Representación Computacional de Datos

Diego Feroldi
feroldi@fceia.unr.edu.ar

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

*Actualizado 25 de agosto de 2023 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

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 binarios en papel podemos utilizar un número arbitrario de bits. 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*[1], 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, *Double word* el doble que una palabra y *Quad word* 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 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

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

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.

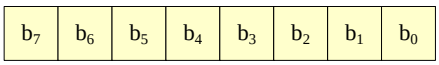
2.2. Nibble

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 BCD o un dígito hexadecimal.

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[2]" los bits no deseados.

Los bits en un byte se pueden numerar 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.

Notar que un byte contiene exactamente dos nibbles. Los bits del $0 \ldots 3$ 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.

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

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:

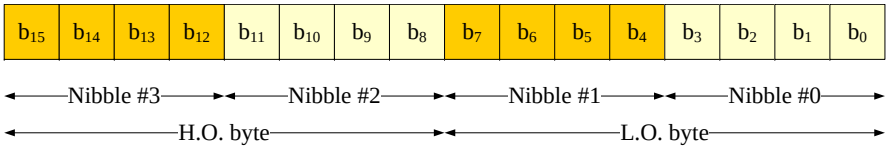


Figura 1: Bytes y nibbles en una palabra de 16 bits.

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

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)[3], 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

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

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

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, como en el caso decimal, 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:

- $(1001)_2$, sistema binario
- $(159.23)_{10}$, sistema decimal
- $(0A45)_{16}$, 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 *binary digits*). 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 de 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

4.2.1. Binario a decimal

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

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

Ejemplo

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

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

4.2.2. 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:*

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

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:

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

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 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 en función del número de dígitos empleados para representar la parte fraccionaria, es decir:

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

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

4.3. Operaciones elementales en sistema binario

4.3.1. Suma

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

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=10$$

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 $A=(0101)_2$ *y* $B=(0011)_2$*:*

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

En sistema decimal sería $A+B=5+3=8$.

4.3.2. Resta

La tabla de resta toma la siguiente forma:

$$0-0=0$$
$$1-0=1$$
$$1-1=0$$
$$0-1=\mathbf{1}$$

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 $B=0010_2$ *a* $A=1001_2$*:*

$$\begin{array}{cccc} \boldsymbol{1} & \boldsymbol{1} & & \\ 1 & 0 & 0 & 1 \\ - \quad 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 1 \end{array}$$

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 4.5.*

4.3.3. 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 lemento neutro de la multiplicación.

Ejemplo
Multiplicar $A = (1100)_2$ por $B = (0110)_2$:

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

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

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 en formato magnitud y signo 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 operadores complementos son muy usados en los sistemas digitales para representar números negativos y, por lo tanto, al momento de realizar operaciones de resta, transformando la resta en una suma de dos números positivos.

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, etc.

4.5.1. Operador complemento a la base

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

$$\boxed{C_\beta(N) = \beta^n - |N|}$$

Esto se cumple para todo N 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

Utilizando 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.

Observaciones

- *Supongamos que queremos hacer la operación* $987 - 987 = 0$. *Podemos hacer* $987 + (-987) = 0$. *Para ello utilizamos el operador complemento a la base para obtener* -987. *Aplicando la definición:* $C_{10}(N) = 1000 - 987 = 13$. *Entonces* $987 + (-987) = 987 + 13 = 1000$. *¿El resultados es correcto? Sí, es correcto si tenemos en cuenta que estamos trabajando con tres dígitos. Entonces, despreciando el uno tenemos el resultado correcto.*
- *Notar que* $C_\beta\Big(C_\beta(N)\Big) = N$. *Es decir, si a un número* N *le aplicamos el operador complemento y al resultado le volvemos a aplicar el mismo operador obtenemos el número original. Por ejemplo, si* $N = 987$ *con* $n = 3$, *tenemos que* $C_{10}(987) = 10^3 - 987 = 13$ *y* $C_{10}(13) = 10^3 - 13 = 987$.

4.5.2. Operador complemento a la base menos uno

Sea un número N representado con n dígitos en la parte entera y m dígitos en la parte fraccionaria, el operador complemento a la base menos uno de N se define como

$$\boxed{C_{\beta-1}(N) = \beta^n - \beta^{-m} - |N|}$$

Ejemplos

- *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 los resultados.

4.6. Representación binaria utilizando el operador complemento a dos

La representación binaria utilizando el operador complemento a dos es un sistema posicional de representación para números enteros en el que los positivos se representan en binario natural[4] y los negativos se representan utilizando el operador complemento a dos que se define como:

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

Ejemplos

- *Si tenemos el número* $N_1 = (84)_{10}$ *y lo representamos con 8 bits:*

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

[4]Nos referimos a binario natural como el binario resultante de aplicar la definición general de sistema posicional en base dos.

dado que al ser N_1 un número positivo lo representamos utilizando binario natural: $2^6 + 2^4 + 2^2 = (84)_{10}$.

- *Si tenemos el número* $N_2 = (-84)_{10}$ *y lo representamos con 8 bits:*

$$N_2 = (1010\ 1100)_2$$

dado que al ser N_2 un número negativo lo representamos aplicando la definición de complemento a dos:

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

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

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

El resultado es correcto si despreciamos el acarreo.

Observaciones

- *La secuencia de bits del número N_2 también se puede interpretar como* $N_2 = -2^7 + 2^5 + 2^3 + 2^2 = -84$.
- *Al trabajar en binario complemento a dos (es decir, con números con signos) si el bit más significativo es uno implica que el número es negativo.*

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.

Otro método alternativo

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.

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

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

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 $-8 \leq Rango \leq 7$.

4.7. 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 bits se define como:

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

Ejemplos

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

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

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

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

En estos 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.

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 es $-7 \leq Rango \leq 7$.

En la Tabla 1 se encuentra la representación de números con signo con enteros de 4 bits. 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.

Tabla 1: Representación de números con signo

Decimal con signo	**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	0111	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		

Observación

El complemento a dos es el sistema utilizado para la representación de los números con signo. Sin embargo, es importante notar que dada una secuen-

cia 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 el concepto de complemento a dos. La siguiente tabla muestra esta diferencia en la interpretación para números con 8 bits de acuerdo lo visto previamente.*

Número decimal sin signo	***Representación binaria***	***Número signado con signo***
0	*0000 0000*	*0*
1	*0000 0001*	*+1*
⋮ ⋮	⋮ ⋮	⋮ ⋮
127	*0111 1111*	*+127*
128	*1000 0000*	*-128*
⋮ ⋮	⋮ ⋮	⋮ ⋮
254	*1111 1110*	*-2*
255	*1111 1111*	*-1*

4.8. 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. Sin bien lo podemos aplicar para cualquier base, a continuación, veremos de forma particular el caso con base $\beta = 2$ mediante ejemplos.

Ejemplos

1. $A = 5$, $B = 12$, $S = A + B = 17$

   En binario:

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

y acarreo $\mathbf{c} = 0$

2. $A = -5$, $B = -12$, $S = A + B = -17$

   En binario:

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

   y acarreo $\mathbf{c} = 1$

3. $A = -5$, $B = 12$, $S = A + B = 7$

   En binario:

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

   y acarreo $\mathbf{c} = 1$

4. $A = 5$, $B = -12$, $S = A + B = -7$

   En binario:

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

   y acarreo $\mathbf{c} = 0$

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

Observación

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 CF y OF como veremos a continuación.

4.9. Banderas

Al momento de operar con valores en representación computacional es importante analizar el estado de las banderas en el registro `rflags`[5], en particular la *carry flag* (CF) y la *overflow flag* (OF). Es importante tenerlas en cuenta (y no confundirlas!) dado que la ALU (*Arithmetic Logic Unit*)[6] no tiene en cuenta si el programador está trabajando con matemática con signo (*Signed*) o sin signo (*Unsigned*). La ALU simplemente realiza la operación binaria bit a bit y setea las banderas apropiadamente. Depende del programador chequear el valor de las banderas una vez realizada la operación.

Si el programa trata los bits como números sin signo, debe verificarse si se ha seteado en "*1*" la bandera de *carry*, indicando que el resultado es erróneo. En este contexto, el valor de la bandera *overflow* es irrelevante. Por el contrario, si el programa trata a los bits como valores con signo en complemento a dos, lo que hay que verificar es la bandera *overflow* en lugar de la bandera *carry*. Veamos con ejemplos el uso de cada una de estas dos banderas.

4.9.1. *Carry Flag*

La bandera de acarreo (*Carry Flag*, CF) se "enciende" si una operación matemática con valores enteros sin signo genera un acarreo o un préstamo para el bit más significativo.

Ejemplos

- *Si la suma de dos números en binario entero causa un acarreo en el bit más significativo, la bandera carry se enciende.*

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

`CF=1`

El resultado es incorrecto dado que $15 + 1 \neq 0$.

[5] El registro `rflags` se estudiará en detalle en el apunte correspondiente a *Assembler X86-64*.

[6] La ALU es un circuito digital en el microprocesador que calcula operaciones aritméticas y operaciones lógicas, entre valores de los argumentos.

- *Si la resta de dos números requiere un acarreo en el bit más significativo, la bandera carry se enciende.*

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

`CF=1`

El resultado es incorrecto dado que $0 - 1 \neq 15$.

En conclusión, si se está realizando aritmética con números sin signo, la bandera *carry* encendida significa que el resultado es incorrecto, dado que el verdadero resultado es demasiado grande en valor absoluto para poder ser representado con el número de bits disponible.

4.9.2. *Overflow Flag*

La bandera de desbordamiento (*Overflow Flag*, OF) se "enciende" si una operación matemática con valores enteros con signo genera un resultado fuera del rango de representación.

Ejemplos

- *Si la suma de dos números con el bit más significativo en cero resulta en un número con el bit de signo en uno, la bandera overflow se enciende.*

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

Las banderas quedan `CF=0` *y* `OF=1`*. El resultado es incorrecto dado que interpretando los números en complento a dos es* $4 + 4 \neq -8$.

- *Si la suma de dos números con el bit más significativo en uno resulta en un número con el bit de signo en cero, la bandera overflow se enciende.*

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

Las banderas quedan `CF=1` *y* `OF=1`*. El resultado es incorrecto dado que* $-8 + (-8) \neq 0$.

Si se está realizando aritmética en complemento a dos, la bandera *overflow* encendida significa que el resultado es incorrecto. Por ejemplo, si se están sumando dos números positivos y el resultado es negativo o bien si se están sumando dos números negativos y el resultado es positivo. En resumen, si se está operando con números *signed*, la bandera *overflow* indica si el resultado es correcto o no.

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 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: la notación es muy compacta y es muy simple convertir entre hexadecimal y binario, y viceversa. Por estos motivos, en el ámbito de la computación se utiliza mucho el sistema de numeración hexadecimal.

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 $(15\text{E})_{16}$ representa a

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

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 2 contiene toda la información necesaria. Para convertir de hexadecimal

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

a binario simplemente hay que sustituir cada dígito hexadecimal por sus correspondientes cuatro bits.

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 en la Tabla 2.

Ejemplo
Para convertir el número binario $(0001\ 0011\ 0010\ 1110)_2$ en hexadecimal:

BINARIO	*0001*	*0011*	*0010*	*1110*
HEXADECIMAL	*1*	*3*	*2*	*E*

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

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

Resta

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

Ejemplo

$$\begin{array}{rccc} & & \boldsymbol{1} & \\ & 1 & A & 6 \\ - & & 1 & C \\ \hline & 1 & 8 & A \end{array}$$

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 $C_{15}(N)$:

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

Entonces,

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

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 consiste en poner `0x` *antes de los dígitos hexadecimales. Por ejemplo,* `0xffe8`.

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*

Observación

La notación más usual consiste en poner un `0` *antes de los dígitos octales. Por ejemplo,* `0777`.

6. Representación de caracteres y cadenas

Además de los datos numéricos, a menudo se requieren datos simbólicos. Los datos simbólicos o no numéricos pueden mostrar mensajes tales como "Hola mundo". Dichos símbolos son bien entendidos por los hablantes de un determinado lenguaje. Sin embargo, la memoria de la computadora está diseñada para almacenar y recuperar números. En consecuencia, los símbolos se representan asignando valores numéricos a cada símbolo o caracter.

6.1. 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[7], 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 "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[8].

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

[7] Consulte https://www.asciitable.com/ para ver una tabla ASCII completa.
[8] Para mayor información, consultar: http://en.wikipedia.org/wiki/Unicode.

(8 bits) de espacio. Esto funciona bien ya que la memoria es direccionable por bytes.

6.2. 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).

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.