Representación Computacional de Datos

Diego Feroldi
feroldi@fceia.unr.edu.ar

Arquitectura del Computador
Departamento de Ciencias de la Computación
FCEIA-UNR

Setiembre 2018

Índice

1. Introducción **1**

2. Organización de los datos **1**
2.1. Bits 1
2.2. Nibble 2
2.3. Byte 2
2.4. Palabra 3

3. Sistemas de numeración posicionales **3**

4. Sistema binario **5**
4.1. Formatos binarios 6
4.2. Conversión entre sistemas 7
4.3. Operaciones elementales en sistema binario 8
4.4. Números con signo 10
4.5. Complementos a la base y a la base menos uno 11
4.5.1. Definición de complemento a la base 11
4.5.2. Operación resta utilizando complemento a la base . . . 12
4.5.3. Definición de complemento a la base menos uno 12
4.5.4. Definición de complemento a dos 13
4.5.5. Definición de complemento a uno 15
4.6. Operaciones en complemento a dos 16
4.7. Operaciones en complemento a uno 17
4.8. Las banderas *Carry* y *Overflow* en aritmética binaria 18

5. Otras representaciones **19**
5.1. Sistema hexadecimal 19
5.1.1. Operaciones en sistema hexadecimal 21
5.2. Representación octal 22
5.3. Representación hexadecimal y octal en C 23
5.4. Representación BCD 23
5.4.1. Suma en formato BCD 24

6. Operaciones para números de precisión arbitraria **25**

1. Introducción

La aritmética que utilizan las computadoras difiere de la que estamos acostumbrados a usar por lo cual es importante estudiar primero como se realiza la representación de datos dentro la computadora antes de abordar temas más específicos de programación. La diferencia fundamental radica en que las computadoras solo pueden trabajar con números de precisión finita y, además, esta precisión generalmente es fija. Por otra parte, la mayoría de las computadoras trabajan en sistema binario en lugar del sistema decimal al que estamos habituados a utilizar. Por eso, haremos una revisión de los sistemas de numeración posicionales y, en particular, del sistema binario. Veremos también en este apunte otros dos sistemas de numeración muy útiles en computación: el hexadecimal y el BCD. En primer lugar, veremos como se organizan los datos dentro de la computadora.

2. Organización de los datos

Cuando escribimos números en formato digital en papel podemos utilizar un número arbitrario de dígitos. Análogamente, cuando escribimos números binarios en papel podemos utilizar un número arbitrario de dígitos binarios[1]. Sin embargo, esto no es posible dentro de la computadora. Las computadoras trabajan con cantidades especificas de bits dependiendo de la máquina en particular (actualmente 64 bits). Veremos a continuación que se suele trabajar con grupos de bits y que estos grupos reciben denominaciones en particular. En primer lugar, como ya se mencionó, un grupo de un solo dígito binario recibe el nombre de *bit*. Por otra parte, un grupo de 4 bits recibe el nombre de *nibble*[2], un grupo de ocho bits se denomina *byte*, un grupo de 16, 32 ó 64 bits se denomina *palabra* (*word*), dependiendo de la máquina en cuestión. Análogamente, *Dword* el doble que una palabra y *Qword* cuatro palabras. Estos tamaños no son arbitrarios. Cuando se diseña una arquitectura de computación, la elección de la longitud de palabra es una cuestión de suma importancia.

2.1. Bits

El bit es la mínima unidad de información en un sistema binario. Dado que solo puede representar dos valores distintos (cero o uno), en principio

[1] Ya veremos que a los dígitos en formato binario se le da un nombre particular: *bits*.

[2] Las denominaciones se dan directamente en inglés dado que es la forma usual en que las vamos a encontrar en la bibliografía.

pareciera que no es de mucha utilidad pero en realidad se puede usar para representar una gran cantidad de cosas: verdadero o falso, encendido o apagado, masculino o femenino, presente o ausente, etc. Todo depende de como definamos la estructura de datos. Por ejemplo, podemos definir que si el bit es cero representa que algo es falso mientras que si el bit es uno representa que es verdadero. Notar que esto es una convención y que aunque es muy usada tiene notables excepciones: bash y comandos unix.

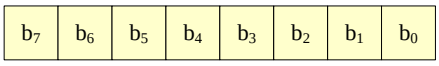
2.2. Nibble

Un nibble es una colección de cuatro bits. No es una estructura de datos muy interesante excepto en dos casos: números en formato BCD (binary coded decimal) y números en formato hexadecimal. En efecto, con un nibble podemos representar 16 valores distintos. En el caso de números hexadecimales, los valores 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, y F. En formato BCD se usan diez dígitos diferentes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), por lo tanto también se requiere cuatro bits. Por lo tanto, con un nibble (4 bits) podemos representar un dígito BCDm un dígito octal o un dígito hexadecimal como veremos más adelante.

2.3. Byte

La estructura de datos más utilizada en los microprocesadores 80x86 es el byte. Un byte consiste en ocho bits y es la estructura de datos más pequeña que se puede direccionar en un microprocesador 80x86. En efecto, la memoria principal en 80x86 está direccionada por bytes. Esto significa que el ítem más pequeño al que se puede acceder individualmente mediante un programa 80x86 es un valor de 8 bits. Para acceder a cualquier ítem más pequeño es necesario leer el byte que contiene el dato y "enmascarar[3]" los bits no deseados.

Los bits en un byte se numeran normalmente desde cero hasta siete usando la siguiente convención:



El bit 0 es el bit de menor orden o bit menos significativo mientras que el bit 7 es el bit de mayor orden o bit más significativo. Todos los bits se pueden referir por sus respectivos números.

[3]En informática se denomina máscara al conjunto de datos que junto con una determinada operación permite extraer selectivamente ciertos datos almacenados en otro conjunto.

Notar que un byte contiene exactamente dos nibbles. Los bits $0 \ldots 3$ comprenden el *low order nibble*, mientras que los bits $4 \ldots 7$ forman el *high order nibble*. Dado que un byte contiene exactamente dos nibbles, un byte requiere dos dígitos hexadecimales para ser representados, como se verá en la Sección 5.1.

2.4. Palabra

Una *palabra* es un conjunto de n bits que son manejados como un conjunto por la máquina. Por lo tanto, numeraremos los bits desde el cero hasta el $n-1$. Análogamente al byte, el bit cero es el menos significativo mientras que el $n-1$ es el más significativo. Notar que una palabra de 16 bits contiene dos bytes y por tanto 4 nibbles como se observa en la siguiente figura:

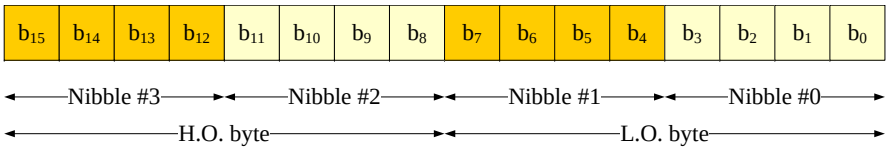


Figura 1: Bytes y nibbles en una palabra de 16 bits.

Con n bits se pueden representar 2^n valores diferentes. Las máquinas más actuales trabajan con palabras de 64 bits. Uno de los principales usos de las palabras consiste en representar valores enteros (*Integers*). Con una palabra de n bits se pueden representar enteros sin signo en el rango $[0,\ 2^n - 1]$ o enteros con signo en el rango $[-2^{n-1},\ 2^{n-1} - 1]$.

Los valores numéricos sin signo (*Unsigned*) se representan directamente por el valor en binario de los bits. Para representar valores numéricos con signo existen diferentes enfoques. Uno de los más utilizados es el concepto de complemento a dos (ver Sección 4.5).

3. Sistemas de numeración posicionales

Para representar números, lo más habitual es utilizar un sistema posicional de base 10, llamado *sistema decimal*. En este sistema, los números son representados usando diez diferentes caracteres, llamados dígitos decimales: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. La magnitud con la que un dígito dado contribuye al valor del número depende de su posición en el número de manera tal que si el dígito a ocupa la posición n a la izquierda del punto decimal (o coma)[4],

[4]En este apunte utilizaremos el punto como símbolo para indicar la separación entre la parte entera y la parte fraccional.

el valor con que contribuye es $a \times 10^{n-1}$, mientras que si ocupa la posición n a la derecha del punto decimal, su contribución es $a \times 10^{-n}$. Por ejemplo, la secuencia de dígitos 123.59 significa

$$123.59 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 9 \times 10^{-2}.$$

En general, la representación decimal

$$(-1)^s(a_n a_{n-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots)$$

corresponde al número

$$(-1)^s(a_n 10^n + a_{n-1} 10^{n-1} + \ldots + a_1 10^1 + a_0 10^0 + a_{-1} 10^{-1} + a_{-2} 10^{-2} + \ldots),$$

donde s depende del signo del número ($s = 0$ si el número es positivo y $s = 1$ si es negativo). De manera análoga, se pueden concebir otros sistemas posicionales con una base distinta de 10.

En principio, cualquier número natural $\beta \geq 2$ puede ser utilizado como base. Entonces, fijada una base, todo número real admite una *representación posicional* en la base β de la forma

$$(-1)^s(a_n \beta^n + a_{n-1} \beta^{n-1} + \ldots + a_1 \beta^1 + a_0 \beta^0 + a_{-1} \beta^{-1} + a_{-2} \beta^{-2} + \ldots),$$

donde los coeficientes a_i son los dígitos en el sistema con base β, esto es, enteros positivos tales que $0 \leq a_i \leq \beta - 1$. Los coeficientes a_i con $i \geq 0$ se consideran como los dígitos de la parte entera, en tanto que los a_i con $i < 0$, son los dígitos de la parte fraccionaria. Si utilizamos un punto para separar tales partes, el número es representado en la base β como

$$(-1)^s(a_n a_{n-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots)_\beta,$$

donde hemos utilizado el subíndice β para evitar cualquier ambigüedad con la base escogida. En efecto, a lo largo de este apunte utilizaremos un subíndice en cada número para indicar cual es su base y por lo tanto en qué sistema está representado. Por ejemplo:

- $(101001000101)_2$, sistema binario
- $(2629)_{10}$, sistema decimal
- $(5105)_8$, sistema octal
- $(0A45)_{16}$, sistema hexadecimal

4. Sistema binario

Una de las grandes ventajas de los sistemas posicionales es que se pueden dar reglas generales simples para las operaciones aritméticas. Además, tales reglas resultan más simples cuanto más pequeña es la base. Esta observación nos lleva a considerar el sistema de base $\beta = 2$, o sistema binario, en donde sólo tenemos los dígitos 0 y 1. Pero existe otra importante razón. Una computadora, en su nivel más básico, solo puede registrar si fluye o no electricidad por cierta parte de un circuito. Estos dos estados pueden representar entonces dos dígitos, convencionalmente, 1 cuando hay flujo de electricidad y 0 cuando no lo hay. Con una serie de circuitos apropiados una computadora puede entonces contar (y realizar operaciones aritméticas) en el sistema binario. El sistema binario consta, pues, solo de los dígitos 0 y 1, llamados *bits* (del inglés ***b**inary dig**its***). El 1 y el 0 en notación binaria tienen el mismo significado que en notación decimal:

$$0_2 = 0_{10}, \qquad 1_2 = 1_{10}.$$

Se puede lograr una equivalencia entre sistemas de representación. Así, por ejemplo, 1101.01 es la representación binaria del número 13.25 del sistema decimal, esto es, $(1101.01)_2 = (13.25)_{10}$, puesto que,

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 13.25$$

Además del sistema binario, otros dos sistemas posicionales resultan de interés en el ámbito computacional, a saber, el sistema con base $\beta = 8$, denominado *sistema octal*, y el sistema con base $\beta = 16$, denominado *sistema hexadecimal*. El sistema octal usa dígitos del 0 al 7, en tanto que el sistema hexadecimal usa los dígitos del 0 al 9 y las letras A, B, C, D, E, F. Siguiendo la misma regla se pueden hallar las siguientes equivalencias:

$$(13.25)_{10} = (1101.01)_2 = (15.2)_8 = (\mathrm{D}.4)_{16}$$

El sistema hexadecimal se desarrollará en detalle en la Sección 5.1.

La gran mayoría de las computadoras actuales (y efectivamente todas las computadoras personales, o PC) utilizan internamente el sistema binario ($\beta = 2$). Las calculadoras, por su parte, utilizan el sistema decimal ($\beta = 10$). Ahora bien, cualquiera sea la base β escogida, todo dispositivo de cálculo solo puede almacenar un número finito de dígitos para representar un número. En particular, en una computadora solo se puede disponer de un cierto número finito fijo n de posiciones de memoria para la representación de un número. El valor n se conoce como longitud de palabra. Además, aun cuando en el

sistema binario un número puede representarse tan sólo con los dígitos 1 y 0, el signo "-" y el punto, la representación interna en la computadora no tiene la posibilidad de disponer de los símbolos signo y punto. De este modo una de tales posiciones debe ser reservada de algún modo para indicar el signo y cierta distinción debe hacerse para representar la parte entera y fraccionaria. Esto puede hacerse de distintas formas. En el apunte *Representación de Números Reales* veremos la representación de punto flotante. En este apunte solamente utilizaremos la representación en punto fijo.

4.1. Formatos binarios

Un número en formato binario se puede representar de diferentes maneras ya que cualquier bit cero a la izquierda del uno más significativo no tiene peso en el número. Por ejemplo, el número 6 se puede representar como $(110)_2$, como $(0110)_2$, como $(00000110)_2$, etc. Sin embargo, es usual utilizar una notación donde los ceros a la izquierda no se escriben. Otra convención usual, derivada de las máquinas 80x86, es trabajar con grupos de cuatro u ocho bits. Además, para mayor claridad, es usual separar los grupos de 4 bits con un espacio en blanco, análogamente a la práctica en decimal de poner un punto cada tres dígitos. Por ejemplo, el número seis lo podemos representar en binario como $(0000\ 0110)_2$.

Al momento de trabajar con números binarios, podemos necesitar referirnos a un bit en particular del número. Para ello le asignaremos un valor a cada posición del bit. De esta manera en números binarios con ocho bits tendremos desde el bit cero hasta el bit siete:

$$b_7 b_6 b_5 b_4\ b_3 b_2 b_1 b_0$$

El bit más a la derecha en un número binario es el bit con posición cero. Este bit se denomina bit menos significativo (LSB, *Least Significant Bit*). Cada bit a la izquierda va teniendo una posición mayor. El bit más a la derecha se denomina bit más significativo (MSB, *Most Significant Bit*). Análogamente, se procede para los decimales desde b_{-1}:

$$b_7 b_6 b_5 b_4\ b_3 b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} b_{-4}\ b_{-5} b_{-6} b_{-7} b_{-8}$$

En este caso el bit menos significativo es el b_{-8}.

4.2. Conversión entre sistemas

Binario a decimal

La conversión de binario a decimal es directa empleando la definición de sistema de numeración posicional ya vista:

$$\begin{aligned}(b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4})_2 &= (b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 + \\ &\quad b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4})_{10}\end{aligned}$$

Por ejemplo:

$$(0110.1101)_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} = (6.8125)_{10}$$

Decimal a binario

Para convertir un número de decimal a binario el primer paso es separar la parte *entera* de la parte *fraccionaria*. Para convertir la parte entera, un método de conversión consiste en realizar sucesivas divisiones por dos hasta llegar a cero e ir registrando los restos que resultan ser los bits del número en binario. El primer resto obtenido es b_0, el segundo es b_1 y así sucesivamente.

Por otro lado, para convertir la parte fraccionaria se realizan multiplicaciones sucesivas por dos de las partes fraccionarias (sin el entero) y se van registrando los dígitos enteros obtenidos. El primer dígito obtenido es b_{-1} y así sucesivamente.

Por ejemplo, para convertir el número $(149.56)_{10}$ a binario primero convertimos la parte entera dividiendo sucesivamente por dos hasta que el cociente entero sea 0 y registrando el valor de los restos, tal cual se muestra en el siguiente procedimiento:

149/2=74	Resto=1=b_0
74/2=37	Resto=0=b_1
37/2=18	Resto=1=b_2
18/2=9	Resto=0=b_3
9/2=4	Resto=1=b_4
4/2=2	Resto=0=b_5
2/2=1	Resto=0=b_6
1/2=0	Resto=1=b_7

Notar que el último resto se considera 1 aunque en realidad es menor que 1. Entonces, $(149)_{10} = (1001\ 0101)_2$.

Luego procedemos con la parte fraccionaria:

$$\begin{aligned}
0.56 \text{ x } 2 &= \mathbf{1}.12 \quad \rightarrow \text{b}_{-1}{=}1 \\
0.12 \text{ x } 2 &= \mathbf{0}.24 \quad \rightarrow \text{b}_{-2}{=}0 \\
0.24 \text{ x } 2 &= \mathbf{0}.48 \quad \rightarrow \text{b}_{-3}{=}0 \\
0.48 \text{ x } 2 &= \mathbf{0}.96 \quad \rightarrow \text{b}_{-4}{=}0 \\
0.96 \text{ x } 2 &= \mathbf{1}.92 \quad \rightarrow \text{b}_{-5}{=}1 \\
0.92 \text{ x } 2 &= \mathbf{1}.84 \quad \rightarrow \text{b}_{-6}{=}1 \\
0.84 \text{ x } 2 &= \mathbf{1}.68 \quad \rightarrow \text{b}_{-7}{=}1 \\
0.68 \text{ x } 2 &= \mathbf{1}.36 \quad \rightarrow \text{b}_{-8}{=}1
\end{aligned}$$

De esta manera, $(0.56)_{10} = (1000\ 11111)_2$ empleando 8 bits. Por lo tanto, uniendo ambos resultados obtenemos

$$(149.56)_{10} = (1001\ 0101.1000\ 1111)_2$$

En realidad se podría haber seguido operando, obteniéndose más dígitos binarios. Al haber truncado el número para utilizar solo 8 bits para la parte fraccionaria se ha cometido un error por truncamiento. Se puede ver que en realidad $(1001\ 0101.1000\ 11111)_2 = (149.5586)_{10}$, con lo cual el error absoluto de representación es $\Delta p = |p - p^*| = 0.0014$ y el error relativo es $\frac{|p-p^*|}{|p|} = 9.4026 \times 10^{-6}$, donde p es el verdadero valor y p^* es la aproximación del mismo, siempre que $p \neq 0$.

Si se hubieran empleado más bits para la parte fraccionaria el error hubiera sido menor. Se puede definir una *cota superior* de error en función del número de dígitos empleados para representar la parte fraccionaria, es decir:

$$|p - p^*| \leq \frac{1}{2}\beta^{-t},$$

donde t es la cantidad de dígitos empleados para la parte fraccionaria y β es la base empleada.

4.3. Operaciones elementales en sistema binario

Suma

En binario, la tabla de adición toma la siguiente forma:

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	**1**0

Note que al sumar $1 + 1$ es $(10)_2$, es decir, llevamos 1 a la siguiente posición de la izquierda (acarreo). En el sistema decimal esto es equivalente a sumar, por ejemplo, $8 + 7$ que resulta 15. Por lo tanto, el resultado es un 5 en la posición que estamos sumando y un 1 de acarreo en la siguiente posición a la izquierda.

Ejemplo, sumar $(0101)_2$ y $(0011)_2$:

$$\begin{array}{cccccc} & \mathbf{1} & \mathbf{1} & \mathbf{1} & & \\ & 0 & 1 & 0 & 1 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 1 & 0 & 0 & 0 & \end{array}$$

En sistema decimal sería $5 + 3 = 8$.

Resta

La tabla de resta toma la siguiente forma:

A	B	A-B
0	0	0
1	0	1
1	1	0
0	1	**1**

La resta $0 - 1$ se resuelve igual que en el sistema decimal, tomando una unidad prestada de la posición siguiente: $0 - 1 = 1$ y *me llevo* 1.

Ejemplo, restar $(0010)_2$ a $(1001)_2$:

$$\begin{array}{cccccc} & \mathbf{1} & \mathbf{1} & & & \\ & 1 & 0 & 0 & 1 & \\ - & 0 & 0 & 1 & 0 & \\ \hline & 0 & 1 & 1 & 1 & \end{array}$$

En sistema decimal sería $9 - 2 = 7$. Otra forma mucho más práctica de realizar las restas es utilizar el *complemento a dos* o el *complemento a uno*. Este tema se desarrolla en la Sección 4.5.

Multiplicación

La tabla de multiplicación binaria toma la siguiente forma:

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

La multiplicación de números en binario es muy sencilla dado que cero por cualquier número es cero y el uno es el elemento neutro de la multiplicación.

Ejemplo, multiplicar $(1100)_2$ por $(0110)_2$:

			1	1	0	0
		x	0	1	1	0
			0	0	0	0
	1	1	1	0	0	
1	1	1	0	0		
0	0	0	0			
1	0	0	1	0	0	0

En sistema decimal sería $12 \times 6 = 72$.

División

La tabla de división binaria toma la siguiente forma:

A	B	A / B
0	0	NaN
0	1	0
1	0	Infinito
1	1	1

La división de números en binario también es muy sencilla ya que en el cociente no son posibles otras cifras más que el uno y el cero. Es decir, si la división es posible entonces el divisor sólo podrá estar contenido una vez en el dividendo.

4.4. Números con signo

Hasta el momento hemos tratado con números sin signo donde todos los números son positivos. Obviamente, también necesitamos representar números negativos. Lo primero que se puede pensar para representar números negativos es emplear una analogía con el sistema decimal donde el número negativo se escribe como un número positivo precedido del signo "-". Este enfoque se conoce como formato *magnitud y signo*.

Para representar el número dentro de la computadora se emplea un bit, generalmente el más significativo, para representar el signo mientras que el resto de los bits indican la magnitud (valor absoluto) del número a representar. Por convención, el bit más significativo en uno indica un número negativo.

Por ejemplo, para representar el número $(-28)_{10}$ primero se convierte el número $(28)_{10}$ a binario, esto es $(0001\ 1100)_2$ y luego se indica que es negativo haciendo uno el bit más significativo: $(-28)_{10} = (\mathbf{1}001\ 1100)_2$.

Sin embargo, este enfoque tiene dos grandes desventajas. En primer lugar, tiene doble representación del cero ya que tanto $(\mathbf{1}000\ 0000)_2$ como $(\mathbf{0}000\ 0000)_2$ representan el cero, lo que complica una operación muy usual: la comparación con cero. Por otra parte, las operaciones aritméticas son más complejas ya que, por ejemplo, para realizar una suma primero hay que determinar si los dos números tienen el mismo signo y en caso afirmativo realizar la suma de la parte significativa. En caso contrario, restar el mayor del menor y asignar el signo del mayor (en valor absoluto). Por lo tanto, este enfoque casi no fue usado y veremos a continuación enfoques más prácticos basados en el *complemento del número*.

4.5. Complementos a la base y a la base menos uno

Los complementos son muy usados en los sistemas digitales para representar números negativos y, por lo tanto, al momento de realizar operaciones de resta. Existen dos tipos de complementos. El complemento a la base (β) y el complemento a la base menos uno ($\beta - 1$). Es decir, para los números binarios (donde la base es 2) existen los complementos a 2 y a 1. En base octal serían complemento a 8 y a 7. En el sistema decimal, complemento a 10 y a 9, etc. A continuación veremos en primer lugar como se define el complemento a la base y el complemento a la base menos uno. Luego, veremos las definiciones de complemento a dos y complemento a uno, correspondientes al sistema binario.

4.5.1. Definición de complemento a la base

Sea un número N representado con n dígitos, el complemento a la base β de N se define como

$$C_\beta^N = \begin{cases} \beta^n - N & \text{si } N \neq 0 \\ 0 & \text{si } N = 0 \end{cases} .$$

Esto se cumple para todos los números N positivos incluso con fracción decimal. El único caso especial a considerar es cuando la parte entera es cero. Esto se interpreta como que $n = 0$.

Ejemplos en base 10:

- Complemento a 10 de $(987)_{10}$:

En este caso $N = 987$ y $n = 3$, entonces:

$$C_{10}^{N} = 10^3 - 987 = 1000 - 987 = (13)_{10}$$

- Complemento a 10 de $(0.125)_{10}$:

  En este caso $N = 0.125$ y $n = 0$, entonces:

$$C_{10}^{N} = 10^0 - 0.125 = 1 - 0.125 = (0.875)_{10}$$

- Complemento a 10 de $(987.125)_{10}$:

  En este caso $N = 987.125$ y $n = 3$, por lo tanto

$$C_{10}^{N} = 10^3 - 987.125 = 1000 - 987.125 = (12.875)_{10}$$

Es importante notar que NO es lo mismo calcular el complemento de la parte entera y de la fracción decimal por separado y juntar los resultados.

4.5.2. Operación resta utilizando complemento a la base

El concepto de complemento a la base es útil para realizar la operación resta. Veamos un ejemplo en base 10. Queremos realizar al operación $25-13$. Esto lo podemos escribir como $25 - 13 = 25 + (-13)$. El complemento en base diez de 13 es $C_{10}^{13} = 10^2 - 13 = 87$. Entonces, $25 - 13 = 25 + (-13) = 25 + 87 = 112$. Notar que el resultado es correcto si ignoramos el dígito más significativo.

4.5.3. Definición de complemento a la base menos uno

Sea un número positivo N en base β con n dígitos enteros y m dígitos en la fracción decimal, se define el complemento a la base $\beta - 1$ de N como

$$C_{\beta-1}^{N} = \beta^n - \beta^{-m} - N.$$

Por ejemplo, para el complemento a 9 de $(987)_{10}$ tenemos que $N = 987$, $n = 3$ y $m = 0$, por lo tanto:

$$C_{9}^{N} = 10^3 - 10^0 - 987 = 1000 - 1 - 987 = (12)_{10}$$

Para el complemento a 9 de $(0.125)_{10}$ tenemos que $N = 0.125$, $n = 0$ y $m = 3$, entonces:

$$C_{9}^{N} = 1 - 10^{-3} - 0.125 = 0.999 - 0.125 = (0.874)_{10}$$

Para calcular el complemento a 9 de $(987.125)_{10}$, $N = 987.125$, $n = 3$ y $m = 3$, por lo tanto:

$$C_9^N = 10^3 - 10^{-3} - 987.125 = 1000 - 0.001 - 987.125 = (12.874)_{10}.$$

Observar que en este caso SÍ es lo mismo calcular el complemento de la parte entera y el de la fracción decimal por separado y juntar o sumar los resultados.

4.5.4. Definición de complemento a dos

Siguiendo la definición de complemento a la base, el complemento a dos de un número N que expresado en el sistema binario está compuesto por n dígitos se define como

$$C_2^N = 2^n - N.$$

Ejemplos:

- Para el complemento a 2 de $N = (1010\ 1100)_2$ tenemos que $n = 8$, entonces:

  $$C_2^N = (1\ 0000\ 0000)_2 - (1010\ 1100)_2 = (0101\ 0100)_2$$

  También podemos hacer el cálculo pasando al sistema decimal como paso intermedio:

  $$C_2^N = (2^8)_{10} - (172)_{10} = (256)_{10} - (172)_{10} = (84)_{10} = (0101\ 0100)_2$$

- Análogamente, el complemento a 2 de $(1010)_2$ es $(1\ 0000 - 1010)_2 = (0110)_2$

Método alternativo (primero) El cálculo del complemento a dos puede realizarse de manera muy sencilla mediante un método alternativo. En efecto, se puede ver que para calcular el complemento a 2 de un número binario sólo basta con revisar todos los dígitos desde el menos significativo hacia el más significativo y mientras se consiga un cero dejarlo igual. Al conseguir el primer número 1, dejarlo igual para luego cambiar el resto de ellos hasta llegar al más significativo. Así podemos decir rápidamente que el complemento a 2 de $(1010\ 0000)_2$ es $(0110\ 0000)_2$, que el complemento a 2 de $(111)_2$ es $(001)_2$, etc.

Método alternativo (segundo) Otra forma muy sencilla de hallar el complemento a 2 de un número binario es invirtiendo todos los dígitos (que como veremos a continuación es lo que se conoce como complemento a 1) y sumándole uno al resultado obtenido, esto es:

$$C_2^N = C_1^N + 1$$

Estos métodos son muy fáciles de realizar mediante puertas lógicas, donde reside su mayor utilidad.

Método de conversión de binario en complemento a dos a decimal Una forma práctica de realizar la conversión de binario en complemento a dos a decimal es la siguiente. Para una longitud de palabra de n bits, una secuencia de n dígitos binarios

$$a_{n-1}a_{n-2}\ldots a_1a_0$$

corresponde, en la representación complemento a dos, al entero

$$-a_{n-1}2^{n-1} + \sum_{j=0}^{n-2} a_j 2^j.$$

Para los enteros positivos $-a_{n-1} = 0$ y entonces $-a_{n-1}2^{n-1} = 0$. Así la representación complemento a dos y signo-magnitud coinciden para los enteros positivos. Sin embargo, para los enteros negativos ($a_{n-1} = 1$) el bit más significativo es ponderado con un factor de peso -2^{n-1}. En esta representación el 0 se considera positivo y es claro que hay una única representación del mismo, la cual corresponde a un bit de signo 0 al igual que los restantes bits.

Ejemplo:

La secuencia de dígitos binarios 10101101 corresponde al entero

$$-2^7 + 2^5 + 2^3 + 2^2 + 2^0 = -83$$

Rango representable El rango de valores decimales para n bits usando complemento a dos será:

$$-2^{n-1} \leq \ Rango \leq \ 2^{n-1} - 1$$

El total de números positivos (incluyendo el cero) será 2^{n-1} y el de negativos 2^{n-1}. Por ejemplo, con cuatro bits el rango representable es $-8 \leq Rango \leq 7$.

4.5.5. Definición de complemento a uno

De acuerdo a la definición de complemento a la base menos uno, el complemento a uno de un número N con n dígitos enteros y m dígitos en la parte fraccional se define como:

$$C_1^N = 2^n - 2^{-m} - N$$

Ejemplos:

- Para el complemento a 1 de $N = (1010\ 1100)_2$ sabemos que $n = 8$ y $m = 0$, entonces:

$$C_1^N = (1\ 0000\ 0000 - 1 - 1010\ 1100)_2 = (0101\ 0011)_2.$$

- Análogamente, el complemento a 1 de $N = (1010)_2$ es

$$C_1^N = (1\ 0000 - 1 - 1010)_2 = (0101)_2$$

- Para realizar el complemento a 1 de $N = (101.1)_2$, $n = 3$ y $m = 1$, por lo tanto:

$$C_1^N = (1000 - 0010 - 0101.1)_2 = (0000.1)_2$$

Analizando los dos primeros ejemplos se puede observar que para conseguir el complemento a 1 de un número binario tan solo basta con invertir todos los dígitos (esto quiere decir cambiar 0 por 1 y viceversa). Este método es muy simple y es el que habitualmente se realiza.

Rango representable El rango de valores decimales para n bits usando complemento a uno será:

$$-2^{n-1} + 1 \leq \ Rango \leq \ 2^{n-1} - 1$$

El total de números positivos (incluyendo el cero) será 2^{n-1} y el de negativos $2^{n-1} - 1$.

Por ejemplo, con cuatro bits el rango representable es $-7 \leq Rango \leq 7$.

En la Tabla 1 se encuentra el complemento a dos y el complemento a uno con enteros de 4 bits. Observar que el cero tiene dos representaciones posibles en complemento a uno, lo cual es una desventaja.

Tabla 1: Complemento a dos con enteros de 3 bits más un bit de signo

Decimal	Complemento a dos	Complemento a uno
7	**0**111	**0**111
6	**0**110	**0**110
5	**0**101	**0**101
4	**0**100	**0**100
3	**0**011	**0**011
2	**0**010	**0**010
1	**0**001	**0**001
0	**0**000	**0**000
0	**0**000	**1**111
-1	**1**111	**1**110
-2	**1**110	**1**101
-3	**1**101	**1**100
-4	**1**100	**1**011
-5	**1**011	**1**010
-6	**1**010	**1**001
-7	**1**001	**1**000
-8	**1**000	

4.6. Operaciones en complemento a dos

En primer lugar vamos a analizar como operar con números en complemento a la base de manera genérica y luego vamos a ver de manera particular el complemento a la base dos dada su gran utilidad.

Sean dos números A y B en base β. Si deseamos obtener la suma $S = A + B$, analizaremos los distintos casos posibles según sea el signo de cada uno de los números.

1. $A > 0$ y $B > 0$. En este caso la suma $S = A + B$ es correcta y el resultado es positivo ya que ambos números lo son.

2. $A < 0$ y $B < 0$. En este caso podemos escribir la suma como $S = (\beta^n - |A|) + (\beta^n - |B|) = \beta^n + \beta^n - (|A| + |B|)$, siendo la suma correcta si se ignora β^n. El resultado es negativo ya que ambos números lo son y queda expresado en el complemento a la base.

3. $A < 0$ y $B > 0$ y $|A| < |B|$. En este caso podemos escribir la suma como $S = (\beta^n - |A|) + B = \beta^n + (|B| - |A|)$, siendo la suma correcta si se ignora β^n. El resultado es positivo por la relación propuesta para los valores absolutos.

4. $A > 0$ y $B < 0$ y $|A| < |B|$. En este caso la suma se puede escribir como $S = A + (\beta^n - |B|) = \beta^n - (|B| - |A|)$, siendo la suma correcta pues el resultado es negativo y está expresado en el complemento a la base.

La conclusión es que cuando sumamos números positivos y negativos utilizando complemento a la base se obtendrá siempre el resultado correcto. Solo hay que ignorar β^n, es decir el acarreo, en los casos 2 y 3. Por lo tanto, la suma implementada con complemento a la base es muy simple, motivo por el cual es muy empleada. A continuación veremos de forma particular el caso con base $\beta = 2$ mediante ejemplos.

1. $A = 5$, $B = 12$, $S = A + B = 17$. En binario:
$$(00000101)_2 + (00001100)_2 = (00010001)_2$$
y acarreo $c = 0$.

2. $A = -5$, $B = -12$, $S = A + B = -17$. En binario:
$$(11111011)_2 + (11110100)_2 = (111101111)_2$$
y acarreo $c = 1$.

3. $A = -5$, $B = 12$, $S = A + B = 7$. En binario:
$$(11111011)_2 + (00001100)_2 = (00000111)_2$$
y acarreo $c = 1$.

4. $A = 5$, $B = -12$, $S = A + B = -7$. En binario:
$$(00000101)_2 + (11110100)_2 = (11111001)_2$$
y acarreo $c = 0$.

En todos los casos el resultado es correcto, ignorando el acarreo.

4.7. Operaciones en complemento a uno

Las operaciones en complemento a uno no son habituales. De todas maneras, si se quiere utilizar el complemento a uno es importante observar que al operar se debe sumar el acarreo obtenido al final de la adición/resta realizada, en caso de haberlo obtenido, para conseguir el resultado correcto. De lo contrario el resultado sería incorrecto.

Ejemplo:
$$(00000010)_2 + (11111110)_2 = (\mathbf{1}\ 00000000)_2$$
con acarreo $c = 1$. El resultado es correcto si sumamos este acarreo. Así, el resultado correcto es $(00000001)_2$.

4.8. Las banderas *Carry* y *Overflow* en aritmética binaria

En el apunte "Ensamblador y Arquitectura x86_64"se verá que el procesador cuenta con un registro especial llamado `rflags`, en el cual se refleja el estado del procesador. Esto significa que en este registro se encuentra la información acerca de la última operación realizada y ciertos bits de control que permiten cambiar el comportamiento del procesador. Dentro de esta información se encuentran dos banderas muy importantes: *carry flag* y *overflow flag*.

No hay que confundir la bandera *carry* con la bandera *overflow* al trabajar con aritmética binaria. Cada bandera puede ocurrir por sí sola, o ambas a la vez. La ALU[5] de la CPU[6] no sabe si el programador está realizando operaciones con signo o sin signo. La ALU siempre establece ambos indicadores de manera apropiada cuando se hace cualquier operación y el programador debe comprobar el estado de las bandera después de realizarse la operación matemática.

Si se está trabajando con números sin signo (*unsigned*), el programador debe chequear si la bandera *carry* está encendida indicando que el resultado es incorrecto. Por el contrario, el estado de la bandera *overflow* no tiene relevancia en aritmética sin signo. La bandera *overflow* solo es relevante si se está trabajando con números con signo (*signed*).

En resumen:

- En aritmética sin signo, comprobar la bandera *carry* para detectar errores.
- En aritmética con signo, comprobar la bandera *overflow* para detectar errores.

Las reglas para encender la bandera *carry* son dos:

1. Si la suma de dos números aritmética binaria entera causa un acarreo en el bit más significativo:

   Ejemplo:

   $$(1111)_2 + (0001)_2 = (1\ 0000)_2$$

   Efectivamente, si se está trabajando con números sin signo la bandera *carry* encendida (`CF=1`) indica que el resultado es incorrecto dado que $15 + 1 \neq 0$.

[5] *Arithmetic Logic Unit*
[6] *Central Processing Unit*

2. Si la resta de dos números requiere un "préstamo" en el bit más significativo:

   Ejemplo:
   $$(0000)_2 - (0001)_2 = (\mathbf{1}\ 1111)_2$$
   Efectivamente, si se está trabajando con números sin signo la bandera *carry* encendida (`CF=1`) indica que el resultado es incorrecto dado que $0 - 1 \neq 15$.

Por otra parte, las reglas para encender la bandera *overflow* son dos:

1. Si la suma de dos números con el bit de signo en cero produce un número con el bit de signo en uno:

   Ejemplo:
   $$(0100)_2 + (0100)_2 = (1000)_2$$
   Efectivamente, la bandera `OF=1` indica que el resultado es incorrecto dado que si se está trabajando con números con signo $4 + 4 \neq -8$.

2. Si la suma de dos números con el bit de signo en uno produce un número con el bit de signo en cero:

   Ejemplo:
   $$(1000)_2 + (1000)_2 = (0000)_2$$
   Efectivamente, la bandera `OF=1` indica que el resultado es incorrecto dado que si se está trabajando con números con signo $-8 + (-8) \neq 0$.

5. Otras representaciones

5.1. Sistema hexadecimal

Un problema importante del sistema binario es su "verbosidad", es decir, el considerable número de dígitos necesarios para representar un determinado valor. Por ejemplo, para representar $(158)_{10} = (1001\ 1110)_2$ son necesarios ocho cifras en lugar de las tres necesarias en el sistema decimal. Por lo tanto, una primera conclusión es que la representación en el sistema decimal es más compacta aunque sería mucho más complejo implementar una computadora cuyo microprocesador trabaje directamente con sistema decimal. Por

Tabla 2: Equivalencia entre diferentes sistemas de numeración.

Hexadecimal	Octal	Decimal	Binario	BDC
0	0	0	0000	0000 0000
1	1	1	0001	0000 0001
2	2	2	0010	0000 0010
3	3	3	0011	0000 0011
4	4	4	0100	0000 0100
5	5	5	0101	0000 0101
6	6	6	0110	0000 0110
7	7	7	0111	0000 0111
8	10	8	1000	0000 1000
9	11	9	1001	0000 1001
A	12	10	1010	0001 0000
B	13	11	1011	0001 0001
C	14	12	1100	0001 0010
D	15	13	1101	0001 0011
E	16	14	1110	0001 0100
F	17	15	1111	0001 0101

otra parte, aunque se puede realizar la conversión entre decimal y binario y viceversa, no es una tarea trivial (ver Sección 4.2).

El sistema hexadecimal (base 16) resuelve este problema de manera mucho más eficiente. En efecto, el sistema hexadecimal tiene las dos ventajas que necesitamos: i) la notación es muy compacta y ii) es muy simple convertir entre hexadecimal y binario, y viceversa. Por estos motivos, se utiliza mucho el sistema de numeración hexadecimal en el ámbito de la computación.

Dado que en hexadecimal la base es 16, cada dígito a la izquierda del punto representa el valor de ese dígito multiplicado por una potencia de 16 dependiendo de la posición del dígito. Cada dígito hexadecimal representa un valor dentro de un conjunto de 16 valores distintos entre 0 y $(15)_{10}$. Dado que existen solo 10 dígitos decimales, se necesitan 6 dígitos adicionales para representar el rango entre $(10)_{10}$ y $(15)_{10}$. Los símbolos que se utilizan son las letras del abecedario entre la A y la F. Así, se llega a la tabla de equivalencia entre sistemas mostrada en la Tabla 2. De esta manera, el número $(15\mathrm{E})_{16}$ representa a

$$(15\mathrm{E})_{16} = 1 \times 16^2 + 5 \times 16^1 + 14 \times 16^0 = (350)_{10},$$

dado que $(\mathrm{E})_{16} = (14)_{10}$.

Como se puede observar, el sistema hexadecimal es muy compacto y fácil de leer. Además la conversión entre hexadecimal y binario es muy fácil. La

Tabla 2 contiene toda la información necesaria. En efecto, para convertir de hexadecimal a binario simplemente hay que sustituir cada dígito hexadecimal por sus correspondientes cuatro bits. Por ejemplo, para convertir el número $(0AF3)_{16}$ a binario:

HEXADECIMAL	0	A	F	3
BINARIO	0000	1010	1111	0011

Para convertir un número de binario a hexadecimal es casi tan fácil. El primer paso es rellenar el número con ceros para asegurarse que el número de bits es múltiplo de cuatro. Luego, separar el número en grupos de cuatro bits y convertir cada grupo utilizando la Tabla 2. Por ejemplo, para convertir el número binario $(0001\ 0011\ 0010\ 1110)_2$ en hexadecimal:

BINARIO	0001	0011	0010	1110
HEXADECIMAL	1	3	2	E

Por lo tanto, $(0001001100101110)_2 = (132E)_{16}$.

5.1.1. Operaciones en sistema hexadecimal

A la hora de realizar operaciones en sistema hexadecimal se puede optar por hacer un cambio de sistema como paso intermedio o bien operar directamente en hexadecimal. Veamos la suma y la resta.

Suma

La suma se realiza de manera análoga a lo visto teniendo en cuenta el acarreo correspondiente. Ejemplo:

$$\begin{array}{cccc} & \mathbf{1} & & \\ & 1 & A & E \\ + & & 1 & 8 \\ \hline & 1 & C & 6 \end{array}$$

Resta

Para realizar la resta, podemos proceder restando dígito a dígito y teniendo en cuenta los acarreos. Ejemplo:

$$\begin{array}{cccc} & \mathbf{1} & & \\ & 1 & A & 6 \\ - & & 1 & C \\ \hline & 1 & 8 & A \end{array}$$

Un manera más simple es calcular el complemento a la base (C_{16}^N) del sustraendo y luego sumarlo al minuendo. Previamente, hay que igualar la cantidad de dígitos. En el ejemplo anterior primero se calcularía el complemento a 16 de N=01C. Para ello recordar la relación entre el complemento a la base y el complemento a la base menos uno, dado que es más simple calcular C_{15}^N:

$$\begin{aligned} C_{16}^N &= C_{15}^N + 1 \\ C_{16}^N &= \text{FE3} + 1 = \text{FE4} \end{aligned}$$

Entonces,

$$\begin{array}{cccc} & 1 & A & 6 \\ + & F & E & 4 \\ \hline 1 & 1 & 8 & A \end{array}$$

El resultado es correcto ignorando el primer uno.

También se podría haber realizado la resta convirtiendo a binario como paso intermedio, aprovechando la facilidad de conversión entre hexadecimal y binario:

$$\begin{array}{cccc} & 0001 & 1010 & 0110 \\ - & 0000 & 0001 & 1100 \\ \hline & 0001 & 1000 & 1010 \\ & 1 & 8 & A \end{array}$$

5.2. Representación octal

De manera totalmente análoga se puede trabajar con otras bases. Por ejemplo, otro sistema de representación muy utilizado es el **sistema octal**. Este sistema numérico tiene base 8, por lo cual utiliza 8 ocho cifras para la representación. Las cifras utilizadas son los número enteros del 0 al 7. En la Tabla 2 se puede ver la equivalencia entre sistemas. La conversión entre los mismos se puede realizar de manera análoga a lo ya visto lo visto. Tener en cuenta que para representar los 8 dígitos en sistema octal son necesarios 3 dígitos binarios. Ejemplo:

OCTAL	1	5	1	4
BINARIO	001	101	001	100

Por lo tanto, $(1514)_8 = (001101001100)_2$.

5.3. Representación hexadecimal y octal en C

A menudo es conveniente representar valores enteros dentro de un código escrito en lenguaje C utilizando notación hexadecimal u octal. La convención utilizada es la siguiente:

- En octal al valor correspondiente se le antepone un 0. Ejemplo: 046 es $46_8 = 38_{10}$.
- En hexadecimal se le antepone 0x o 0X. Ejemplo: 0x32 es $32_{16} = 50_{10}$.

Sin embargo, no hay que perder de vista que esto es solo a nivel código y que la representación interna dentro de la computadora será en binario en complemento a dos. Por lo tanto, supongamos que hacemos la siguiente asignación:

```
int A = 0x32;
```

El compilador convierte este valor a binario y no recuerda que fue introducido como hexadecimal. Por lo tanto, si luego se imprime el valor de `A` se mostrará el valor 50 y no 32, a menos que se indique expresamente el formato correspondiente:

```
printf("Hexadecimal: %x\n", A);
printf("Octal: %o\n", A);
```

Por otra parte, si queremos representar un valor negativo le anteponemos el signo "-":

```
int B = -0x32;
int C = -046;
```

5.4. Representación BCD

El sistema Decimal Codificado en Binario (en inglés *Binary-Coded Decimal* (BCD)) es un estándar para representar números decimales en el sistema binario en donde cada dígito decimal es codificado con una secuencia de 4 bits. Con esta codificación especial de los dígitos decimales en el sistema binario se pueden realizar operaciones aritméticas como suma, resta, multiplicación y división de números en representación decimal sin perder en los cálculos la precisión que normalmente ocurre con las conversiones de decimal a binario "puro" y viceversa. La conversión de los números decimales a BCD y viceversa es muy sencilla, pero los cálculos en BCD demandan más tiempo y tienen más complejidad que los realizados con números binarios puros. En primer lugar veamos la tabla de codificación de los dígitos BCD (Tabla 3).

Tabla 3: Representación de dígitos BCD

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

Entonces para convertir un número decimal a formato BCD reemplazamos cada dígito por su correspondiente equivalente en binario. Por ejemplo, para convertir el número $(138)_{10}$ en BCD reemplazamos cada uno de sus dígitos por su equivalente en BCD:

DECIMAL	1	3	8
BCD	0001	0011	1000

Entonces, $(138)_{10} = (0001\ 0011\ 1000)_{\text{BCD}}$. Notar que el número obtenido es diferente al que obtendríamos en formato binario "puro", el cual sería $(1000\ 1010)_2$. Es más, notar que en formato BCD son necesarios más bits para representar el mismo número.

5.4.1. Suma en formato BCD

Se pueden sumar dos números BCD utilizando las reglas de la suma binaria vistas anteriormente. Sin embargo, si una suma de 4 bits es mayor que 9 el resultado no es válido. En este caso, se debe sumar 6, $(0110)_2$, al grupo de 4 bits para saltar así los seis estados no válidos. Si se genera un acarreo al sumar 6, éste se suma al grupo de 4 bits siguiente. Ejemplo: $8 + 7 = 15$

$$\begin{array}{cccc} & 1 & 0 & 0 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

El resultado es correcto en binario pero 1111 NO es un número válido en BCD porque es mayor que 9. Entonces sumamos 6 a este resultado:

$$\begin{array}{cccccc} & & 1 & 1 & 1 & 1 \\ + & & 0 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 0 & 1 \end{array}$$

Esto se interpreta como 0001 0101, es decir 15 en BCD. Con este simple ejemplo se puede notar la mayor complejidad de las operaciones en BCD. Sin embargo, el formato BCD es muy común en sistemas electrónicos donde las operaciones se realizan directamente mediante sistemas digitales sin programación.

6. Operaciones para números de precisión arbitraria

La aritmética con precisión arbitraria no se utiliza como aritmética nativa de procesadores, sino sólo en librerías especializadas. La precisión máxima está limitado por los parámetros de la librería, la memoria disponible y el tiempo de procesador disponible. Veamos como se procede. La instrucción `add` del 80x86 suma dos números de 8, 16, or 32 bits. Luego de la ejecución de la instrucción, la *carry flag* queda seteada en uno si se produce un *overflow* en el bit MSB. Esta información se puede usar para realizar operaciones con precisión arbitraria de manera secuencial análogamente a como procedemos cuando sumamos números manualmente.

Por ejemplo, si queremos realizar $125 + 158$ primero sumamos $5 + 8 = 3$ y tenemos un acarreo de uno. Luego hacemos $2 + 5 + 1 = 8$ y así sucesivamente. El 80x86 procede de manera análoga excepto que en lugar de dígitos son palabras. En los pasos mostrados en la Figura 2 se indica cómo se procedería para sumar dos números de 48 bits representados cada uno con tres palabras.

Referencias

[1] Andrew S. Tanembaum , *Organización de computadoras: Un enfoque estructurado*, cuarta edición, Pearson Education, 2000

[2] Paul A. Carter, *PC Assembly Language*, Disponible en formato electrónico: http://www.drpaulcarter.com/pcasm/, 2006.

[3] M. Morris Mano, *Computer system architecture*, tercera edición, Prentice-Hall, 1993.

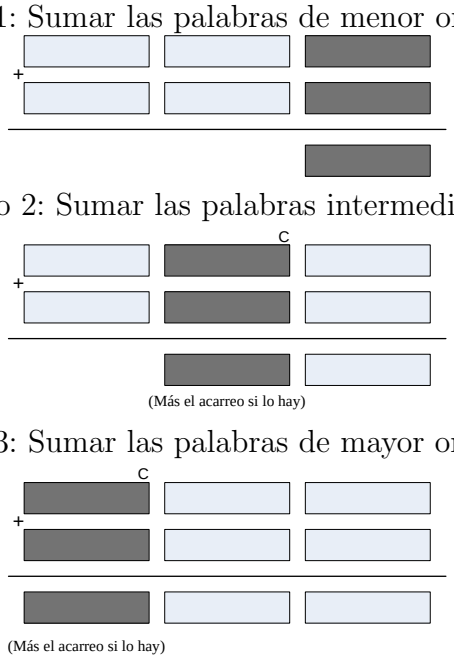


Figura 2: Ejemplo de suma con precisión arbitraria

[4] Randall Hyde, *The art of assembly language*, segunda edición, No Starch Pr, 2003.