Apunte de Sistemas de Numeración

Introducción

Las computadoras pueden almacenar datos de diferente índole, como ser el saldo de una cuenta corriente, el nombre de una persona o la temperatura de una determinada región. Para ello debe existir una forma de representar el valor $128,5 otra de representar JORGE y otra también de representar el valor -10°. Estos temas son los que se tratarán a continuación.

Sistemas de Numeración no posicionales

Los sistemas de numeración no posicionales son aquellos en los cuales el valor de un símbolo es independiente de la posición que ocupa. Un claro ejemplo de esto es el sistema romano. Analicemos un ejemplo:

221 en números romanos es CCXXI

En el 221 decimal el primer 2 tiene un valor de 200 y el segundo de 20, entonces decimos que el símbolo 2 adquiere un valor de acuerdo a la posición que esté ocupando; en cambio en CCXXI ambas C valen 100 y ambas X valen 10 por lo cual el valor de C y X es igual para cualquier posición.

Sistemas de Numeración posicionales

En los sistemas de numeración posicionales el valor de un símbolo depende del lugar que este ocupe dentro del número. El más conocido es el sistema decimal.

Generalizando, en un sistema de numeración posicional de base b en donde la base siempre es igual a la cantidad de símbolos que posee el sistema, en el **decimal** es igual a 10 dado que tenemos un total de 10 dígitos diferentes (0,1,2,3,4,5,6,7,8,9) la representación de un número se define a partir del *Teorema Fundamental de la Numeración*:

$$(\ldots ABC,DEF\ldots) = Ab^2 + Bb^1 + Cb^0 + Db^{-1} + Eb^{-2} + Fb^{-3}$$

$$\text{Ej. } (423,1)_6 = (4x6^2 + 2x6^1 + 3x6^0 + 1x6^{-1})_{10}$$

Las generalizaciones más simples del sistema decimal se obtienen cuando b es un entero no negativo mayor a 1 y cuando los {A,B,C,etc} pertenecen al conjunto de enteros en el rango [0,b). Así, cuando b es 2 se obtiene el sistema de numeración **binario**, cuando b es 8 el **octal** y cuando b es 16 el **hexadecimal**. Pero en general, se podría elegir cualquier b distinto de cero, y los {A,B,C,etc} de cualquier conjunto de números, obteniendo sistemas muy interesantes. (Ej. Base ternaria balanceada: formada por los símbolos –1, 0, 1)

Siguiendo con el ejemplo anterior, el punto (o la coma dependiendo del símbolo usado) que se encuentra entre C y D se llama **punto fraccionario.** Cuando b es 10 se lo llama punto decimal y cuando b es 2, punto binario.

Los {A,B,C…} se llaman **dígitos** de la representación. Se dice que un dígito A es más significativo que B si A está ubicado a la izquierda de B en el número. Así, el dígito del extremo izquierdo es denominado comúnmente el **dígito más significativo** y el del extremo derecho el **menos significativo.** En el número del ejemplo el 4 es el dígito más significativo y el 1 el menos significante; también es cierto que 2 es un dígito más significativo que 3.

- Dígitos del Sistema Binario: Son el 0 y el 1 y generalmente se denominan Bits.
- Dígitos del Sistema Octal: Son 0, 1, 2, 3, 4, 5, 6 y 7.
- Dígitos del Sistema Hexadecimal: Son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

Aritmética de Base b

Las operaciones entre números de base b se lleva a cabo conforme las tablas de adición y multiplicación correspondientes a dicha base.

Ejemplos:

Sistema Decimal

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

*	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

789	789	789
+201	x 2	-201
990	1578	588

Sistema de Base 4

+	0	1	2	3
0	0	1	2	3
1	1	2	3	10
2	2	3	10	11
3	3	10	11	12

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	10	12
3	0	3	12	21

123_4 + 201_4 = 330_4

123_4 x 201_4 = $123_4 + 31200_4 = 31323_4$

Cambio de Base

Suponer que se quiere convertir un número de base **b** a **p**. La mayoría de los algoritmos existentes se basan en multiplicaciones y divisiones. Hay tres casos elementales.

Números Enteros

A. $N_b \rightarrow (\ )_{10}$

$$ABCD_b = (A\mathbf{b}^3 + B\mathbf{b}^2 + C\mathbf{b}^1 + D\mathbf{b}^0)_{10}$$

Ejemplos:

$34_6 = (3x6^1 + 4x6^0)_{10} = (22)_{10}$

$A7_{16} = (10x16^1 + 7x16^0)_{10} = (167)_{10}$

B. $N_{10} \rightarrow (\ )_b$

Consiste en realizar divisiones sucesivas hasta que el cociente sea menor que el divisor (base deseada), luego juntar el último cociente obtenido junto con todos los restos de abajo hacia arriba.

Ejemplos:

$34_{10} \rightarrow (\ )_2$

34 |2
0 17 |2
1 8 |2
0 4 |2
0 2 |2
0 **1**

Hemos llegado al final de las divisiones sucesivas notar que 1 < 2 →
El resultado final es **$(100010)_2$**

$34_{10} \rightarrow (\ )_{16}$ 34 |16
2 **2**

Hemos llegado al final de las divisiones sucesivas notar que 2 < 16 → El resultado final es **$(22)_{16}$**

c. $N_b \rightarrow (\ )_p$

Para resolver este cambio de base se utiliza la combinación de los dos métodos anteriormente explicados.

1) $N_b \rightarrow (\ )_{10}$
2) $(\ )_{10} \rightarrow (\ )_p$

Ejemplo $32_4 \rightarrow (\ )_6$

1) $N_b \rightarrow (\ )_{10}$
$32_4 \rightarrow (\ )_{10}$ 324 = (3x4 + 2)10 = 1410

2) $(\ )_{10} \rightarrow (\ )_p$
$14_{10} \rightarrow (\ )_6$ 14 |6
2 **2** → 2 < 6 por lo tanto, resultado final **$(22)_6$**

Es decir que el 34 en base 4 corresponde al 22 en base 6, en otras palabras:
$32_4 = 22_6$

Partes Fraccionarias

A. $0,N_b \rightarrow (\ )_{10}$

$$0,ABC_b = (Ab^{-1} + Bb^{-2} + Cb^{-3})_{10}$$

Ejemplo:

$$0,132_4 = (1x4^{-1} + 3x4\text{-}2 + 2x4\text{-}3)10 = 0,4687510$$

B. $0,N_{10} \rightarrow (\ )_b$

Se resuelve mediante multiplicaciones sucesivas tomando de cada resultado la parte entera. Se termina cuando la parte fraccionaria es igual a cero. En caso de no llegar a este resultado cuantos más decimales se toman mayor precisión se alcanza.

Ejemplo:

$0,125_{10} \rightarrow (\ )_2$ 0,125 x 2 = **0**,250 **0**
0,250 x 2 = **0**,500 **0**
0,500 x 2 = **1**,000 **1** **$0,125_{10} = 0,001_2$**

c. $0,N_b \rightarrow (\ )_p$

A igual que lo visto para números enteros primero se realiza el cambio $0,N_b \rightarrow (\ )_{10}$ y finalmente se pasa de $(\ )_{10} \rightarrow (\ )_p$.

Ejemplo $0.001_2 \rightarrow (\ )_4$

1) $N_b \rightarrow (\ )_{10}$
   $0.0012 \rightarrow (\ )10$ $\quad 0,0012 = (1x2\text{-}3)10 = 0,12510$

2) $(\ )_{10} \rightarrow (\ )_p$
   $0,125_{10} \rightarrow (\ )_4$ $\quad 0,125 \times 4 = 0,5$ **0**
   $0,500 \times 4 = 2.0$ **2**

Es decir que el 0.001 en base 2 corresponde al 0.02 en base 4, en otras $0,001_2 = 0,02_4$

Recordar:
Si se tiene un número que posee parte entera y fraccionaria se realiza el cambio para cada parte por separado y luego se suman los resultados obtenidos.

Ejemplo:

$354,102_{16} \rightarrow (\ )_{10}$
$354,102_{16} = 354_{16} \rightarrow (\ )_{10} + 0,102_{16} \rightarrow (\ )_{10}$
$354,102_{16} = 3x16^2 + 5x16^1 + 4x16^0 + 1x16^{-1} + 2x16^{-3}$

<u>Casos Especiales de Cambio de Base</u>

a) $b^x = p$

Se irán formando grupos de x dígitos y se hará el cambio para cada uno de estos grupos en forma independiente. Para la parte entera se empiezan a formar los grupos de derecha a izquierda en caso de ser necesario se completa con ceros a izquierda. Para la parte fraccionaria se comienza a formar los grupos de izquierda a derecha y de ser necesario se completa con ceros a derecha.

Ejemplo:

$10110_2 \rightarrow (\ )_8$ $\quad 2^3 = 8$ **tomo de 3 dígitos**
$(010 \mid 110)_2 = 26_8$
$110_2 = 6_8$
$010_2 = 2_8$

b) $b^{1/x} = p$

Cada dígito en la base b se expandirá en x dígitos en la base p.

Ejemplo:

$AE75_{16} \rightarrow (\ )_2$ $\quad 16^{1/4} = 2$ **se expande en 4 dígitos binarios**

$5_{16} = 0101_2$
$7_{16} = 0111_2$
$E_{16} = 1110_2$
$A_{16} = 1010_2$ $\quad$ **$(1010111001110101)_2$**

Números de Precisión Finita

Al hacer operaciones aritméticas, en muy raras ocasiones uno se preocupa por la cantidad de dígitos decimales que son necesarios para representar a un número. Se puede calcular que hay 10^{78} electrones en el universo sin molestarse por el hecho de que se requieren 79 lugares decimales para escribir el número completo. Una persona que evalúa una función a mano buscando una solución de 6 dígitos significativos, simplemente trabaja con resultados intermedios de 7,8 o cuántos dígitos necesite.

Con las computadoras las cosas son bastantes diferentes. En la mayoría la cantidad de memoria disponible para guardar números se fija en el momento de su diseño. La cantidad de dígitos disponibles para representar un número siempre será fija. Llamaremos a estos números: **números de precisión finita.**

Desafortunadamente el conjunto de los números de precisión finita no es cerrado con respecto a las operaciones aritméticas básicas. Analicemos un ejemplo con los números enteros positivos de tres dígitos:

600 + 600 = 1200 (muy grande)
003 – 005 = -002 (negativo)
050 x 050 = 2500 (muy grande)
007 / 002 = 3,5 (no es un entero)

Las exclusiones se pueden dividir en dos grupos: las operaciones cuyo resultado es mayor al máximo número del conjunto o menor al mínimo número del conjunto (**error de overflow**), y las operaciones cuyos resultados simplemente no pertenecen al conjunto.

Representación de números negativos

En general los números negativos en cualquier base b se representan del modo habitual, precediéndolos con un signo "−". En cambio, en una computadora (sistema binario) existen 4 métodos posibles que son:

- Signo y magnitud
- Complemento a uno
- Complemento a la base
- Exceso a la Base

Signo y magnitud

Para representar un número signado de n-bits usando el método de "signo y magnitud" consiste en:

1) un bit para representar el signo. Ese bit a menudo es el bit más significativo y, por convención se usa un 0 para números positivos ,y 1 para los negativos;
2) los n-1 bits restantes para representar el significando que es la magnitud del número en valor absoluto.

Ejemplo:

Usando 8 bits → 1 bit para el signo y 7 bits para la magnitud; queremos representar el número -97_{10}. Procedemos a:

a) Mirar el signo del número -97_{10}, apreciamos que es negativo, llevará como bit de signo un 1;
b) Realizar la conversión: el valor absoluto de -97_{10} es $|-97_{10}| = 97_{10}$. Que en binario es: 1100001_2;
c) Colocar todo junto → el número -97_{10} queda representado de la siguiente manera usando **Signo y Magnitud**: 11100001_2. Donde el 1 en el bit más significativo indica un número negativo, y 1100001_2 es el significando en valor absoluto.

Para el caso inverso, dado un número binario en **Signo y Magnitud** de 8 bits, por ejemplo, 10110101_2, procedemos a:

1) Analizar el bit más significativo, que siendo un 1 indica que el número es negativo;
2) Convertir el significado a la base deseada, por ejemplo, en decimal, tomando en cuenta que el valor obtenido está en valor absoluto y la magnitud real estará dada por el bit de signo obtenido antes → $0110101_2 = |53_{10}|$.
3) Siendo que el bit de signo es 1, el número real es -53_{10}. Si el bit de signo fuese 0, el número hubiese sido $+53_{10}$.

Ventajas y desventajas

Posee un rango simétrico es decir para *n*-bits, el rango en decimal es $(-2^{n-1}-1; 2^{n-1}-1)$; usando n=8, los números van del $+127_{10} = \mathbf{0}1111111_2$, pasando por el $+0_{10} = \mathbf{0}0000000_2$ y el $-0_{10} = \mathbf{1}0000000_2$, hasta el $-127_{10} = \mathbf{1}1111111_2$

Con esta ventaja podemos apreciar fácilmente las desventajas que se tienen:

- No permite operar aritméticamente ya que se obtienen resultados incorrectos. Ej. $00010101_2 + 11100001_2 = 11110110_2$ $(-118_{10}) \neq -76_{10}$ → $+21_{10} + -97_{10} = -76_{10}$
- Posee doble representación del cero. Al representar en Signo y Magnitud, aparece el cero signado: $\mathbf{0}0000000_2$ $(+0_{10})$ y $\mathbf{1}0000000_2$ (-0_{10}).

Complemento a uno

Este método es otra alternativa para representar números negativos y consiste en aplicarle un **NOT bit a bit** al número, es decir en otras palabras, la inversión de unos por ceros y ceros por unos. De esta forma, la representación por Complemento a Uno de un número signado de *n*-bits es:

1) un bit para representar el signo. Ese bit a menudo es el bit más significativo y, por convención un 0 es para los positivos, y un 1 para negativos;
2) los n-1 bits restantes para representar el significando que es la magnitud del número en valor absoluto para el caso de números positivos, o bien, es el complemento a uno del valor absoluto del número, en caso de ser negativo.

Ejemplo:
Con 8 bits, tenemos 1 bit para el signo y 7 bits para la magnitud. Entonces para representar el número -97_{10} procedemos a:

a) Tomar nota del signo del número -97_{10} que, siendo negativo, llevará como bit de signo un 1;
b) Como el signo es negativo, el número a continuación del bit de signo, deberá expresarse en complemento a uno. Al realizar la conversión: el valor absoluto de -97_{10} es $|-97_{10}| = 97_{10}$. Que en binario es: 1100001_2, y el complemento a uno de 1100001_2 es **C1**$(1100001) = 0011110_2$;
c) Colocar todo junto, por lo tanto, el número -97_{10} en binario con el método de **Complemento a Uno** es: 10011110_2. Donde el 1 en el bit más significativo indica un número negativo, y 0011110_2 es el significando en complemento a uno del valor absoluto del número.

Para el caso inverso, dado un número binario en **Complemento a uno** de 8 bits, por ejemplo, 10110101_2, realizamos lo siguiente:

1) Analizar el bit más significativo, que siendo un 1 indica que el número es negativo;
2) Convertir el significado a la base deseada, por ejemplo, en decimal, tomando en cuenta que: el valor obtenido está en valor absoluto y que la magnitud real estará dada por el bit de signo obtenido antes, en caso de que el bit del signo sea negativo (como es el caso) se deberá obtener el complemento a uno; aplicarle el NOT al significado → **C1**$(0110101) = 1001010_2 = |74_{10}|$.
3) Siendo que el bit de signo es 1, el número real es -74_{10}. Si el bit de signo fuese 0, el número hubiese sido $0110101_2 = +53_{10}$ (no se complementaria).

Ventajas y desventajas
Las desventajas son:

- Posee doble representación del cero → $00000000_2 = +0_{10}$
  $11111111_2 = -0_{10}$

Mientras que las ventajas de la representación en Complemento a uno son:

- Posee un rango simétrico tomando como ejemplo n = 8 como la cantidad de bits: los números van del $+127_{10} = \mathbf{0}1111111_2$, pasando por el $+0_{10} = \mathbf{0}0000000_2$ y el $-0_{10} = \mathbf{1}1111111_2$, hasta el $-127_{10} = \mathbf{1}0000000_2$. Generalizando, para *n*-bits, el rango (en decimal) para Complemento a uno es $(-2^{n-1}-1; 2^{n-1}-1)$
- Permite operar aritméticamente y para obtener el resultado correcto al operar se debe sumar el acarreo obtenido al final de la suma/resta realizadas en caso de haberlo obtenido, este acarreo se lo conoce con el nombre de **end-around carry**. Por ejemplo:

a) $00010101_2 + 10011110_2 = 10110011_2$ ($+21_{10} + -97_{10} = -76_{10}$) end-around carry = 0;
b) $00000010_2 + 11111110_2 = \mathbf{1}00000000_2$ ($+2_{10} + -1_{10} = 0_{10} \neq +1_{10}$), que corregimos mediante $00000010_2 + 11111110_2 = 00000000_2 + \mathbf{1}_2 = 00000001_2$, que es el resultado correcto, se puede apreciar el end-around-carry = **1**

Complemento a la Base

El complemento de un número dado en una base es aquel que sumado al número original da la base a la n, siendo n la cantidad de dígitos que componen a ese número.

Formalmente, el complemento a *b* de un número *r*, representado en *n* dígitos en base *b* se define como:

$$C_b(r_b) = (10_n \ldots 0_1) - r_b$$

Nótese que el número que se utiliza como minuendo tiene un dígito más que los usados en la representación, es decir n+1 dígitos, un 1 seguido de n ceros a la derecha. Usando el complemento a 10 de 13579_{10} → $C_{10}(13579_{10}) = 100000 - 13579 = 86421_{10}$

Ejemplo:

El concepto de complemento puede ser usado para transformar una operación de resta de dos números en la suma de uno de ellos más el complemento del otro. Veamos a continuación como se aplicaría esto:

A – B = d
A – **B + b^n** = d + b^n → donde **b^n - B = B complemento**
A + **Bcomp** = d + b^n

a) $A = 10376_{10}$ y $B = 234_{10}$ con n = 5
A - B = d
A + **Bcomp** = d + b^n → **Bcomp** = $100000_{10} - 234_{10} = 99766_{10}$
10376_{10} + **99766_{10}** = d + 100000_{10}
~~1~~10142_{10} = d + ~~100000_{10}~~
10142_{10} = d

b) $A = 101_2$ y $B = 1000_2$ con n = 4
B – A = d
B + **Acomp** = d + b^n → **Acomp** = $10000_2 - 101_2 = 1011_2$
1000_2 + **1011_2** = d + 10000_2
~~1~~0011_2 = d + ~~10000_2~~
11_2 = d

Complemento a 2

Permite representar números negativos en el sistema binario y la realización de restas mediante sumas. Tener en cuenta que estos números negativos están

representados a través de su complemento. Es decir un número más su negativo (inverso aditivo), nos da un único cero → a + (-a) = 0

El complemento a 2 de un número X_2 se obtiene de sumar 1 al número negado bit a bit de X_2, en otras palabras → **Not (X) + 1**
Ejemplo: $C_2(01101_2) = 10010 + 00001 = 10011_2$

Complemento a 2 con 4 bits.

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

Supongamos que queremos representar el número -97_{10} usando 8 bits → n = 8. Procedemos a:

a) Tomar nota del signo del número -97_{10} que, siendo negativo, llevará como bit de signo un 1;
b) Como el signo es negativo, el número deberá expresarse en complemento a dos. Aplicamos el Not al valor absoluto y le sumamos 1, en otras palabras: $Not(97_{10}) + 1$ → Que en binario es: $Not(01100001_2) + 1 = 10011110_2 + 1 = 10011111_2$ (complemento a dos);

Para el caso inverso, dado un número binario en **Complemento a 2**, por ejemplo, 10110101_2, y teniendo en cuenta que n = 8, lo que debemos hacer es:

1) Analizar el bit más significativo, que siendo un 1 indica que el número es negativo;
2) Convertir el significando a la base deseada, por ejemplo, en decimal, tomando en cuenta que: el valor obtenido está en valor absoluto, que la magnitud real estará dada por el bit de signo obtenido antes, y que en caso de ser bit de signo negativo (como es el caso) se deberá obtener el complemento a dos: C2 (10110101) = Not (10110101) + 1 = $01001010_2 + 1$ → $01001011_2 = |75_{10}|$.
3) Siendo que el bit de signo es 1, el número real es -75_{10}. Si el bit de signo fuese 0, el número hubiese sido $00110101_2 = +53_{10}$ (sin complementar a dos).

Ventajas y desventajas
Usando n = 8 bits como ejemplo podemos decir que:
Desventajas:

- Posee un rango asimétrico de números → los números van del $+127_{10}$ (01111111_2) pasando por el $+0_{10}$ (00000000_2). Pero el 11111111_2, ya no es -0_{10} como en la representación anterior, sino que es -1_{10} y al llegar al 10000000_2 nos encontramos con que el

complemento a 2 de 10000000_2 es el mismo valor 10000000_2. Por lo tanto, por convención, se asigna a este número particular el valor de -128_{10} (para 8 bits). Generalizando el rango decimal para n-bits, usando el **Complemento a 2** es $(-2^{n-1}; 2^{n-1}-1)$.

Ventajas:

- No posee doble representación del cero.
- Permite operar aritméticamente.

Exceso a Base

Otro método para representar un número signado de *n*-bits consiste en:

1) Tomar el valor real del número a representar;
2) Sumarle la base elevada según la cantidad de dígitos menos 1 que se tienen disponibles;

Esto se lo conoce como representación en **Exceso a base (b^{n-1})**, puesto que a cada número se le suma el mismo valor y está en exceso por dicho valor.

El formato en exceso es habitual para la representación del exponente en números en punto flotante, por ejemplo, para la norma IEEE-754, con una salvedad ya que el exceso es calculado de la siguiente manera: $x + 2^{n-1}-1$.

Ejemplo de Exceso 2^{n-1}

Siendo n = 8 la cantidad de bits disponibles, los números serán representados en Exceso de $2^{8-1} = 128_{10}$. Cabe destacar que, con 8 bits, podemos representar, $2^8 = 256$ números.

Ahora, supongamos que tenemos que representar el número -97_{10} (decimal), la manera de representarlo usando Exceso 2^{n-1} es la siguiente:

a) Tomar el número -97_{10} y sumarle el exceso, en este caso 128_{10} →
   $-97_{10} + 128_{10} = 31_{10}$;
b) Convertimos a binario → $31_{10} = 00011111_2$.

Para el caso inverso, dado un número binario en **Exceso 128_{10}**, por ejemplo, 10110101_2, procedemos a:

1) Convertir el número a la base deseada → $10110101_2 = 181_{10}$;
2) Pero el valor obtenido está en exceso a 128, entonces debemos quitarle dicho exceso, restando 128 → $181_{10} - 128_{10} = 53_{10}$.

Ventajas y desventajas de Exceso 2^{n-1}

Siguiendo con n = 8, las desventajas son:

- Requiere de operaciones aritméticas intermedias para su obtención, y de cambiar el número de bits se deben actualizar dichas operaciones intermedias para reflejar el nuevo exceso.
- Posee un rango asimétrico que va desde $+127_{10} = 11111111_2$ hasta el $-128_{10} = 00000000_2$. Generalizando el rango en decimal para exceso es $(-2^{n-1}; 2^{n-1}-1)$.

Y las ventajas son:

- No hay empaquetación del número. Esto significa que no hay que recordar que partes del número son signo y valor, sino que el número esta formado por los n-bits.
- Permite operar aritméticamente, teniendo en cuenta que cada operación lleva asociado su exceso y debemos restarlo al resultado final para obtener la correcta representación. Por ejemplo:
  - 00011111_2 + 10110101_2 = 11010100_2
  - -97_{10} + 53_{10} = $84_{10} \neq -44_{10}$

  Al resultado obtenido debemos quitarle dicho exceso:

  → $84_{10} - 128_{10} = -44_{10}$ (1010100_2)

Formato y Configuración

Es importante tener en claro los conceptos de Formato y Configuración.
Cuando hablamos de Formato nos referimos a la representación computacional de un número.

Ejemplos:

- Binario de Punto Fijo sin Signo
- Binario de Punto Fijo con Signo
- Empaquetado
- Zoneado
- Binario de Punto Flotante

En cambio, la configuración es la representación en una determina base de un número en un formato.
Ejemplo:
$15_{10} = 1111_2$ → Usando 16 bits, 0000000000001111_2 es la configuración binaria de un BPF sin signo

Expansión y truncamiento

Cuando hablamos de **Expandir formato**, nos referimos a completar su representación computacional sin alterar el número representado en el mismo.

Mientras que, cuando hablamos de **Truncar formato**, nos referimos a descartar dígitos de su representación sin alterar el número representado en el mismo.

Formato Binario de punto fijo sin signo

- Base: 2
- Representa: Números enteros positivos.
- Máximo: $(2^n - 1)_{10}$

¿Cómo almacenar un número en el formato?

1) Pasar el número a base 2.
2) Completar con ceros a izquierda la capacidad (cantidad de bits) del formato.

Ejemplo: Capacidad: 8 bits Nro.: 27_{10}
1) $27_{10} = 11011_2$
2) 00001101_2

¿Cómo recuperamos un número en el formato?

Se realizan en orden inverso los pasos para almacenar.

Expansión y truncamiento

- Expandir formato: se completa con ceros a izquierda.
  00011011_2 8 bits → 0000000000011011_2 16 bits
- Truncar formato: Si es posible se sacan ceros a izquierda.
  00001101_2 8 bits → 1101_2 4 bits
  00010111_2 8 bits → No es posible truncarlo a 4 bits

Formato Binario de Punto Fijo con Signo

- Base: 2
- Representa: Enteros positivos y negativos.
- El primer bit se reserva para el signo: 0 (+) y 1 (-).
- Máximo: $2^{n-1} – 1$
- Mínimo: -2^{n-1}

¿Cómo almacenar un número en el formato?

1) Pasar el número a base 2.
2) Completar con ceros a izquierda.
3) Si es un número negativo complementar usando método "Complemento a 2".

Ejemplo: -125_{10} BPF con signo 8 bits
1) $-125_{10} = -1111101_2$
2) 01111101_2
3) Método Complemento a 2:
   a. Aplico el Not (01111101_2) = 10000010_2
   b. Sumo 1
   c. 10000011_2 → Resultado final BPF c/s 8 bits

En el ejemplo anterior podemos notar que una vez complementado, se obtiene el bit del signo 1.

¿Cómo recuperamos un número en el formato?

1) Si el bit de signo (primero de izquierda) es 1, eso implica que el número es negativo, por lo tanto, debemos complementarlo antes, sino dejamos el número tal cual está.
2) Quitar los 0 a izquierda.
3) Pasar de base 2 a base 10
4) Colocar el signo que corresponde.

Expansión y Truncamiento

- Expansión: Se completa con el bit de signo a la izquierda.
  Ej: 10110101 BPF c/s 8 bits → BPF c/s 16 bits
  11111111**1**0110101
- Truncamiento: Se extraen bits a izquierda siempre y cuando no se esté alterando el bit de signo del número.
  Ej: 00000011 BPF c/s 8 bits → BPF c/s 4bits
  ~~0000~~0011
  00001011 BPF c/s 8 bits → BPF c/s 4 bits
  ~~0000~~**1**011 No se puede ya que representan números diferentes

Formato Empaquetado

- Base: 16.
- Representa: Enteros positivos y negativos.
- Máximo: 10^{2n-1}-1
- Mínimo: -10^{2n-1}+1

9	9	9	9	9	9	9	S

Un byte se divide en dos conjuntos de 4 bits (uno a la izquierda y el otro a la derecha) a cada grupo se lo denominado **NIBBLE**, en otras palabras, el **NIBBLE** es el conjunto de cuatro dígitos binarios.

¿Cómo guardar un número en el formato?

1) Pasar el número a base 10.
2) Colocar cada dígito decimal en un nibble, dejando el último nibble ya que en el mismo se almacena el signo.
3) Colocar en el último nibble el signo según:
   - C,A,F,E → indican positivo.
   - B,D → indican negativo.

Ejemplo: -127_{10} → Empaquetado de 3 bytes

0	0	1	2	7	**D**

Nota: se rellena con 0 hasta alcanzar la cantidad de bytes usados

¿Cómo recuperamos un número en el formato?

Se realizan en orden inverso los pasos para almacenar

Formato Zoneado

- Base: 16.
- Representa: Enteros positivos y negativos.
- Máximo: $10^n - 1$
- Mínimo: $-10^n + 1$

F	9	F	9	F	9	S	9

¿Cómo almacenar un número en el formato?

1) Pasar el número a base 10.
2) Colocar cada uno de los dígitos decimales en un Nibble derecho.
3) Completar todos los Nibbles de izquierdo con F salvo el último que se completa con el signo siguiendo las mismas reglas que para empaquetados.
   - C,A,F,E → indican positivo.
   - B,D → indican negativo

Ejemplo: -127 → Zoneado de 4 Bytes.

F	0	F	1	F	2	**B**	7

Nota: se rellena con F0 hasta alcanzar la cantidad de bytes usados

¿Cómo recuperamos un número en el formato?

Se realizan en orden inverso los pasos para almacenar

Formato Punto Flotante

Es la manera que tiene una arquitectura de representar a los números reales. Su notación científica se expresa de la siguiente manera: $M \times B^E$ donde M corresponde a la mantisa, B a la base y E al exponente. A continuación, se muestran diferentes notaciones científicas equivalentes para expresar 12.31:

- 12.31×10^0
- 1231×10^{-2}
- 1.231×10^1

La representación de números fraccionarios que necesita de una menor cantidad de dígitos en notación científica, es aquella que utiliza un punto decimal después de la primera cifra significativa de la mantisa. Esta forma de representación se la denomina *normalizada.* En otras palabras, un numero esta *normalizado* si el digito a la izquierda del punto esta entre 0 y la base (0 < dígito a la izquierda < B).
En particular, decimos que un numero binario esta normalizado si el digito de la izquierda del punto es igual a 1

Formato Binario Punto Flotante IEEE754

Un poco de historia:
Hasta 1980, cada fabricante tenía su propio sistema de representación de números de punto flotante. No sólo todos eran diferentes, sino que algunos cometían errores de aritmética, dado que los números de punto flotante tienen algunos detalles que no son obvios al diseñar hardware.

Para corregir esta situación, a fines de los ´70 la IEEE estableció un comité para fijar una representación estándar para los números de punto flotante, no sólo para permitir el intercambio de números de punto flotante entre diferentes computadoras, sino para dar también a los diseñadores de hardware un modelo correcto. Como resultado, en 1985 surgió el estándar 754 de la IEEE. En la actualidad, la mayoría de las CPU´s tiene una unidad de punto flotante, y todas éstas siguen al estándar fijado por la IEEE.

El estándar define tres formatos: precisión simple (32 bits), precisión doble (64 bits) y precisión extendida (80 bits).

Tanto los números de precisión simple como los de precisión doble usan base 2 para las fracciones y notación exceso (quiere decir que si el exceso es m cualquier número n se representará como n + m) para los exponentes. Los formatos utilizados se muestran en la **figura 2.**

Los dos formatos comienzan con un dígito indicando el signo del número, siendo 0 positivo y 1 negativo. Luego sigue el exponente, que se representa en notación exceso 127 para los números de precisión simple y exceso 1023 para los números de precisión doble. Los exponentes 0 y 255 ó 2047 (según la precisión) no se utilizan para números normalizados, si no que tienen usos especiales que se describirán luego. Finalmente se tiene la fracción, de 23 y 52 bits respectivamente.

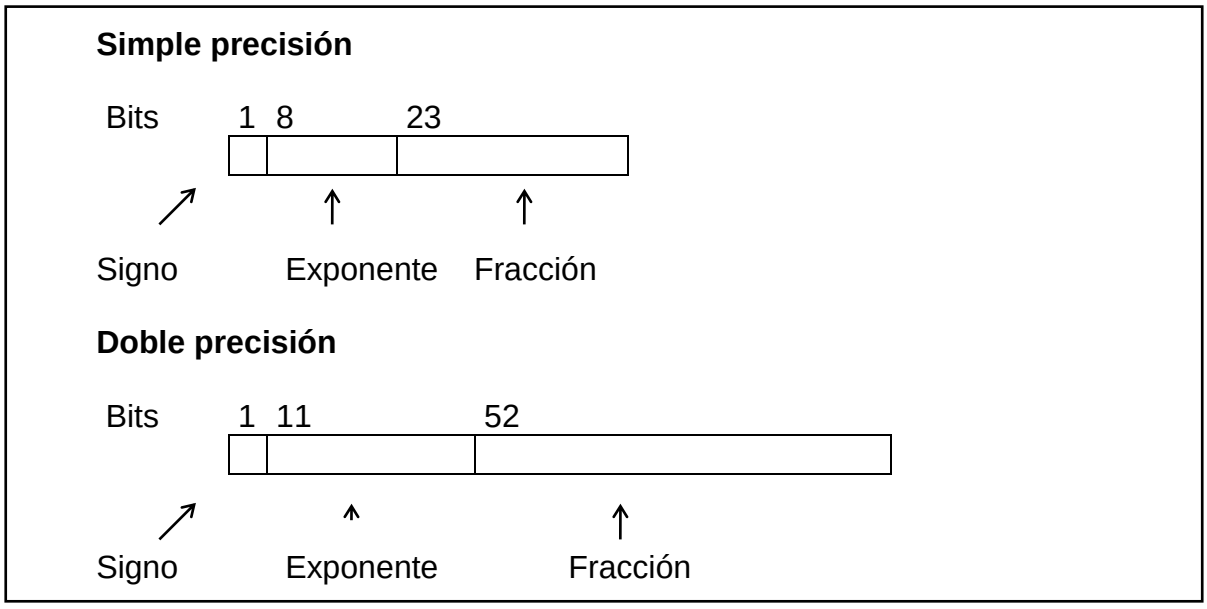

Figura 2: Esquema de los números de punto flotante de la IEEE

Una fracción normalizada comienza con el punto binario, seguido de un bit 1 y el resto de la fracción. Como se puede suponer que en una fracción normalizada siempre el primer bit después del punto será un 1, no es necesario guardar ese primer bit. Consecuentemente, el estándar define a la fracción de una manera un poco inusual. La fracción consiste de un primer bit 1 implícito, un punto binario implícito, y 23 o 52 bits arbitrarios. Si todos estos bits son cero, la fracción tiene un valor de 1,0; si todos son 1´s, la fracción es numéricamente apenas menor que 2. Todos los números normalizados tendrán una fracción *f*, en el rango $1 \leq f < 2$.

Ítem	Precisión simple	Precisión doble
Bits en el signo	1	1
Bits en el exponente	8	11
Bits en la fracción	23	52
Total de bits	32	64
Sistema del exponente	Exceso 127	Exceso 1023
Rango del exponente	-126 a 127	-1022 a 1023
Mínimo, normalizado	2^{-126}	2^{-1022}
Máximo, normalizado	aprox. 2^{+128}	aprox. 2^{+1024}
Rango decimal	aprox. 10^{-38} a 10^{38}	aprox. 10^{-308} a 10^{308}
Mínimo, renormalizado	aprox. 10^{-45}	aprox. $10^{.324}$

Tabla 3: Características de la representación de punto flotante del estándar de la IEEE

Ejemplo: Los números 0,5, 1 y 1,5 se representan en formato de punto flotante de precisión simple normalizado con las cadenas hexadecimales 3F000000, 3F800000 y 3FC00000

3F000000 representa a los bits 0 01111110 00000000000000000000000

- Signo: 0 (positivo)
- Exponente: $01111110_2 = 126 \rightarrow$ Exceso 127: 126 - 127 = -1
- Bits de la fracción: 00000000000000000000

Luego agregamos el 1 implícito: 1.00000000000000000000, y como resultado final obtenemos:

$$\mathbf{1{,}00000000000000000_2 \times 2^{-1} = 0{,}1_2 \times 2^0 = 0{,}5}$$

La verificación de los otros dos números se deja para el lector.

Ancho de Paso

Marca cuál es la distancia entre un flotante y su siguiente número representable en el formato. Es importante visualizar que el ancho de paso no es el mismo para todos los números flotantes. A mayor exponente mayor ancho de paso.

Overflow y Underflow

Al trabajar con números de punto flotante podemos encontrarnos con distintas situaciones en la ejecución de instrucciones con estos números. Los problemas que pueden surgir son los de Overflow y Underflow.

Ahora bien, pasemos a dar una breve definición de estos.
Overflow: Se refiere a la condición de que el exponente resultante exceda el límite superior, tanto para mantisas positivas como negativas, dando lugar a $+\infty$, $-\infty$.
Underflow: Se refiere a la condición de que el exponente resultante exceda el mínimo valor permitido y caiga en el intervalo $(-\infty, -0)$ y $(+0, +\infty)$. Es de observar que el verdadero cero al origen de coordenadas es una excepción. En otras palabras, cero no cae en el intervalo $(-\infty, -0)$ y $(0+, +\infty)$.

Como vimos anteriormente uno de los problemas que surgen al trabajar con números de punto flotante es el manejo del overflow, underflow y números no inicializados. El estándar trata con estos problemas de manera explícita, y define además de los números normalizados, otros cuatro tipos de números, que se muestran a continuación.

Normalizado	$\pm$	$0 < Exp < Max$	Cualquier patrón de bits
Desnormalizado	$\pm$	0	Cualquier patrón de bits $\neq 0$
Cero	$\pm$	0	0
Infinito	$\pm$	111…1	0
NAN	$\pm$	111…1	Cualquier patrón de bits $\neq 0$

Cuando el resultado de una operación es un número menor al mínimo número de punto flotante normalizado que puede ser representado, surge un problema. En un principio la mayoría de los diseños de hardware siguió una de dos alternativas para resolver este problema: Se establecía el resultado como cero y se continuaba, o se causaba una excepción de punto flotante. Ninguna de estas es realmente satisfactoria, así que la IEEE inventó los números **desnormalizados (subnormales).** Estos números tienen como exponente al cero[1] y el bit 1 que se hallaba implícito a la izquierda del punto binario es ahora un 0 implícito. Los números desnormalizados se distinguen de lo normalizados porque estos últimos no permiten al cero como exponente.

El menor número normalizado de precisión simple tiene un 1 como exponente y un cero como fracción, representando al 1.0×2^{-126}. El mayor número desnormalizado tiene un cero como exponente y todos 1´s como fracción, representando aproximadamente 0.999999×2^{-126}, valor muy similar al anterior. Pero vale la pena notar que este último tiene solamente 23 bits significativos, mientras que todos los normalizados tienen 24.

1 Cuando se dice que el exponente es cero se quiere decir que todos los bits que representan al exponente son cero. En el caso de los números desnormalizados no se emplea la notación exceso *m*, sino que el verdadero valor del exponente es fijo y dependerá de la precisión: -126 para los números de precisión simple y -1022 para los de doble.

A medida que las operaciones hacen a este número aún más chico, el exponente sigue siendo cero y los primeros bits de la fracción se transforman en ceros, reduciendo por lo tanto el valor de la fracción y la cantidad de bits significativos. El número desnormalizado más chico consiste de un 1 en la posición menos significativa de la fracción, y ceros en las posiciones restantes. En este caso el exponente es -126 y la fracción representa 2^{-23}, siendo entonces el valor del número $2^{-23}.2^{-126} = 2^{-149}$.

Existen dos formas de representar al cero en este esquema: una con el bit de signo en 1 y la otra con el bit de signo en 0. Ambas tienen un exponente cero y una fracción cero. También en este caso el bit a la izquierda del punto binario es un cero implícito en lugar de un uno implícito.

No existe para el overflow un manejo análogo al del underflow. No hay más combinaciones de bits disponibles. En cambio, se provee de una representación especial para infinito, que consiste de un exponente de todos 1´s (no permitido para números normalizados) y una fracción 0. Este número se puede usar como un operando y se comporta según las reglas matemáticas para el infinito. Por ejemplo, infinito + algo es infinito y cualquier número finito dividido infinito es cero. De manera similar, cualquier número finito dividido cero resulta en infinito.

¿Qué pasa al evaluar infinito dividido infinito? El resultado no está definido. Para manejar este caso se provee otro formato especial, llamado **Nan (Not a number)**, que también se puede usar como un operando con resultados predecibles.

Significados Especiales:
Como se mencionó anteriormente, otra de las características del formato IEEE 754 es la existencia de configuraciones especiales para representar sucesos inusuales:

Cero: Puesto que el significando se supone almacenado en forma normalizada, no es posible representar el cero (se supone siempre precedido de un 1). Por esta razón se convino que el cero se representaría con valores 0 en el exponente y en el significando. Ejemplo:
0 00000000 00000000000000000000000 = +0
1 00000000 00000000000000000000000 = -0

Infinitos: Se ha convenido que cuando todos los bits del exponente están a 1 y todos los del significando a 0, el valor es +/- infinito (según el valor S). Esta distinción ha permitido al Estándar definir procedimientos para continuar las operaciones después que se ha alcanzado uno de estos valores (después de un overflow).
Ejemplo:
0 11111111 00000000000000000000000 = +Infinito
1 11111111 00000000000000000000000 = -Infinito

Valores no-normalizados (denominados también "subnormales"). En estos casos no se asume que haya que añadir un 1 al significado para obtener su

valor. Se identifican porque todos los bits del exponente son 0 pero el significado presenta un valor distinto de cero (en caso contrario se trataría de un cero).
Ejemplo:
1 00000000 00100010001001010101010
Estos valores sirven para disminuir el hueco entre el 0 y el número normalizado más pequeño, Permiten reducir paulatinamente la magnitud de un número hasta llegar a 0.

Valores no-numéricos: Denominados NaN ("Not-a-number"). Se identifican por un exponente con todos sus valores a 1, y un significando distinto de cero. Existen dos tipos QNaN ("Quiet NaN") y SNaN ("Signalling NaN"), que se distinguen dependiendo del valor 0/1 del bit más significativo del significando, es decir si ese bit está seteado o no. QNaN tiene el primer bit a 1, y significa "Indeterminado", SNaN tiene el primer bit a 0 y significa "Operación no-válida". QNaN son los resultados de todas aquellas operaciones aritméticas con resultado matemáticamente no definido, mientras que SNaN es la ejecución de una operación invalidad.

Ejemplo:
0 11111111 10000100000000000000000 = QNaN
1 11111111 00100010001001010101010 = SNaN

Por ejemplo, dar como resultado una tira de bits que representa +1 o -1 como resultado de dividir por 0. El símbolo especial para este tipo de operaciones no valida es un NaN (Not a Number).

Operaciones Especiales:
A continuación, se muestran algunos resultados definidos por IEEE usando los significados especiales como operandos.

Operación	Resultado
n ÷ ±Infinito	0
±Infinito × ±Infinito	±Infinito
±n ÷ 0	±Infinito
Infinito + Infinito	Infinito
Cualquier operación contra un NaN	NaN
±0 ÷ ±0	NaN
Infinito - Infinito	NaN
±Infinito ÷ ±Infinito	NaN
±Infinito × 0	NaN

¿Cómo almacenar un número en el formato?

Usaremos los siguientes ejemplos para explicar como almacenamos un número en BPFlotante IEEE 754 de precisión simple:

- A = $-6{,}125_{10}$.
  1) El bit 31 tomará el valor del signo del número. (-6,125 → 1)
  2) Pasar a binario la mantisa decimal.
     $6=110_2$
     $0{,}125=0{,}001_2$
     $6{,}125=110{,}001_2$
  3) Normalizar. Correr la coma a derecha o izquierda hasta convertir el número binario en un número de la forma 1,……..
     El número de desplazamientos va a dar valor al exponente de forma que:
     - Desplazamiento a la derecha → Exponente negativo
     - Desplazamiento a la izquierda → Exponente positivo
       $6{,}125_{10}=110{,}001_2$ → 1,10001 → Exponente = 2
       2 expresado en Exceso 127 es 129 → 10000001_2
  4) Mantisa representada con bit implícito: **1**,10001 → 10001 (el bit **1** de la parte entera no se representa)
  5) El número final es 1 10000001 $10001000000000000000000_2$ (Se agregan a la derecha los "0" necesarios para completar los 23 bits de la mantisa)
  6) Pasado a hexadecimal 1 100 0000 1 100 0100 0000 0000 0000 $0000_2 = C0C40000_{16}$

- B = -134144_{10}
  1) Representación en binario: $-134144_{10} = -100000110000000000_2$
  2) Normalización: $-100000110000000000_2 = -1{,}0000011\square_2 \times 2^{17}{}_{10}$
  3) Mantisa de 23+1(bit imp.)+1(s) bits en módulo y signo:
     $1\ 10000011000000000000000_2$
  4) Exponente de 8 bits en exceso a 127:
     Exceso almacenar (EA) = (E+127) en base 10 ⇒ EA = 17+127 →144 base 10 → 10010000_2
  5) Resultado final →1 10010000 $00000110000000000000000_2$ □= $C8030000_{16}$

¿Cómo recupero un número en el formato?

El Procedimiento para pasar de coma flotante a decimal es el siguiente:

1) Convertir a binario el número hexadecimal
   $C0C40000_{16} = 1100\ 0000\ 1100\ 0100\ 0000\ 0000\ 0000\ 0000_2$
2) Identificar los campos de la configuración binaria
   1 10000001 $10001000000000000000000_2$
   - Signo de la mantisa
   - Exponente representado en exceso 127
   - Mantisa normalizada con bit implícito
3) Convertir cada uno de los campos a decimal
   - 1 → Mantisa negativa

- $10000001_2 = 129_{10}$ → +2 es el exceso
- $10001000000000000000000_2$

4) Corremos la coma binaria según el exponente:
   $1,10001000000000000000000_2$ → debemos desplazar la coma a la derecha en 2 posiciones, quedando $110,00100000000000000000_2$
5) El numero final es la combinación de todos los valores de los campos → $-6,125_{10}$

Ahora procedemos a realizar un ejemplo más complejo. Dada la siguiente configuración hexadecimal 42378000_{16} de un número almacenado en el estándar IEEE 754 de precisión simple, hallar el valor decimal de la cantidad representada.

1) Convertir a binario → $01000010001101111000000000000000_2$
2) Exponente: 10000100_2 en exceso 127 →132= E+127 →E = 5_{10}
3) Signo - mantisa: 0 - **1**01101111000000000000000 (añadiendo el bit implícito, en negrita)
   Signo: 0
   Mantisa: $(-1)^0$ x $1,011011110...0_2 = 1,43359_{10}$
4) Total: X = 1,43359 x $2^5{}_{10} = 45,875_{10}$

Formato Binario de Punto Flotante (IBM mainframe)

Se trata del formato propietario que IBM desarrolló para sus equipos mainframe de la serie 360/370/390/z. Sus características particulares son:

- Base: 16.
- Representa: Enteros con coma decimal positivos y negativos.
- Precisión: Simple (4 bytes), Doble (8 bytes) y Extendida (16 bytes).

Estructura

S nnnnnnn dddddd

Donde

- S = signo → 1 para negativos y 0 para positivos
- n = dígitos de la característica, en total son 7 bits y se usan para calcular el exponente de la siguiente manera:
  C = E + 40_{16} donde E corresponde al exponente

- d = corresponde a la mantisa normalizada → $0,dddddd \times 10^e{}_{16}$

¿Cómo guardar un número en el formato?

1) Pasar el número a base 16.
2) Normalizar en base 16.
3) Calcular la característica.
4) Armar el formato anteponiendo el bit de signo a la característica.

Ejemplo: - $321,54_{10}$ → Binario de punto flotante precisión simple.

1) $321{,}54_{10} = 141{,}8A3_{16}$
2) $0{,}1418A3 \times 10^{3}{}_{16}$
3) $C = E + 40_{16} = 3 + 40_{16} = 43_{16}$ en base 2 sería 100011_2
4) Agregamos el bit de signo: **1**100011_2 que en base 16 es $C3_{16}$
5) Resultado final: **$C31418A3_{16}$**

Ancho de Paso
Distancia entre un flotante y su siguiente número representable en el formato.
Ejemplo:

$$\begin{array}{l} 0{,}FE04AC \times 10^{-1A}{}_{16} \\ -\ 0{,}FE04AB \times 10^{-1A}{}_{16} \\ \hline \mathbf{0{,}000001 \times 10^{-1A}{}_{16}} \end{array} \quad \mathbf{\Rightarrow\ 0{,}1^{-1F}{}_{16}} \quad \mathbf{ANCHO\ DE\ PASO}$$

Absorción:
Se da en las operaciones de suma y resta entre flotantes.
Ejemplo: realizar la suma entre los siguientes números:

$A = 0{,}15A4 \times 10^{5}{}_{16}$
$B = 0{,}54F \times 10^{-2}{}_{16}$

Para poder operar entre flotantes debemos igualar los exponentes llevándolos al mayor de todos, en este caso 5.

$$\begin{array}{l} 0{,}15A400 \times 10^{5} \\ +\ 0{,}00000005AF \times 10^{5} \\ \hline \mathbf{0{,}15A400 \times 10^{5}{}_{16}} \end{array} \rightarrow \mathbf{Fenómeno\ de\ Absorción.\ A\ absorve\ a\ B.}$$

Lo mínimo que se puede sumar es el ancho de paso del número de mayor exponente.

Cadenas de caracteres

En una cadena de caracteres cada carácter ocupa 1 byte y se representa según el código de caracteres que se esté utilizando (ASCII o EBCDIC).

Ejemplos:

- La configuración hexadecimal de 4 bytes **48 4E 4C 41** representa, según la tabla de códigos ASCII, a la cadena de caracteres HOLA.
- Mientras que, si usamos el código EBCDIC, HOLA se representa con la configuración hexadecimal **C8 D6 D3 C1**.

Unicode

Es la codificación de caracteres universal comenzada a diseñarse a fines del año 1987 por ingenieros de Microsoft y Xerox: Joe Becker, Lee Collins y Mark Davis quienes publicaron en agosto de 1989 la primera versión borrador.
Durante el año 1989 continuó el trabajo y se sumaron mas colaboradores de Microsoft y Sun Microsystems y en febrero de 1991 crearon el llamado "Consorcio Unicode" quien publica en octubre del mismo año la primera versión estándar.

Antes de la creación de Unicode existían muchos sistemas distintos para la codificación, pero ninguno suficiente para representar todos los caracteres y símbolos de todos los idiomas.
Uno de los principales problemas que esto acarreaba es que toda computadora (especialmente servidores) al necesitar compatibilidad con muchos sistemas de codificación distintos generaba riesgo de que los datos sufran daños.

Codificación
La codificación está organizada en planos. Cada plano tiene un espacio de 16 bits y la cantidad de planos definida es 17 (plano 0 al plano 16). Por lo tanto la cantidad de códigos posibles es $17*2^{16} = 1.114.112$

La notación de un código unicode se expresa de la siguiente manera:

U-XXXXYYYY

Notación de un código Unicode

Siendo
- XXXX el valor hexadecimal del plano.
- YYYY el valor hexadecimal correspondiente al caracter dentro del plano.

Ej.: U-00000040 significa plano 0, código 40_{16} y corresponde específicamente a la representación de la letra **"M"**.

Para el caso particular del plano 0 suele usarse una notación simplificada la cual omite expresar el número de plano y reemplazar el "–" posterior a la U por el signo "+" → U+YYYY. Para el caso del ejemplo quedaría **U+0040**

Por lo tanto el rango total seria 00000000 al 0010FFFF, o bien [0..10FFFF], para lo cual se necesitan como máximo 21 bits ($10FFFF_{16}$ = 10000 1111 1111 1111 1111_2).
Notar que el 10 previo al FFFF corresponde al hexadecimal del mayor plano que es el 16.

El espacio final que ocupa y la forma en que se almacena un carácter Unicode depende la forma de transformación (Transformation Format) que se implemente.

UTF – Unicode Transformation Format
El UTF es un mapeo algorítmico de un código Unicode a una secuencia de bytes.
Hay 3 tipos de algoritmos según la longitud (en bits) de representación:
- UTF-8
- UTF-16
- UTF-32

UTF-8
Según el rango al cual corresponde dentro de la tabla Unicode, un caracter puede representarse como 1, 2, 3 o 4 tiras de 8 bits cada una.

A continuación, la representación según los rangos:

Rango [00...7F]
Se necesitan **7 bits** ($X_1X_2X_3X_4X_5X_6X_7$) y se representan en **1 byte**

- Primer bit en 0 y el resto igual que el código ASCII

$$0X_1X_2X_3X_4X_5\ X_6X_7$$

Rango [80...7FF]
Se necesitan **11 bits** ($X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}$) y se representan en **2 bytes**

- Byte 1
  - Los primeros **3 bits** son fijos 110. Los 2 primeros en 1 previos a un 0 dan la señal de que se usaran 2 bytes para la representación.
  - Los **5 bits** siguientes contienen los 5 primeros bits del caracter según la tabla Unicode
- Byte 2
  - Los primeros **2 bits** son fijos 10
  - Los **6 bits** siguientes contienen los últimos 6 del caracter según la tabla Unicode

$$110X_1X_2X_3X_4X_5\ 10X_6X_7X_8X_9X_{10}X_{11}$$

Rango [800...FFFF]
Se necesitan **16 bits** ($X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}$) y se representan en **3 bytes**

- Byte 1
  - Los primeros **4 bits** son fijos 1110. Los 3 primeros en 1 previos a un 0 dan la señal de que se usaran 3 bytes para la representación.
  - Los **4 bits** siguientes contienen los 4 primeros del caracter según la tabla Unicode
- Byte 2
  - Los primeros **2 bits** son fijos 10
  - Los siguientes **6 bits** contienen los siguientes 6 del caracter según la tabla Unicode
- Byte 3
  - Ídem segundo byte

$$1110\ X_1X_2X_3X_4\ 10X_5X_6X_7X_8X_9X_{10}\ 10X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}$$

Rango [10000... 10FFFF]
Se necesitan **21 bits** ($X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}X_{19}X_{20}X_{21}$) y se representan en **3 bytes**

- Byte 1
  - Los primeros **5 bits** son fijos 11110. Los 4 primeros en 1 previos a un 0 dan la señal de que se usaran 4 bytes para la representación.
  - Los **3 bits** siguientes contienen los 3 primeros del caracter según la tabla Unicode

- Byte 2
  - Los primeros **2 bits** son fijos 10
  - Los siguientes **6 bits** contienen los siguientes 6 del caracter según la tabla Unicode
- Byte 3
  - Ídem segundo byte
- Byte 4
  - Ídem segundo byte.

$$11110X_1X_2X_3\ 10X_4X_5X_6X_7X_8X_9\ 10X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}\ 10X_{16}X_{17}X_{18}X_{19}X_{20}X_{21}$$

Ejemplo UTF-8

Representación de la letra ***ñ***

- El código Unicode de la ***ñ*** es el $00F1_{16} = 0000\ 0000\ 1111\ 0001_2$
- Se encuentra en el rango entre el "80 y 7FF", por lo tanto se necesitan 11 bits (00011110001) y se usarán 2 bytes para la representación:
  - Byte 1: 11000011
  - Byte 2: 10110001

Representación Unicode **11000011****10****110001**$_2$ = **C3B1**$_{16}$

UTF-16

Según el rango al cual corresponde dentro de la tabla Unicode, un caracter puede representarse como 1 o 2 tiras de 16 bits cada una.

Rango [0...FFFF]

Se necesitan 16 bits y se representan en 2 bytes tal cual se define en la tabla Unicode

$$X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}$$

Rango [10000... 10FFFF]

Siendo **U** el código unicode del caracter a representar, y dado que el máximo valor es 10FFFF se calcula:

$$C = U\text{-}10000 \qquad \rightarrow \text{donde } C \text{ siempre tendrá 20 bits}^2$$

Se necesitarán 20 bits ($X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}X_{19}X_{20}$) y se representan en **4 bytes:**

- Byte 1 y 2
  - Los primeros **6 bits** son fijos 110110[3]
  - Los restantes **10 bits** son los primeros 10 de *C*

[2] El máximo código Unicode a representar es el 10FFFF y restándole 10000 a este valor se obtiene FFFF que es el máximo en 20 bits.

[3] Cuando un caracter en UTF-16 requiere 2 datos, el primero estará en el rango [D800..DFFF] y el segundo en el [DC00...DFFF]. Y el formato NO define ningún caracter en esos rangos justamente para permitir que esta codificación sea posible

- Byte 3 y 4
  - primeros **6 bits** son fijos 110111^3
  - Los restantes **10 bits** son los últimos 10 de *C*

$$110110X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}\ 110111X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}X_{19}X_{20}$$

Ejemplo UTF-16:

Representación del símbolo llamado clave de sol 𝄞

- El código Unicode es el $1D11E_{16}$
- C = 10000 - 1D11E = $0D11E_{16}$ = **0000 1101 0001 0001 1110**$_2$
  - Byte 1 y 2: 110110**0000 1101 00**
  - Byte 3 y 4: 110111**01 0001 1110**

Representación Unicode:

110110000011010011011101000100011110$_2$ = **D834 DD1E**$_{16}$

UTF-32

Es la mas simple de las 3 formas ya que todos los caracteres se representan en 32 bits en forma idéntica a como se codifican en la tabla Unicode.

Decodificación y detección de errores

UTF-8

Al decodificar una tira de bits codificados en UTF-8 se mira mirar el primer bit

- Si es 0, entonces el caracter a decodificar está en los siguientes 7 bits.
- Si es 1:
  - Si el siguiente 0, es un error.
  - Sino hay que contar cuantos unos (1) en total hay antes del primer 0.
  - Si son más de 4, es un error.
  - Sino, los siguientes N bytes (siendo N la cantidad de unos encontrados) deben comenzar con los bits 10, sino, es error.
  - El código Unicode final se arma con los últimos 7-N bits del primer byte seguido de los últimos 6 bits de los bytes siguientes.

UTF-16

Para decodificar una tira de bits codificados en UTF-16 se comienza evaluando los primeros 6 bits del primer byte.

- Si son distintos de 110110, es un dato coficado en 2 bytes y se lo busca en la tabla Unicode
- Sino, se verifican los primeros 6 bits del tercer byte:
  - Si son distintos de 110111 es un error.
  - Sino se arma el código Unicode con los últimos 2 bits del primer byte, seguidos por los 8 bits del segundo byte. A esto se le suman los últimos 2 bits del siguiente byte, mas los 8 bits del último byte.