Representación Computacional de Datos

Diego Feroldi
feroldi@fceia.unr.edu.ar

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

*Actualizado 4 de septiembre de 2025 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. Introducción

La aritmética utilizada por las computadoras difiere de la que empleamos habitualmente, por lo que resulta fundamental comprender primero cómo se representan los datos en su interior antes de abordar temas más específicos de programación. La principal diferencia radica en que las computadoras solo pueden operar con números de precisión finita, que en general es fija. Otra diferencia clave es que la mayoría de ellas operan en sistema binario (base dos), a diferencia del sistema decimal (base diez) al que estamos acostumbrados. Por esta razón, revisaremos los sistemas de numeración posicional, con especial énfasis en el sistema binario. También exploraremos otros dos sistemas de numeración ampliamente utilizados en computación: el hexadecimal y el octal.

2. 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)[1], 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 representa el valor:

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

En forma 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.

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

Cualquier número natural $\beta \geq 2$ puede utilizarse como base. Fijada una base, todo número real N admite una **representación posicional** en base β de la forma

$$\begin{aligned} N =& (-1)^s (a_n a_{n-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots)_\beta = \\ & (-1)^s (a_n \beta^n + a_{n-1}\beta^{n-1} + \ldots + a_1 \beta^1 + a_0 \beta^0 \\ & + a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + \ldots), \end{aligned} \tag{1}$$

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.

Observación

El subíndice β se utiliza para evitar cualquier ambigüedad respecto a la base empleada. A lo largo de este apunte, indicaremos la base de cada número mediante un subíndice, especificando así el sistema en el que está representado.

Como hemos visto, es posible utilizar diferentes bases para representar datos. Por ejemplo, un mismo valor puede expresarse en base decimal, binaria, octal o hexadecimal según convenga, como veremos a lo largo de este apunte. En particular, nos interesa comenzar por el sistema binario, ya que es el que se utiliza internamente en las computadoras. Para ello, analizaremos primero cómo se organizan los datos dentro de la computadora.

3. Sistema binario

Una de las principales ventajas de los sistemas posicionales es que permiten definir reglas generales y sencillas para las operaciones aritméticas. Además, dichas reglas tienden a simplificarse cuanto menor es la base del sistema. Esta observación nos lleva a considerar el sistema de base $\beta = 2$, o **sistema binario**, en el que solo existen los dígitos 0 y 1.

Sin embargo, existe también una razón fundamental de naturaleza física para su uso. En su nivel más básico, una computadora solo puede registrar dos estados: la presencia o ausencia de flujo eléctrico en una determinada parte del circuito. Estos dos estados pueden asociarse, convencionalmente, a los dígitos 1 (flujo de electricidad) y 0 (ausencia de flujo). Por medio de circuitos adecuados, una computadora puede entonces realizar conteo y operaciones aritméticas utilizando el sistema binario.

3.1. Bit

El sistema binario consta únicamente de los dígitos 0 y 1, denominados **bits** (del inglés *binary digits*). En notación binaria, los valores 0 y 1 tienen el mismo significado que en notación decimal:

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

El bit es la unidad mínima de información en un sistema binario. Dado que solo puede representar dos valores distintos (cero o uno), podría parecer que tiene poca utilidad. Sin embargo, puede emplearse para representar una gran variedad de conceptos: verdadero o falso, encendido o apagado, presente o ausente, entre otros. Todo depende de cómo se defina la estructura de datos. Por ejemplo, podemos establecer que un bit con valor cero represente "falso" y un bit con valor uno represente "verdadero".

3.2. Nibble

Un **nibble** es un conjunto de cuatro bits. No se trata de una estructura de datos particularmente destacada, salvo en dos casos relevantes: la representación de números en formato BCD (*binary coded decimal*) y en formato hexadecimal[2].

Con un nibble se pueden representar $2^4 = 16$ valores distintos. En el caso del sistema hexadecimal, estos valores son: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. En formato BCD se utilizan solo diez dígitos (del 0 al 9), pero también se requiere un conjunto de cuatro bits para codificarlos.

En resumen, un nibble (4 bits) permite representar un dígito en formato BCD o un dígito en formato hexadecimal.

3.3. Byte

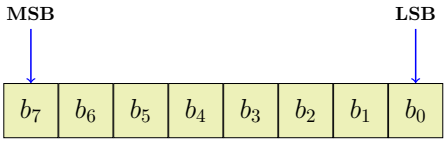
Un **byte** es una unidad de información digital compuesta por 8 bits. Es la unidad básica de almacenamiento en la mayoría de las arquitecturas de computadoras y se utiliza para representar caracteres, números y otros tipos de datos. La memoria principal está direccionada por bytes, lo que significa que el elemento más pequeño al que se puede acceder de forma individual es un valor de 8 bits.

Para acceder a una porción menor de información —como un solo bit o un conjunto de pocos bits— es necesario leer el byte completo que los

[2]El sistema hexadecimal se analiza en detalle en la Sección 4.1

contiene y aplicar una operación de enmascaramiento[3] para descartar los bits no deseados.

Los bits en un byte se pueden numerar desde cero hasta siete usando la siguiente convención:



El bit 0 (b_0) se denomina bit de menor orden o bit menos significativo (LSB, por la sigla en inglés *Least Significant Bit*), mientras que el bit 7 (b_7) es el bit de mayor orden o bit más significativo (MSB, por *Most Significant Bit*).

Cabe destacar que un byte contiene exactamente dos nibbles. Los bits del 0 al 3 conforman el *low-order nibble*, mientras que los bits del 4 al 7 forman el *high-order nibble*. Como cada nibble puede representarse con un dígito hexadecimal, se necesitan dos dígitos hexadecimales para representar un byte completo.

3.4. Palabra

En informática, una **palabra** (o *word*) es una unidad de datos cuyo tamaño está determinado por la arquitectura del procesador. Por lo general, una palabra puede tener 16, 32 o 64 bits, según la arquitectura. Se utiliza como la cantidad estándar de datos que una computadora puede procesar, leer o escribir de manera eficiente en una sola operación.

Una palabra puede definirse entonces como un conjunto de n bits que la máquina maneja como una unidad. Numeraremos sus bits desde el cero (b_0) hasta el bit $n-1$ (b_{n-1}). Al igual que en el caso del byte, el bit b_0 es el menos significativo, mientras que el bit b_{n-1} es el más significativo.

Por ejemplo, una palabra de 16 bits contiene dos bytes, y por lo tanto, cuatro nibbles, como se ilustra en la siguiente figura:

Con n bits se pueden representar 2^n valores diferentes. Por ejemplo, en las arquitecturas modernas que utilizan palabras de 64 bits, es posible representar hasta 2^{64} valores distintos.

Uno de los principales usos de las palabras es la representación de valores enteros. En el caso de valores sin signo (*unsigned*), la representación se realiza

[3]En informática, se denomina *"máscara"* al conjunto de datos que, junto con una operación determinada, permite extraer selectivamente ciertos bits almacenados en otro conjunto. Abordaremos este concepto con mayor detalle en el próximo apunte.

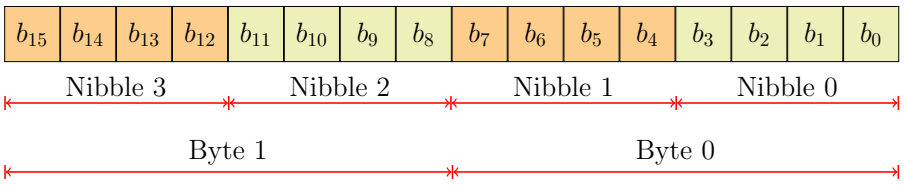


Figura 1: Bytes y nibbles en una palabra de 16 bits.

directamente a partir del valor binario de los bits. Para representar números con signo existen diferentes enfoques; uno de los más utilizados es el método del *complemento a dos*, que se abordará en la Sección 3.9.2.

3.5. Definición del sistema binario

Es posible establecer una equivalencia entre la representación binaria y la decimal a partir de la forma general del sistema de numeración posicional presentada en la Ecuación 1:

$$(b_n b_{n-1} \dots b_1 b_0 . b_{-1} \dots b_{-n})_2 = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + \dots + b_{-n} \times 2^{-n}.$$

Así, por ejemplo, $(1101.01)_2$ representa el número decimal $(13.25)_{10}$, ya que:

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

En la Tabla 1 se muestra como ejemplo la equivalencia entre la representación decimal y binaria utilizando números enteros de 4 bits. Vemos que pueden representarse números decimales en el rango de 0 a 15. De forma general, el rango de valores representables está dado por:

$$\boxed{0 \leq \text{valor} \leq 2^n - 1}$$

donde n es la cantidad de bits utilizados para representar el número entero.

Observación

En este apunte nos enfocaremos exclusivamente en la representación de números enteros. La representación computacional de números reales, que requiere herramientas conceptuales adicionales, será abordada en el próximo apunte.

Tabla 1: Representación de números sin signo en binario.

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

3.6. Conversión entre sistemas

3.6.1. Binario a decimal

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

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

Ejemplo

Supongamos que queremos convertir el número binario $(0110.1101)_2$ a decimal:

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

3.6.2. 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.

Ejemplo

Supongamos que queremos 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:

$$\begin{array}{ll} 149/2=74 & \text{Resto}=1 \rightarrow b_0=1 \\ 74/2=37 & \text{Resto}=0 \rightarrow b_1=0 \\ 37/2=18 & \text{Resto}=1 \rightarrow b_2=1 \\ 18/2=9 & \text{Resto}=0 \rightarrow b_3=0 \\ 9/2=4 & \text{Resto}=1 \rightarrow b_4=1 \\ 4/2=2 & \text{Resto}=0 \rightarrow b_5=0 \\ 2/2=1 & \text{Resto}=0 \rightarrow b_6=0 \\ 1/2=0 & \text{Resto}=1 \rightarrow b_7=1 \end{array}$$

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:

$$\begin{array}{ll} 0.56 \times 2 = \mathbf{1}.12 & \rightarrow b_{-1}=1 \\ 0.12 \times 2 = \mathbf{0}.24 & \rightarrow b_{-2}=0 \\ 0.24 \times 2 = \mathbf{0}.48 & \rightarrow b_{-3}=0 \\ 0.48 \times 2 = \mathbf{0}.96 & \rightarrow b_{-4}=0 \\ 0.96 \times 2 = \mathbf{1}.92 & \rightarrow b_{-5}=1 \\ 0.92 \times 2 = \mathbf{1}.84 & \rightarrow b_{-6}=1 \\ 0.84 \times 2 = \mathbf{1}.68 & \rightarrow b_{-7}=1 \\ 0.68 \times 2 = \mathbf{1}.36 & \rightarrow b_{-8}=1 \end{array}$$

De esta manera, $(0.56)_{10} = (1000\ 11111)_2$ empleando 8 bits. Por lo tanto, uniendo ambos resultados obtenemos la representación del número $(149.56)_{10}$:

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

En realidad en el ejemplo anterior 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 $(1001\ 0101.1000\ 11111)_2 = 149.5586$, 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 relativo en función del número de dígitos empleados para representar la parte fraccionaria:

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

donde t es la cantidad de dígitos empleados para la parte fraccionaria y β es la base empleada. En este ejemplo la cota de error es 1.953×10^{-3}.

3.7. Operaciones elementales en sistema binario

3.7.1. Suma

En binario, la suma de bits tiene la siguiente forma:

$$\begin{aligned} 0 + 0 &= \quad 0 \\ 0 + 1 &= \quad 1 \\ 1 + 0 &= \quad 1 \\ 1 + 1 &= \mathbf{1}0 \end{aligned}$$

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.

La siguiente tabla muestra la operación de suma binaria considerando dos operandos (a_i y b_i) y el acarreo de entrada de la posición anterior (c_{in}). Se presentan los resultados de la suma (s_i) y el acarreo de salida (c_{out}), que se propaga al siguiente bit en sumas de múltiples bits:

a_i	b_i	c_{in}	$a_i + b_i + c_{in}$	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ejemplo
Dados números $A = (0101)_2$ y $B = (0011)_2$, realizar la suma $A + B$:

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

En sistema decimal sería $A + B = 5 + 3 = 8$.

3.7.2. Resta

La tabla de la resta toma la siguiente forma:

$$\begin{aligned} 0 - 0 &= \quad 0 \\ 1 - 0 &= \quad 1 \\ 1 - 1 &= \quad 0 \\ 0 - 1 &= \mathbf{1}1 \end{aligned}$$

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.

La siguiente tabla muestra la operación de resta binaria considerando dos operandos (a_i y b_i) y un préstamo de entrada (c_{in}). Se presentan los resultados de la diferencia (d_i) y el préstamo de salida (c_{out}), que indica si es necesario tomar un préstamo en la siguiente posición en restas de múltiples bits.

a_i	b_i	c_{in}	$a_i - b_i - c_{in}$	c_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Ejemplo

Dados los números $A = (1001)_2$ y $B = (0010)_2$, realizar la resta $A - B$:

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

En sistema decimal sería $A - B = 9 - 2 = 7$. 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 3.9.2.

3.7.3. Multiplicación

La tabla de multiplicación binaria toma la siguiente forma:

$$\begin{aligned} 0 \times 0 &= 0 \\ 0 \times 1 &= 0 \\ 1 \times 0 &= 0 \\ 1 \times 1 &= 1 \end{aligned}$$

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

Dados los números $A = (1100)_2$ y $B = (0110)_2$, realizar la multiplicación $A \times B$:

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 0 \\
 & & & \times & 0 & 1 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & \mathbf{1} & 1 & 1 & 0 & 0 & \\
 & \mathbf{1} & 1 & 1 & 0 & 0 & & \\
 0 & 0 & 0 & 0 & & & & \\
\hline
 1 & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

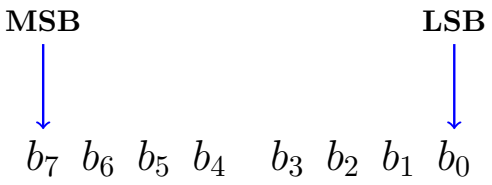
En sistema decimal sería $A \times B = 12 \times 6 = 72$.

3.8. Formatos binarios

Un número en formato binario puede representarse de distintas maneras, ya que los bits en cero a la izquierda del bit más significativo no aportan valor numérico. Por ejemplo, el número 6 puede escribirse como $(110)_2$, $(0110)_2$, $(00000110)_2$, entre otras variantes.

Sin embargo, es habitual emplear una notación en la que se omiten los ceros a la izquierda. Otra convención común consiste en agrupar los bits de cuatro en cuatro o de ocho en ocho. Para mejorar la legibilidad, se acostumbra a separar los grupos de 4 bits con un espacio en blanco, de manera análoga a la práctica decimal de usar puntos o comas para agrupar dígitos cada tres cifras. Por ejemplo, el número 6 puede representarse en binario como $(0000\ 0110)_2$.

Al trabajar con números binarios, es frecuente la necesidad de referirse a un bit en particular dentro de la representación. Para ello, como se mencionó anteriormente, se asigna un índice a cada posición del bit. En números binarios de 8 bits, las posiciones se numeran desde el bit 0 hasta el bit 7, de derecha a izquierda:



El bit más a la derecha es el de posición cero, y se lo denomina *bit menos significativo* (**LSB**, por *Least Significant Bit*). A medida que se avanza hacia

la izquierda, los bits corresponden a posiciones de mayor peso, siendo el bit más a la izquierda el *bit más significativo* (**MSB**, por *Most Significant Bit*).

De forma análoga, para representar números con parte fraccionaria, se utilizan posiciones con índice negativo a partir del punto binario (equivalente al punto decimal), comenzando por 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 de menor peso (el menos significativo) es b_{-8}, mientras que el más significativo continúa siendo b_7.

3.9. Números con signo

Hasta ahora hemos trabajado con números sin signo, donde todos son positivos. Sin embargo, también es necesario representar números negativos. Una opción natural es seguir la analogía con el sistema decimal, en el que un número negativo se expresa como su magnitud precedida por el signo "-". Este método se conoce como formato **magnitud y signo**.

3.9.1. Representación Magnitud y Signo

En el formato magnitud y signo, se utiliza un bit, generalmente el más significativo, para representar el signo, mientras que los bits restantes indican la magnitud (valor absoluto) del número:

$$(b_{n-1} b_{n-2} \cdots b_1 b_0)_2 = (-1)^s \times (b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0),$$

donde $s = b_{n-1}$. De este modo, si el bit más significativo es "uno", el número es 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: $(-28)_{10} = (\mathbf{1}001\ 1100)_2$.

$$(1001\ 1100)_2 = (-1)^1 \times (2^4 + 2^3 + 2^2) = (-28)_{10}$$

En la Tabla 2 se muestra la representación de números con signo utilizando números de 4 bits. El rango de valores representables es:

$$\boxed{-2^{n-1} + 1 \leq \text{rango} \leq 2^{n-1} - 1}$$

donde n es la cantidad de bits utilizados para representar el número.

El enfoque magnitud y signo presenta dos desventajas importantes. En primer lugar, existe una doble representación del cero, ya que tanto $(\mathbf{1}000)_2$

como $(\mathbf{0}000)_2$ representan el mismo valor, lo que complica una operación fundamental: la comparación con cero. Por otro lado, las operaciones aritméticas resultan más complejas. Por ejemplo, para realizar una suma, primero es necesario determinar si los números tienen el mismo signo y, en caso afirmativo, sumar sus magnitudes. Si los signos son diferentes, se debe restar el menor del mayor y asignar el signo del número con mayor valor absoluto.

Ejemplo

Supongamos que se utilizan 4 bits para representar números en magnitud y signo. Ahora sumemos $(+3) + (-3)$:

$$\begin{array}{rcccc|c} & 0 & 0 & 1 & 1 & (+3) \\ + & 1 & 0 & 1 & 1 & (-3) \\ \hline & 1 & 1 & 1 & 0 & (-6) \end{array} \quad \text{✗}$$

Esperábamos obtener cero, pero el resultado es -6, que es incorrecto.

Debido a estas limitaciones, este enfoque tuvo un uso limitado. A continuación, exploraremos alternativas más eficientes basadas en el ***complemento del número***.

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

Otro enfoque para representar números con signo se basa en los operadores complemento. Estos operadores se utilizan ampliamente para representar números negativos. De este modo, al realizar una resta, la operación se puede transformar en una suma de dos números positivos mediante la aplicación del operador complemento.

Existen dos operadores complemento: el **complemento a la base** y el **complemento a la base menos uno**. 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, y así sucesivamente.

Desde el punto de vista computacional, nos interesa especialmente el complemento a dos, ya que es la representación más utilizada para el manejo de números negativos en sistemas digitales. Esta técnica permite realizar operaciones aritméticas de manera eficiente, simplificando la implementación del hardware y optimizando el procesamiento de datos.

Tabla 2: Representación de números con signo en magnitud y signo.

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

3.9.3. Operador complemento a dos

El operador complemento a dos de un número N se define como:

$$\boxed{C_2(N) = 2^n - N}$$

donde n es la cantidad de bits que utilizamos para representar el número.

Ejemplo

Supongamos que tenemos el número $N = 84$. Su representación en binario utilizando 8 bits es $(01010100)_2$. Aplicando el operador complemento a dos según la definición de la ecuación anterior:

$$C_2(N) = (100000000)_2 - (01010100)_2 = (10101100)_2$$

Observación

Notar que $C_2\Big(C_2(N)\Big) = N$. Es decir, si a un número N le aplicamos el operador complemento a dos y al resultado le volvemos a aplicar el mismo operador obtenemos el número original.

Operador complemento a dos: método alternativo

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 $(1111)_2$ es $(0001)_2$, etc.

3.9.4. Representación en complemento a dos

La representación binaria en complemento a dos es un sistema posicional para números enteros en el que los positivos se expresan en binario natural, mientras que los negativos se obtienen aplicando el complemento a dos. Por lo tanto, para un N su representación en binario complemento a dos es la siguiente:

- **Representación de números positivos**: Los números positivos se representan de forma estándar en binario, igual que en el sistema binario sin signo. El bit más significativo (MSB) es 0, indicando que el número es positivo.
- **Representación de números negativos**: Para obtener la representación de un número negativo en complemento a dos, primero se toma el valor absoluto del número y se representa en binario. Luego, se aplica el operador complemento a dos al valor absoluto. El bit más significativo (MSB) es 1, indicando que el número es negativo.

Ejemplos

- Si tenemos el número $N_1 = (84)_{10}$ y lo representamos con 8 bits, dado que N_1 es un número positivo, utilizamos la representación en binario natural:

$$(84)_{10} = 2^6 + 2^4 + 2^2$$

Por lo tanto:

$$N_1 = (0101\ 0100)_2$$

- Si tenemos el número $N_2 = (-84)_{10}$ y lo representamos con 8 bits, dado que N_2 es un número negativo, aplicamos la definición de complemento a dos:

$$N_2 = \underbrace{(1\ 0000\ 0000)_2}_{2^8} - \underbrace{(0101\ 0100)_2}_{84} = (1010\ 1100)_2$$

Por lo tanto:

$$N_2 = (-84)_{10} = (1010\ 1100)_2$$

- Si queremos hacer la operación $N_1 + N_2 = 84 + (-84) = 0$, resulta

$$(0101\ 0100)_2 + (1010\ 1100)_2 = (\mathbf{1}\ 0000\ 0000)_2$$

El resultado es correcto si despreciamos el acarreo.

Aquí vemos la gran ventaja del sistema de representación en complemento a dos.

Observaciones

- Notar que, al trabajar en binario complemento a dos (es decir, con números con signo), si el bit más significativo es uno, el número es negativo; mientras que si es cero, el número es positivo:

$$\begin{aligned} N_1 = &\quad (84)_{10} &&= (0101\ 0100)_2 \\ N_2 = &\quad (-84)_{10} &&= (1010\ 1100)_2 \end{aligned}$$

- La secuencia de bits del número $N_2 = (1010\ 1100)_2$ también se puede interpretar como $N_2 = -2^7 + 2^5 + 2^3 + 2^2 = -84$.

En efecto, un número binario de n bits en complemento a dos se puede interpretar de la siguiente manera:

$$N = (b_{n-1}b_{n-2}\ldots b_1b_0)_2 = -b_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

donde:

- b_{n-1} = bit más significativo (MSB).
- b_i = bit en la posición i (contando desde la derecha, empezando en 0).

La diferencia con binario sin signo es que el MSB tiene peso negativo.

De la primera observación se obtiene el método para convertir un número binario en complemento a dos a su equivalente decimal:

1. Si el bit más significativo (MSB) es 0, el número es positivo y su valor en decimal se obtiene directamente convirtiendo el número binario a decimal.

2. Si el bit más significativo (MSB) es 1, el número es negativo y se debe aplicar el complemento a dos para obtener su valor absoluto. Es decir, aplicar el complemento a dos y agregar el signo negativo.

De la segunda observación surge una forma alternativa de convertir un número binario en complemento a dos a su equivalente decimal.

En la Tabla 3 se muestra la representación de números con signo utilizando enteros de 4 bits.

El rango de valores decimales utilizando el complemento a dos para n bits es:

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

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 abarca todos los números enteros desde -8 hasta 7, como se muestra en la siguiente figura:

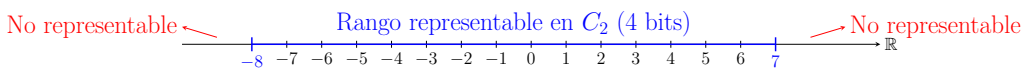


Tabla 3: Representación de números con signo en complemento a dos.

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

Si necesitamos representar un rango mayor debemos aumentar la cantidad de bits. La cuestión del rango es crucial al momento de realizar operaciones, dado que el resultado pude quedar fuera de rango, dando un resultado incorrecto. Esto lo veremos en detalle en la Sección 3.11.

Observación

Es importante notar que dada una secuencia de n bits, esa secuencia de bits puede ser interpretada como un número sin signo en el rango $[0 : 2^n - 1]$ utilizando la definición de binario natural o un número con signo en el rango $[-2^{n-1} : 2^{n-1} - 1]$ utilizando la representación en **complemento a dos**. La siguiente tabla muestra esta diferencia en la interpretación para números con 8 bits de acuerdo lo visto previamente.

Representación binaria	Números *unsigned*	Números *signed* complemento a dos
0000 0000	0	0
0000 0001	1	+1
⋮ ⋮	⋮ ⋮	⋮ ⋮
0111 1111	127	+127
1000 0000	128	-128
⋮ ⋮	⋮ ⋮	⋮ ⋮
1111 1110	254	-2
1111 1111	255	-1

3.9.5. Operador complemento a uno

El complemento a uno de un número N se define como:

$$\boxed{C_1(N) = 2^n - N - 1}$$

donde n es la cantidad de bits que se utiliza para la representación del número.

Ejemplo

Supongamos que tenemos el número $N = 84$. Su representación en binario es utilizando 8 bits es $(01010100)_2$. Aplicando el operador complemento a uno según la definición de la ecuación anterior:

$$C_1(N) = (100000000)_2 - (01010100)_2 - 1 = (10101011)_2$$

Operador complemento a uno: método alternativo

Observando el ejemplo anterior, vemos que una forma sencilla de hallar el complemento a uno de un número binario es invirtiendo todos sus dígitos. Este método es fácil de implementar mediante puertas lógicas.

Observaciones

- Notar que $C_1\Big(C_1(N)\Big) = N$. Es decir, si a un número N le aplicamos el operador complemento a uno y al resultado le volvemos a aplicar el mismo operador obtenemos el número original.
- Notar que el complemento a dos se puede obtener también tomando el complemento a uno de un número y luego sumando 1 al resultado: $C_2(N) = C_1(N) + 1$.

3.9.6. Representación en complemento a uno

La representación binaria en complemento a uno es un sistema de representación posicional para números enteros en el que los positivos se representan en binario natural y los negativos se representan utilizando el operador complemento a uno. Por lo tanto, para un N su representación en binario complemento a uno es la siguiente:

- **Representación de números positivos**: Los números positivos se representan de forma estándar en binario, igual que en el sistema binario sin signo. El bit más significativo (MSB) es 0, indicando que el número es positivo.
- **Representación de números negativos**: Para obtener la representación de un número negativo en complemento a uno, primero se toma el valor absoluto del número y se representa en binario. Luego, se aplica el operador complemento a uno al valor absoluto. El bit más significativo (MSB) es 1, indicando que el número es negativo.

Ejemplos

- Si tenemos el número $N_1 = (84)_{10}$ y lo representamos con 8 bits, dado que N_1 es un número positivo, utilizamos la representación en binario natural:

  $$(84)_{10} = 2^6 + 2^4 + 2^2$$

  Por lo tanto:

  $$N_1 = (0101\ 0100)_2$$

- Si tenemos el número $N_2 = (-84)_{10}$ y lo representamos con 8 bits, dado que N_2 es un número negativo, aplicamos la definición de complemento a uno:

$$N_2 = \underbrace{(1\ 0000\ 0000)_2}_{2^8} - \underbrace{(0101\ 0100)_2}_{84} - 1 = (1010\ 1011)_2$$

Por lo tanto:

$$N_2 = (-84)_{10} = (1010\ 1011)_2$$

- Si queremos hacer la operación $N_1 + N_2 = 84 + (-84) = 0$, resulta

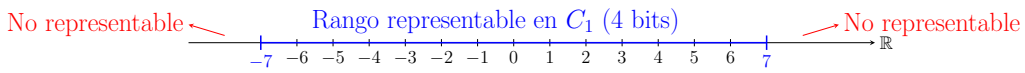
$$(0101\ 0100)_2 + (1010\ 1011)_2 = (1111\ 1111)_2$$

¿El resultado es correcto? Sí, en complemento a uno el resultado es correcto.

El rango de valores decimales utilizando el complemento a uno para n bits será:

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

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 abarca todos los números enteros desde -7 hasta 7, como se muestra en la siguiente figura:



En la Tabla 4 se presenta, de manera comparativa, la representación de números con signo utilizando enteros de 4 bits para los tres sistemas de representación estudiados. Observar que tanto la representación en complemento a uno como la representación en magnitud y signo tienen dos representaciones posibles del cero, lo cual es una desventaja. Notar también que el bit más significativo indica el signo de la representación del número en decimal con signo en cualquiera de las representaciones.

Observaciones

Tabla 4: Representación de números con signo con diferentes enfoques.

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

- Se puede observar que para conseguir el complemento a uno 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.
- Se observa además que hay doble representación del cero. Es decir, tanto $(0\cdots00)_2$ como $(1\cdots11)_2$ representan al cero.
- Las operaciones con números en complemento uno tienen inconvenientes. Supongamos que queremos hacer la operación $(+3)+(-2)$ en complemento a uno utilizando 4 bits:

$$\begin{array}{rcccc|c} & 0 & 0 & 1 & 1 & (+3) \\ + & 1 & 1 & 0 & 1 & (\text{-}2) \\ \hline \mathbf{1} & 0 & 0 & 0 & 0 & (0) \end{array} \quad \text{✗}$$

  Esperábamos obtener uno, pero el resultado es cero, que es incorrecto. Notemos que, si le sumamos el uno correspondiente al acarreo, obtenemos el resultado correcto:

$$\begin{array}{rcccc|c} & 0 & 0 & 0 & 0 & (0) \\ + & 0 & 0 & 0 & 1 & (1) \\ \hline & 0 & 0 & 0 & 1 & (1) \end{array} \checkmark$$

- Estas dos últimas observaciones explican por qué se prefiere el complemento a dos en lugar del complemento a uno.

3.10. Operaciones en complemento a dos

3.10.1. Suma

Las operaciones en complemento a dos permiten realizar sumas y restas de números con signo de manera eficiente en sistemas digitales. A continuación, se presentan algunos ejemplos fundamentales para operar en complemento a dos.

Ejemplos

Sean dos números A y B representados en complemento en binario complemento a dos con 8 bits. Si deseamos obtener la suma $S = A + B$, analizaremos los distintos casos posibles.

1. **A** $>$ **0** y **B** $>$ **0**:

   $A = 5$, $B = 12$

   En binario complemento a dos:

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

   $S = A + B = 17$ y acarreo $\mathbf{c} = 0$.

2. **A** $<$ **0** y **B** $<$ **0**:

   $A = -5$, $B = -12$

   En binario complemento a dos:

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

$S = A + B = -17$ y acarreo $\mathbf{c} = 1$.

3. $\mathbf{A < 0}$, $\mathbf{B > 0}$ y $|\mathbf{A}| < |\mathbf{B}|$:

   $A = -5$, $B = 12$

   En binario complemento a dos:

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

   $S = A + B = 7$ y acarreo $\mathbf{c} = 1$.

4. $\mathbf{A > 0}$, $\mathbf{B < 0}$ y $|\mathbf{A}| < |\mathbf{B}|$:

   $A = 5$, $B = -12$

   En binario complemento a dos:

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

   $S = A + B = -7$ y acarreo $\mathbf{c} = 0$.

En todos los casos, el resultado es correcto si se ignora el acarreo.

3.10.2. Resta

La operación de resta en complemento a dos se puede presentar como una suma, evitando la necesidad de un circuito específico para la sustracción. Para realizar la operación resta $A - B$ se sigue el siguiente procedimiento:

- Obtener el complemento a dos de B.
- Sumar el complemento a dos de B a A.

Este método unifica las operaciones aritméticas, permitiendo que la resta se implemente con el mismo circuito que la suma. A continuación, se presentan un ejemplo ilustrativo para comprender su aplicación.

Ejemplo
Realizar la operación $84 - 37$ en complemento a dos.

- Representar los números en binario con 8 bits:
  - $(84)_{10} = (0101\ 0100)_2$
  - $(37)_{10} = (0010\ 0101)_2$
- Obtener el complemento a dos de 37:

$$C_2(-37) = (1101\ 1011)_2$$

- Sumar 84 y -37 en binario:

$$(0101\ 0100)_2 + (1101\ 1011)_2 = (1\ 0011\ 1111)_2$$

- Descartar el acarreo más significativo.

El resultado es $(0011\ 1111)_2$, que en decimal equivale a 47, el resultado esperado de $84 - 37$.

Este ejemplo muestra cómo la resta se convierte en una suma utilizando el complemento a dos, permitiendo una implementación eficiente en hardware.

Observaciones

- El complemento a dos es el sistema utilizado en computadoras modernas para la representación de los números con signo, debido a las ventajas al realizar operaciones como hemos visto en los ejemplo anteriores.
- Es importante notar que lo anterior es cierto siempre en cuando nos mantengamos dentro del rango de números representables para la cantidad de dígitos con los que estamos trabajando. Sin embargo, al realizar una operación es posible que nos salgamos de rango, con lo cual el resultado obtenido es erróneo. Por lo tanto, debemos tener en cuenta las banderas de acarreo y desbordamiento como veremos a continuación.

3.11. Banderas

Al operar con valores en representación computacional, es fundamental analizar el estado de las banderas. Una **bandera** (*flag* en inglés) es un bit especial que se utiliza para indicar el resultado de ciertas operaciones o para controlar el comportamiento del procesador. Las banderas forman parte de un registro denominado **registro de estado** o **registro de banderas** (*status register* o *flags register*), el cual es actualizado automáticamente por el procesador tras la ejecución de instrucciones aritméticas, lógicas, de comparación, entre otras.

El registro de banderas[4] contiene diversas banderas, entre ellas la *carry flag* (CF) y la *overflow flag* (OF). Es importante considerarlas —y no confundirlas—, ya que la ALU (*Arithmetic Logic Unit*) no distingue si el programador está trabajando con aritmética con signo (*signed*) o sin signo (*unsigned*). La ALU simplemente realiza la operación binaria bit a bit y actualiza las banderas correspondientes. Corresponde al programador interpretar el valor de estas banderas luego de la operación. A continuación, analizaremos las dos banderas principales del registro de banderas, que inciden directamente en la ejecución de operaciones aritméticas.

3.11.1. *Carry Flag*

La bandera de **acarreo** (*Carry Flag*, CF) se activa cuando ocurre un acarreo en una suma o un préstamo en una resta, lo que indica que se ha excedido el rango representable para números sin signo.

Ejemplos

A continuación se presentan ejemplos que trabajan con números sin signo representados con 8 bits:

- Supongamos que realizamos la suma $255 + 1$:

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

[4] En arquitecturas x86-64, el registro `rflags` almacena indicadores de estado y control del procesador. Contiene bits que reflejan el resultado de operaciones aritméticas (como el *carry* o el *zero flag*) y configuraciones que afectan el comportamiento de ejecución. El registro `rflags` se estudiará en detalle en el apunte correspondiente a *Assembler x86-64*.

La bandera *carry* queda en `CF=1` debido a que se produjo un acarreo al sumar los bits más significativos. El resultado es incorrecto si se interpretan los operandos como números *unsigned*, ya que $255 + 1 \neq 0$. Sin embargo, el resultado es correcto si se los interpreta como números *signed*, dado que $-1 + 1 = 0$.

- Supongamos que realizamos la resta $0 - 1$:

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

La bandera *carry* queda en `CF=1` porque se produjo un préstamo en la última posición. El resultado es incorrecto si se interpretan los operandos como números *unsigned*, ya que $0 - 1 \neq 255$. Sin embargo, es correcto si se los interpreta como números *signed*, dado que $0 - 1 = -1$.

Conclusión: Si la suma de dos números enteros en binario genera un acarreo en el bit más significativo, o si la resta requiere un préstamo en ese mismo bit, se activa la bandera *carry* (`CF=1`). Esto indica que el resultado es incorrecto si se interpretan los operandos como números *signed*, ya que queda fuera del rango representable para números con signo con la cantidad de bits disponible. En consecuencia, sería necesario utilizar más bits para almacenar el resultado correctamente.

3.11.2. *Overflow Flag*

La bandera de **desbordamiento** (*Overflow Flag*, OF) se activa cuando el resultado de una operación aritmética excede el rango representable para enteros con signo.

Ejemplos

A continuación se presentan ejemplos que trabajan con números con signo representados con 8 bits:

- Supongamos que se realiza la suma $64 + 64$:

$$\begin{array}{cccccccccc} & & \mathbf{1} & & & & & & & \\ & 0 & 1 & 0 & 0 & & 0 & 0 & 0 & 0 \\ + & 0 & 1 & 0 & 0 & & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 \end{array} \quad ✗$$

El resultado es incorrecto si se interpretan los operandos como números *signed*, ya que $64 + 64 \neq -128$; es decir, se sumaron dos números positivos y el resultado fue negativo. Sin embargo, el resultado es correcto si se interpretan como números *unsigned*, dado que $64 + 64 = 128$. En este ejemplo, las banderas quedan en `CF=0` y `OF=1`.

- Supongamos que se realiza la suma $(-128) + (-128)$:

$$\begin{array}{cccccccccc} & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ + & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \quad ✗$$

El resultado es incorrecto, ya que $-128 + (-128) \neq 0$. El resultado también es incorrecto si se interpretan como números *unsigned*, dado que $128 + 128 \neq 0$. En este ejemplo, las banderas quedan en `CF=1` y `OF=1`.

- Supongamos que se realiza la resta $(-96) - 64$:

$$\begin{array}{cccccccccc} & \mathbf{1} & & & & & & & & \\ & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \\ - & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \\ \hline & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \end{array} \quad ✗$$

El resultado es incorrecto, ya que $(-96) - 64 \neq 96$. Sin embargo, el resultado es correcto si se interpretan como números *unsigned*, dado que $160 - 64 = 96$. En este ejemplo, las banderas quedan en `CF=0` y `OF=1`.

Esta resta también puede expresarse como una suma utilizando el complemento a dos del sustraendo:

$$\begin{array}{cccccccccc} & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ + & & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & \mathbf{1} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \quad ✗$$

Notar que, en este caso, `CF=1` porque hubo un acarreo.

Conclusión: Si la suma de dos números cuyo bit más significativo es cero da como resultado un número con el bit de signo en uno, o si la suma de dos números cuyo bit más significativo es uno da como resultado un número con el bit de signo en cero, la bandera overflow se activa. En resumen, al operar con números con signo, la bandera overflow indica si el resultado es correcto o no, mientras que la bandera carry no tiene relevancia.

4. Otras representaciones

Además del sistema binario, existen otros dos sistemas posicionales de interés en el ámbito computacional: el **sistema octal** (base 8) y el **sistema hexadecimal** (base 16).

4.1. Sistema hexadecimal

Un inconveniente importante del sistema binario es su "*verbosidad*", es decir, la gran cantidad de dígitos necesarios para representar un valor. Por ejemplo, para representar $(158)_{10} = (1001\ 1110)_2$ se requieren ocho cifras, en contraste con las tres necesarias en el sistema decimal. Esto sugiere que la representación decimal es más compacta. Sin embargo, implementar un microprocesador que opere directamente en base decimal sería considerablemente más complejo. Además, si bien es posible convertir entre decimal y binario, el proceso no resulta directa (como se analizó en la Sección 3.6).

El **sistema hexadecimal** (base 16) ofrece una solución eficiente a este problema. Presenta dos ventajas clave:

- Su notación es más compacta.
- La conversión con el sistema binario es directa.

Por estas razones, el sistema hexadecimal se utiliza ampliamente en el ámbito de la computación.

Dado que en el sistema 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 su posición. Cada dígito hexadecimal puede tomar uno de 16 valores distintos, que van de 0 hasta $(15)_{10}$. Como el sistema decimal solo cuenta con 10 dígitos, se utilizan seis símbolos adicionales para representar los valores del $(10)_{10}$ al $(15)_{10}$. Estos símbolos son las letras del abecedario de la A a la F. Así se obtiene la tabla de equivalencias entre sistemas, presentada en la Tabla 5.

De esta manera, el número $(15\text{E})_{16}$ representa:

$$(15\text{E})_{16} = 1 \times 16^2 + 5 \times 16^1 + 14 \times 16^0 = (350)_{10} = (0001\ 0101\ 1110)_2,$$

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

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 5 contiene toda la información necesaria. Para convertir de hexadecimal a binario simplemente hay que sustituir cada dígito hexadecimal por sus correspondientes cuatro bits.

Tabla 5: Equivalencia entre diferentes sistemas de numeración.

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

Ejemplo

Para convertir el número $(8AF3)_{16}$ a binario:

HEXADECIMAL	8	A	F	3
BINARIO	1000	1010	1111	0011

Por lo tanto, $(8AF3)_{16} = (1000\ 1010\ 1111\ 0011)_2$.

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 en la Tabla 5.

Ejemplo

Convertir el número binario $(0001\ 0011\ 0010\ 1110)_2$ en hexadecimal:

BINARIO	0001	0011	0010	1110
HEXADECIMAL	1	3	2	E

Por lo tanto, $(0001\,0011\,0010\,1110)_2 = (132E)_{16}$.

4.2. 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.

4.2.1. Suma

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

Ejemplo

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

4.2.2. Resta

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

Ejemplo
$R = N_1 - N_2$, con $N_1 = (1\mathrm{A}6)_{16}$ y $N_2 = (1\mathrm{C})_{16}$.

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

Otra forma 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_2. 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 $C_{15}(N)$:

$$\begin{aligned} C_{16}(N) &= C_{15}(N) + 1 \\ C_{16}(N) &= (\mathrm{FE3})_{16} + 1 = (\mathrm{FE4})_{16} \end{aligned}$$

Entonces,

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

El resultado es correcto ignorando el primer uno.

Observación

Si bien hemos mantenido la notación $()_{16}$ para mantener la coherencia con lo anterior, en realidad la notación más usual para el formato hexadecimal consiste en poner `0x` antes de los dígitos hexadecimales. Por ejemplo, $(\texttt{ffe8})_{16}$ =`0xffe8`.

Observación

Si bien el sistema hexadecimal resulta más práctico que el binario para representar e ingresar información en sistemas computacionales —por su notación más compacta y legible para personas—, el sistema binario es más adecuado para las operaciones internas de la computadora. De hecho, todas las computadoras modernas operan en binario a nivel de hardware, ya que los circuitos electrónicos trabajan naturalmente con dos estados (ausencia o presencia de voltaje), lo que se corresponde con los valores 0 y 1.

4.3. Sistema octal

De manera análoga, se puede trabajar con otras bases. Por ejemplo, otro sistema de representación muy utilizado es el **sistema octal**, cuya base es 8. Este sistema utiliza ocho cifras: los números enteros del 0 al 7. En la Tabla 5 se muestra la equivalencia entre sistemas. La conversión puede realizarse de forma similar a la ya vista. Cabe destacar que cada dígito en sistema octal puede representarse exactamente con tres bits.

Ejemplo

Convertir el número $(1514)_8$ en formato octal a binario:

OCTAL	1	5	1	4
BINARIO	001	101	001	100

Entonces, $(1514)_8 = (0011\,0100\,1100)_2$

Observación

La notación más usual para el formato octal consiste en poner un `0` antes de los dígitos octales. Por ejemplo, $(777)_8$ =`0777`.

El sistema octal fue especialmente útil en las primeras etapas de la computación, cuando se trabajaba con una menor cantidad de bits, y aún se utiliza en algunos casos puntuales, como en los sistemas de archivos tipo UNIX.

5. Representación de caracteres y cadenas

Además de los datos numéricos, también es necesario almacenar datos simbólicos. Estos datos, también llamados no numéricos, permiten representar mensajes como "Hola mundo". Si bien dichos símbolos son fácilmente comprensibles para los hablantes de un determinado idioma, la memoria de la computadora está diseñada para almacenar y procesar números. Por lo tanto, cada símbolo se representa asignándole un valor numérico a cada carácter.

5.1. Representación de caracteres

En una computadora, un caracter es una unidad de información que corresponde a un símbolo tal como una letra en el alfabeto. Los caracteres incluyen letras, dígitos numéricos, signos de puntuación comunes (como "." o "!") y espacios en blanco. El concepto general también incluye caracteres de control, que no corresponden a símbolos en un idioma en particular, sino a otra información utilizada para procesar texto. Ejemplos de caracteres de control incluyen el retorno de carro o la tabulación.

Los caracteres se pueden representar mediante el Código ASCII (acrónimo inglés de *American Standard Code for Information Interchange* — Código Estándar estadounidense para el Intercambio de Información). Según la tabla ASCII[5], a cada carácter y carácter de control se le asigna un valor numérico. Cuando se utiliza ASCII, el carácter que se muestra se basa en el valor numérico asignado. Esto solo funciona si todos están de acuerdo con los valores comunes, que es el propósito de la tabla ASCII. Por ejemplo, la letra

[5]Consulte https://www.asciitable.com/ para ver una tabla ASCII completa.

"A" se define como $(65)_{10}$. Este número se almacena en la memoria de la computadora y, cuando se muestra en la consola, se muestra la letra "A". Existen otros estándares ampliamente utilizados, por ejemplo el estándar UNICODE[6].

Los símbolos numéricos también se pueden representar en ASCII. Por ejemplo, "9" se representa como $(57)_{10}$ en la memoria de la computadora. El "9" se puede mostrar como salida a la consola. Si se envía a la consola, el valor entero $(9)_{10}$ se interpretaría como un valor ASCII que, en este caso, sería una tabulación. Es muy importante comprender la diferencia entre caracteres (como "2") y números enteros (como $(2)_{10}$). Los caracteres se pueden mostrar en la consola, pero no se pueden usar para cálculos. Los números enteros se pueden utilizar para los cálculos, pero no se pueden mostrar en la consola (sin cambiar la representación). Normalmente, un carácter se almacena en un byte (8 bits) de espacio. Esto funciona bien ya que la memoria es direccionable por bytes.

5.2. Representación de cadenas

Una cadena es una serie de caracteres ASCII, normalmente terminados en NULL. El NULL es un carácter de control ASCII no imprimible. Dado que no es imprimible, se puede utilizar para marcar el final de una cadena. Por ejemplo, la cadena "Hello" se representaría de la siguiente manera:

Caracter	"H"	"e"	"l"	"l"	"o"	NULL
Valor ASCII (decimal)	72	101	108	108	111	0
Valor ASCII (hexadecimal)	0x48	0x65	0x6C	0x6C	0x6F	0x0

Una cadena puede consistir parcial o completamente en símbolos numéricos. Por ejemplo, la cadena "19653" se representaría de la siguiente manera:

Caracter	"1"	"9"	"6"	"5"	"3"	NULL
Valor ASCII (decimal)	49	57	54	53	51	0
Valor ASCII (hexadecimal)	0x31	0x39	0x36	0x35	0x33	0x0

Nuevamente, es muy importante comprender la diferencia entre la cadena "19653" (que necesita 6 bytes para ser almacenada) y el entero 19653_{10} (que se puede almacenar con 2 bytes).

[6]Para mayor información, consultar: http://en.wikipedia.org/wiki/Unicode.

A. Apéndice: Representación de valores en C

El lenguaje C no proporciona un tipo de dato específico para trabajar con bits. Sin embargo, es posible operar a nivel de bits utilizando datos de tipo entero, considerando su representación interna. Por ejemplo, para expresar el valor `01000001b`, podemos declarar una variable de tipo `char` y escribir el siguiente código en C:

```
#include <stdio.h>
int main() {
    char A = 65;              // Decimal
    char B = 0x41;            // Hexadecimal
    char C = 0101;            // Octal
    char D = 0b01000001;      // Binario (requiere C23)
    char E = 'A';             // Caracter
    printf("%d %d %d %d %d\n", A, B, C, D, E);
    return 0;
}
```

Todas estas expresiones son equivalentes, ya que representan la misma secuencia de bits en memoria.

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.

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

[5] Ed Jorgensen, *x86-64 Assembly Language Programming with Ubuntu*, 2020.