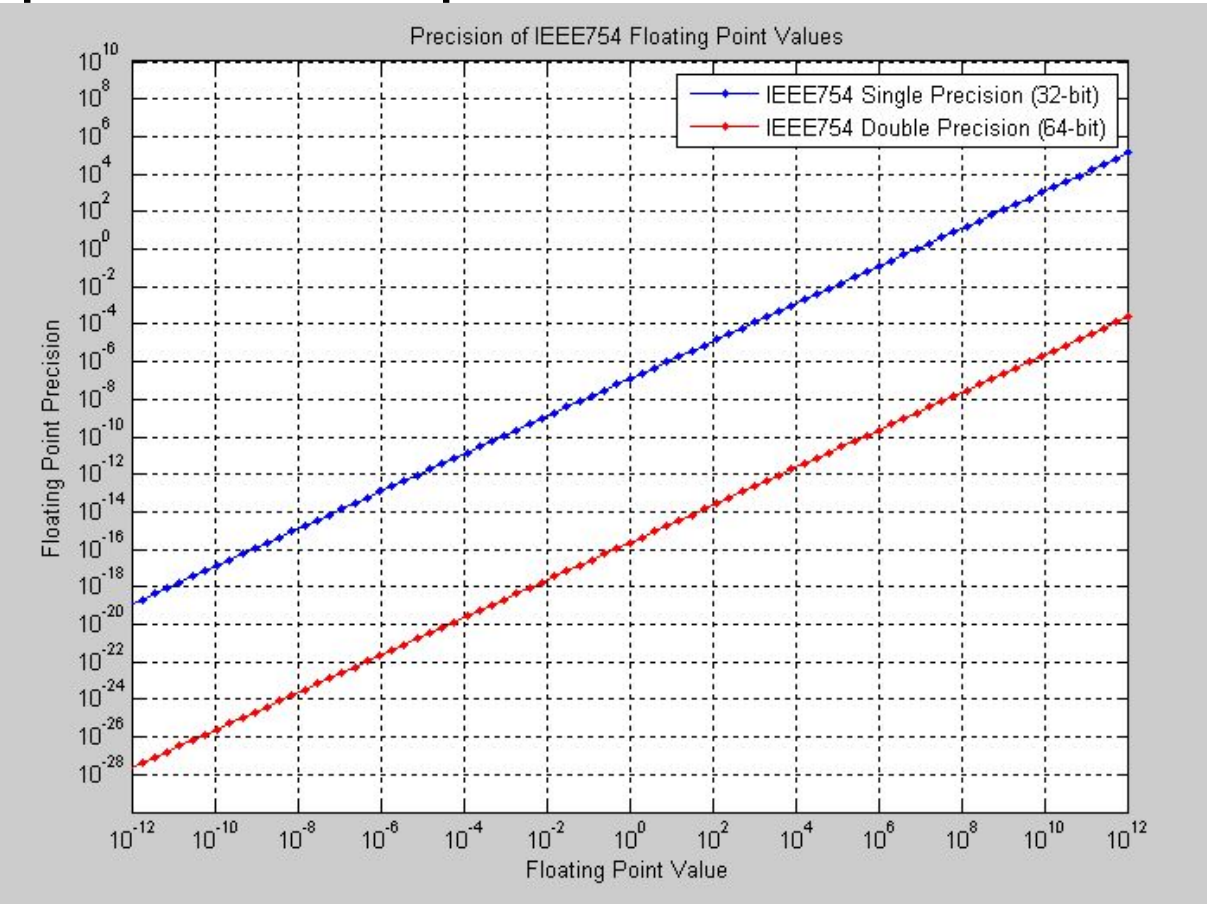
Máximo valor representable doble precisión:

$$2^{1024} \approx 1{,}79\times10^{308} \approx$$

17.900.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000

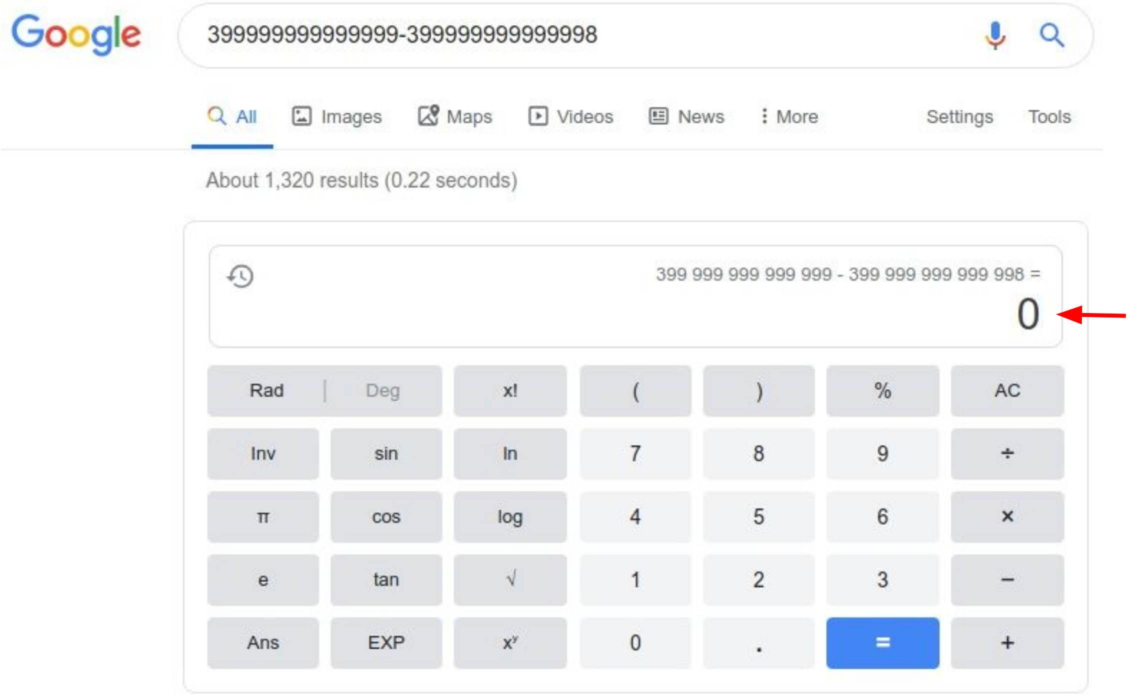
IEEE 754. Simple vs doble precisión.

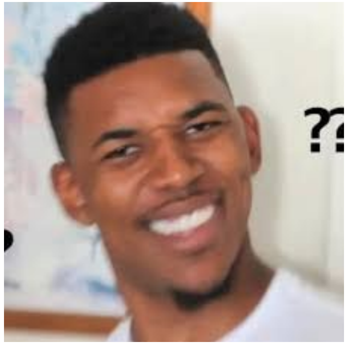
El trade-off con
la precisión persiste



IEEE 754. Simple vs doble precisión. Ejemplo.

http://www.google.com/search?&q=399999999999999-399999999999998





https://blog.codinghorror.com/why-do-computers-suck-at-math/

IEEE 754. Simple vs doble precisión. Ejemplo.

http://www.google.com/search?&q=3999999999999999-3999999999999998

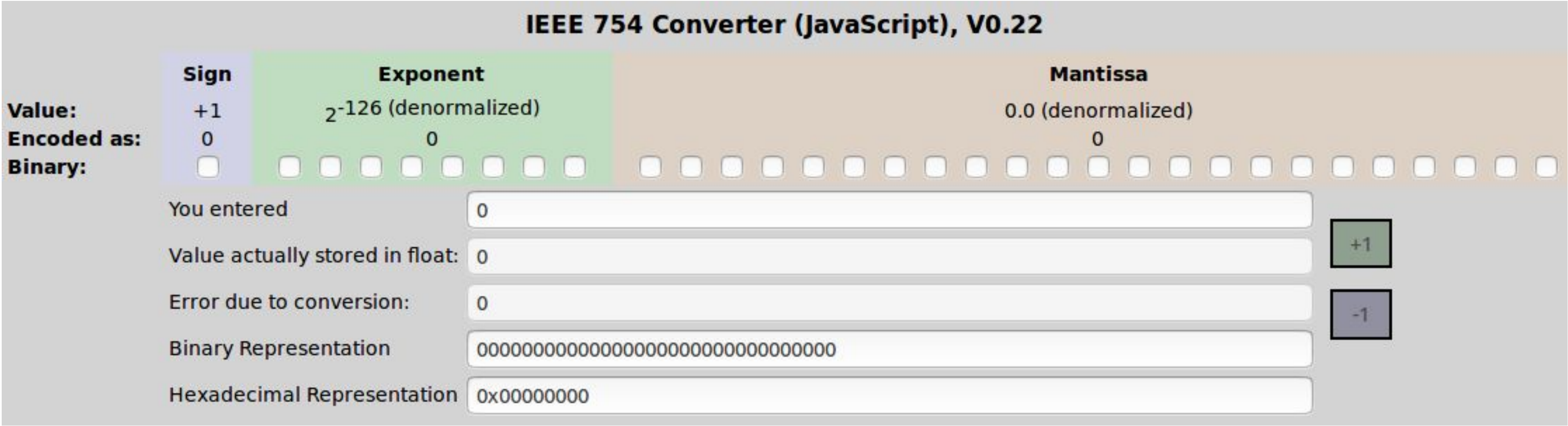
```
int main() {
    char* a = "3999999999999999";
    char* b = "3999999999999998";
    float da = atof(a);
    float db = atof(b);
    printf("%s - %s = %f\n", a, b, da-db);
}
```

3999999999999999 - 3999999999999998 = 0.000000

Google pareciera estar utilizando **simple precisión** (32 bits) en lugar de **doble precisión** (64 bits)
Si cambiamos **float** por **double** la precisión se incrementa y el resultado es el esperado.

https://www.lnds.net/blog/lnds/2008/8/25/por-que-falla-la-calculadora-de-google

IEEE 754. Aplicación interactiva.



https://www.h-schmidt.net/FloatConverter/IEEE754.html