Introducción a la Computación 2017

30 de abril de 2017

Índice

Sistemas de Numeración

Un sistema diferente

Todos conocemos el método tradicional de contar con los dedos. Como tenemos cinco dedos en cada mano, podemos contar hasta diez. Pero también podemos utilizar un método diferente del tradicional, que resulta ser muy interesante.

- Con este método, al llegar a 5 con la mano derecha, representamos el 6 **sólo con un dedo de la izquierda**. Los dedos de la mano derecha **vuelven a 0**, y seguimos contando con la derecha.
- Cada vez que se agotan los dedos de la mano derecha levantamos un nuevo dedo de la izquierda, y la derecha vuelve a 0.
- Cada dedo en alto de la mano izquierda significa que **se agotó la secuencia de la mano derecha una vez**.

Preguntas

- ¿Hasta qué número se puede representar en este sistema, sólo con dos manos?
- Si agregamos una tercera mano, de un amigo, ¿hasta qué número llegamos?
- ¿Y cómo se representa el 36? ¿Y el 37?
- Y con cuatro manos, ¿hasta qué número llegamos?
- Y, si el número no se puede representar con dos manos, ¿cómo es el procedimiento para saber qué dedos levantar?

Notemos que este método tiene mayor capacidad que el tradicional, ya que podemos contar hasta diez y todavía nos queda mucho por contar con los dedos de ambas manos.

Sistema posicional

Notemos además que esta ventaja se debe a que el método asigna **valores diferentes** a ambas manos. La derecha vale la cantidad de dedos que muestre, pero la izquierda vale **seis por su cantidad de dedos**. Esto se abrevia diciendo que se trata de un **sistema de numeración posicional**.

Al tratarse de un sistema posicional, podemos representar números relativamente grandes con pocos dígitos. En este sistema, disponemos únicamente de **6 dígitos (0, 1, 2, 3, 4, 5)** porque ésos son los que podemos representar con cada mano, es decir, **en cada posición**. Pero los números representables solamente dependen de cuántas manos (o, mejor dicho, de cuántas **posiciones**) podamos utilizar.

Calculando cada posición

En este sistema, dado un número no negativo que se pueda representar con dos manos, podemos saber qué dedos levantar en cada mano haciendo una sencilla cuenta de división entera (sin decimales): dividimos el número por 6 y tomamos el cociente y el resto. **El cociente es el número de la izquierda, y el resto, el de la derecha**.

Tomemos por ejemplo el número 15. Al dividir 15 por 6, el cociente es 2 y el resto es 3. En este sistema, escribimos el 15 como **dos dedos en la izquierda, y tres**

dedos en la derecha, lo que podemos abreviar como **(2,3)** o directamente **23** (que se pronuncia **dos tres** porque **no quiere decir veintitrés**, sino **quince**, sólo que escrito en este sistema). Como el dígito 2 de la izquierda vale por 6, si hacemos la operación de sumar $2 \times 6 + 3$ obtenemos, efectivamente, 15.

Base de un sistema de numeración

La **base** de un sistema es la cantidad de dígitos de que dispone, o sea que el sistema decimal habitual es de base 10, mientras que el de los deditos es de base 6.

Número y numeral

Notemos que un mismo número puede escribirse de muchas maneras: en prácticamente cualquier base que se nos ocurra, sin necesidad de contar con los dedos; y que la forma habitual, en base 10, no es más importante o mejor que las otras (salvo, claro, que ya estamos acostumbrados a ella). Otras culturas han desarrollado otros sistemas de numeración y escriben los números de otra manera.

Esto muestra que hay una **diferencia entre número y numeral**, diferencia que es algo difícil de ver debido a la costumbre de identificar a los números con su representación en decimal.

- El **numeral** es lo que escribimos (15, $15_{(10}$ o $23_{(6}$).
- El **número** es la cantidad de la cual estamos hablando (la misma en los tres casos).

Indicando la base

Anteriormente escribíamos **15** en el sistema de base 6 como **23**. Sin embargo, necesitamos evitar la confusión entre ambos significados de **23**. Para esto usamos índices subscriptos que indican la base. Así,

- **Quince** es $15_{(10}$ porque está en base diez (la del sistema decimal, habitual), y
- $23_{(10}$ es **veintitrés**,
- pero $23_{(6}$ es **dos tres en base 6**, y por lo tanto vale **quince**.

Como 10 es nuestra base habitual, cuando no usemos índice subscripto estaremos sobreentendiendo que hablamos **en base 10**. Es decir, $15_{(10}$ se puede escribir, simplemente, 15.

Cuando queremos pasar un número escrito en una base a un sistema con otra cantidad de dígitos, el procedimiento de averiguar los dígitos que van en cada posición se llama **conversión de base**. Más adelante veremos procedimientos de conversión de base para cualquier caso que aparezca.

Sistema decimal

Si reflexionamos sobre el sistema decimal, de diez dígitos, encontramos que también forma un sistema posicional, sólo que con 10 dígitos en lugar de los seis del sistema anterior.

Cuando escribimos **15** en el sistema decimal, esta expresión equivale a decir "para saber de qué cantidad estoy hablando, tome el 1 y multiplíquelo por 10, y luego sume el 5".

Si el número (o, mejor dicho, el **numeral**) tiene más dígitos, esos dígitos van multiplicados por **potencias de 10** que van creciendo hacia la izquierda. La cifra de las unidades está multiplicada por la potencia de 10 de exponente 0 (es decir, por 10^0, que es igual a 1).

Esto se cumple para todos los sistemas posicionales, sólo que con potencias **de la base correspondiente**.

Sistema binario

Comprender y manejar la notación en sistema binario es sumamente importante para el estudio de la computación. El sistema binario comprende únicamente dos dígitos, **0 y 1**.

Igual que ocurre con el sistema decimal, los numerales se escriben como suma de dígitos del sistema multiplicados por potencias de la base. En este sistema, cada 1 en una posición indica que sumamos una potencia de 2. Esa potencia de 2 es 2^n, donde n es la posición, y las posiciones se cuentan desde 0.

Por ejemplo,

$$10 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 1010_{(2}$$

y

$$15 = 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 = 1111_{(2}$$

Trucos para conversión rápida

Las computadoras digitales, tal como las conocemos hoy, almacenan todos sus datos en forma de números binarios. Es **muy recomendable**, para la práctica de esta materia, adquirir velocidad y seguridad en la conversión de y a sistema binario.

Una manera de facilitar esto es memorizar los valores de algunas potencias iniciales de la base 2:

$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^3 = 8$$
$$2^4 = 16$$
$$2^5 = 32$$
$$2^6 = 64$$
$$2^7 = 128$$

¿Qué utilidad tiene memorizar esta tabla? Que nos permite convertir mentalmente algunos casos simples de números en sistema decimal, a base 2. Por ejemplo, el número **40** equivale a **32 + 8**, que son ambas potencias de 2. Luego, la expresión de 40 en sistema binario será **101000**.

Otro truco interesante consiste en ver que si un numeral está en base 2, **multiplicarlo por 2 equivale a desplazar un lugar a la izquierda todos sus dígitos, completando con un 0 al final**. Así, si sabemos que $40_{(10} = 101000_{(2}$, ¿cómo escribimos rápidamente **80**, que es 40×2? Tomamos la expresión de 40 en base 2 y la desplazamos a la izquierda agregando un 0: $1010000_{(2} = 80_{(10}$.

De todas maneras, estos no son más que trucos que pueden servir en no todos los casos. Más adelante veremos el procedimiento de conversión general correcto.

Preguntas

- ¿Cuál es el truco para calcular rápidamente la expresión binaria de 20, si conocemos la de 40?
- ¿Cómo calculamos la de 40, si conocemos la de 10?
- ¿Cómo podemos expresar estas reglas en forma general?

Sistema hexadecimal

Otro sistema de numeración importante es el hexadecimal o de base 16. En este sistema tenemos **más dígitos** que en el decimal, por lo cual tenemos que recurrir a "dígitos" nuevos, tomados del alfabeto. Así, A representa el 10, B el 11, etc.

El sistema hexadecimal nos resultará útil porque con él podremos expresar fácilmente números que llevarían muchos dígitos en sistema binario.

- La conversión entre binarios y hexadecimales es sumamente directa.
- Al ser un sistema con más dígitos que el binario, la expresión de cualquier número será más corta.

Una expresión general

Como hemos visto intuitivamente en el sistema de contar con los dedos, y como hemos confirmado repasando los sistemas decimal, binario y hexadecimal, los sistemas posicionales tienen una cosa muy importante en común: las cifras de un **numeral** escrito en cualquier base no son otra cosa que los **factores por los cuales hay que multiplicar las sucesivas potencias de la base** para saber a qué **número** nos estamos refiriendo.

Por ejemplo, el numeral **2017** escrito en base 10 no es otra cosa que la suma de:

$$2 \times 1000 + 0 \times 100 + 1 \times 10 + 7 \times 1 =$$
$$2 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 7 \times 10^0$$

Los dígitos 2, 0, 1 y 7 multiplican, respectivamente, a 10^3, 10^2, 10^1 y 10^0, que son potencias de la base 10. Este **numeral** designa al **número** 2017 porque esta cuenta, efectivamente, da **2017**.

Sin embargo, si el número está expresado en otra base, la cuenta debe hacerse con potencias de esa otra base. Si hablamos de $2017_{(8}$, entonces las cifras 2, 0, 1 y 7 multiplican a 8^3, 8^2, 8^1 y 8^0. Este **numeral** designa al **número** 1039 porque esta cuenta, efectivamente, da **1039**.

Este análisis permite enunciar una ley o expresión general que indica cómo se escribe un número n cualquiera, no negativo, en una base b:

$$n = x_k \times b^k + \ldots + x_2 \times b^2 + x_1 \times b^1 + x_0 \times b^0$$

Esta ecuación puede escribirse más sintéticamente en notación de sumatoria como:

$$n = \sum_{i=0}^{k} x_i \times b^i$$

En estas ecuaciones (que son equivalentes):

- Los números x_i son las cifras del numeral.
- Los números b^i son potencias de la base, cuyos exponentes crecen de derecha a izquierda y comienzan por 0.
- Las potencias están **ordenadas y completas**, y son tantas como las cifras del numeral.
- Los números x_i son necesariamente **menores que** b, ya que son dígitos en un sistema de numeración que tiene b dígitos.

Conversión de base

Veremos algunos casos interesantes de conversiones de base. Serán especialmente importantes los casos donde el número de origen o de destino de la conversión esté en base 10, nuestro sistema habitual, pero también nos dedicaremos a algunas conversiones de base donde ninguna de ellas sea 10.

Conversión de base 10 a otras bases

El procedimiento para convertir un número escrito en base 10 a cualquier otra base (llamémosla **base destino**) es siempre el mismo y se basa en la división entera (sin decimales):

- Dividir el número original por la base destino, anotando cociente y resto
- Mientras se pueda seguir dividiendo:
    - Volver al paso anterior reemplazando el número original por el nuevo cociente
- Finalmente escribimos los dígitos de nuestro número convertido usando **el último cociente y todos los restos en orden inverso a como aparecieron**.

Ésta es la expresión de nuestro número original en la base destino.

- Notemos que cada uno de los restos obtenidos es con toda seguridad **menor que la base destino**, ya que, en otro caso, podríamos haber seguido adelante con la división entera.
- Notemos también que el último cociente es también **menor que la base destino**, por el mismo motivo de antes (podríamos haber proseguido la división).
- Lo que acabamos de decir garantiza que tanto el último cociente, como todos los restos aparecidos en el proceso, **pueden ser dígitos de un sistema en la base destino** al ser todos menores que ella.

Pregunta

¿Cómo podemos usar la Expresión General para explicar por qué este procedimiento es correcto, al menos para el caso de convertir **61 a base 3**?

Conversión de otras bases a base 10

La conversión en el sentido opuesto, de una base b cualquiera a base 10, se realiza simplemente aplicando la Expresión General. Cada uno de los dígitos del número original (ahora en base b arbitraria) es el coeficiente de alguna potencia de la base original. Esta potencia depende de la posición de dicho dígito. Una vez que escribimos

todos los productos de los dígitos originales por las potencias de la base, hacemos la suma y nos queda el resultado: el número original convertido a base 10.

Es de la mayor importancia cuidar de que las potencias de la base que intervienen en el cálculo estén **ordenadas y completas**. Es fácil si escribimos estas potencias a partir de la derecha, comenzando por la que tiene exponente 0, y vamos completando los términos de derecha a izquierda hasta agotar las posiciones del número original.

Preguntas

¿Cómo sería un sistema de **contar con los dedos en base 2**? Dedo arriba = 1, dedo abajo = 0...

- ¿Cómo hacemos el 1, el 2, el 3...?
- ¿Hasta qué número podemos contar con una mano? ¿Y con dos manos?
- ¿Y cómo se indica el **4** en este sistema?

Conversión entre bases arbitrarias

Hemos visto los casos de conversión entre base 10 y otras bases, en ambos sentidos. Ahora veamos los casos donde ninguna de las bases origen o destino son la base 10.

La buena noticia es que, en general, **esto ya sabemos hacerlo**. Si tenemos dos bases b_1 y b_2 cualesquiera, ninguna de las cuales es 10, sabiendo hacer las conversiones anteriores podemos hacer la conversión de b_1 a b_2 sencillamente haciendo **dos conversiones pasando por la base 10**. Si queremos convertir de b_1 a b_2, convertimos primero **de** b_1 **a base 10**, aplicando el procedimiento ya visto, y luego **de base 10 a** b_2. Eso es todo.

Pero en algunos casos especiales podemos aprovechar cierta relación existente entre las bases a convertir: por ejemplo, cuando son **2 y 16**, o **2 y 8**. La base 2 es la del sistema **binario**, y las bases 16 y 8 son las del sistema **hexadecimal** y del sistema **octal** respectivamente.

En estos casos, como 16 y 8 son potencias de 2 (la otra base), podemos aplicar un truco matemático para hacer la conversión en un solo paso y con muchísima facilidad. Por fortuna son estos casos especiales los que se presentan con mayor frecuencia en nuestra disciplina.

Equivalencias entre sistemas

Para poder aplicar este truco se necesita la tabla de equivalencias entre los dígitos de los diferentes sistemas. Si no logramos memorizarla, conviene al menos saber reproducirla, asegurándose de saber **contar** en las bases 2, 8 y 16 para reconstruir la tabla si es necesario. Pero con la práctica, se logra memorizarla fácilmente.

Notemos que:

- El sistema octal tiene ocho dígitos **(0 ... 7)** y cada uno de ellos se puede representar con **tres dígitos binarios**:
  - 000
  - 001
  - 010
  - 011
  - 100
  - 101
  - 110
  - 111

Notemos que:

- El sistema hexadecimal tiene dieciséis dígitos **(0 ... F)** y cada uno de ellos se puede representar con **cuatro dígitos binarios**:
  - 0000
  - 0001
  - 0010
  - 0011
  - 0100
  - 0101
  - 0110
  - 0111
  - 1000
  - 1001
  - 1010
  - 1011
  - 1100
  - 1101
  - 1110
  - 1111

Conversión entre sistemas binario y hexadecimal

El truco para convertir de base 2 a base 16 consiste simplemente en agrupar los dígitos binarios de a cuatro, y reemplazar cada grupo de cuatro dígitos por su equivalente en base 16 según la tabla anterior.

Si hace falta completar un grupo de cuatro dígitos binarios, se completa con ceros a la izquierda.

Si el problema es convertir, inversamente, de base 16 a base 2, de igual forma reemplazamos cada dígito hexadecimal por los cuatro dígitos binarios que lo representan.

Conversión entre sistemas binario y octal

El problema de convertir entre bases 2 y 8 es igual de sencillo. Basta con reemplazar cada grupo de **tres** dígitos binarios (completando con ceros a la izquierda si hace falta) por el dígito octal equivalente. Lo mismo si la conversión es en el otro sentido.

Unidades de Información

En este segundo tema de la unidad veremos qué es la información y cómo podemos cuantificar, es decir, medir, la cantidad de información que puede alojar un dispositivo, o la cantidad de información que representa una pieza cualquiera de información. Veremos además las relaciones entre las diferentes unidades de información.

Información

A lo largo de la historia se han inventado y fabricado máquinas, que son dispositivos que **transforman la energía**, es decir, convierten una forma de energía en otra. Las computadoras, en cambio, convierten una forma de **información** en otra.

Los programas de computadora reciben alguna forma de información (la **entrada** del programa), la **procesan** de alguna manera, y emiten alguna información de **salida**. La **entrada** es un conjunto de datos de partida, para que trabaje el programa, y la **salida** generada por el programa es alguna forma de respuesta o solución a un problema. Sabemos, además, que el material con el cual trabajan las computadoras son textos, documentos, mensajes, imágenes, sonido, etc. Todas estas son formas en las que se codifica y se almacena la información.

Un epistemólogo dice que la información es "una diferencia relevante". Si vemos que el semáforo cambia de rojo a verde, recibimos información ("podemos avanzar"). Al cambiar el estado del semáforo aparece una **diferencia** que puedo observar. Es **relevante** porque modifica de alguna forma el estado de mi conocimiento o me permite tomar una decisión respecto de algo.

¿Qué es, exactamente, esta información? No podemos tocarla ni pesarla, pero ¿se puede medir? Y si se puede medir, ¿entonces se puede medir la cantidad de información que aporta un texto, una imagen?

Bit

La Teoría de la Información, una teoría matemática desarrollada alrededor de 1950, dice que el **bit** es "la mínima unidad de información". Un bit es la información

que recibimos "cuando se especifica una de dos alternativas igualmente probables". Si tenemos una pregunta **binaria**, es decir, aquella que puede ser respondida **con un sí o con un no**, entonces, al recibir una respuesta, estamos recibiendo un bit de información. Las preguntas binarias son las más simples posibles (porque no podemos decidir entre **menos** respuestas), de ahí que la información necesaria para responderlas sea la mínima unidad de información.

De manera que un bit es una unidad de información que puede tomar sólo dos valores. Podemos pensar estos valores como **verdadero o falso**, como **sí o no**, o como **0 y 1**.

Cuando las computadoras trabajan con piezas de información complejas, como los textos o imágenes, estas piezas son representadas como conjuntos ordenados de bits, de un cierto tamaño. Así, por ejemplo, la secuencia de ocho bits **01000001** puede representar la letra A mayúscula. Un documento estará constituido por palabras; éstas están formadas por símbolos como las letras, y éstas serán representadas por secuencias de bits.

La memoria de las computadoras está diseñada de forma que **no se puede almacenar otra cosa que bits** en esa memoria. Los textos, las imágenes, los sonidos, los videos, los programas que ejecuta, los mensajes que recibe o envía; todo lo que puede guardar, procesar, o emitir una computadora digital, debe estar en algún momento representado por una secuencia de bits. Los bits son, en cierta forma, como los átomos de la información. Por eso el bit es la unidad fundamental que usamos para medirla, y definiremos también algunas unidades mayores, o múltiplos.

El viaje de un bit

En una famosa película de aventuras hay una ciudad en problemas. Uno de los héroes enciende una pila de leña porque se prepara un terrible ataque sobre la ciudad. La pila de leña es el dispositivo preestablecido que tiene la ciudad para pedir ayuda en caso de emergencia.

En la cima de la montaña que está cruzando el valle existe un puesto similar, con su propio montón de leña, y un vigía. El vigía ve el fuego encendido en la ciudad que pide ayuda, y a su vez enciende su señal. Lo mismo se repite de cumbre en cumbre, atravesando grandes distancias en muy poco tiempo, hasta llegar rápidamente a quienes están en condiciones de prestar la ayuda. En una tragedia griega se dice que este ingenioso dispositivo se utilizó en la realidad, para comunicar en tan sólo una noche la noticia de la caída de Troya.

La información que está transportando la señal que viaja es la respuesta a una pregunta muy sencilla: **"¿la ciudad necesita nuestra ayuda?"**.

Esta pregunta es **binaria**: se responde con un sí o con un no. Por lo tanto, lo que ha viajado es **un bit de información**.

Notemos que, en los manuales de lógica o de informática, encontraremos siempre asociados los **bits** con los valores de **0 y 1**. Aunque esto es útil a los efectos de

emplear los bits en computación, no es del todo exacto. Un bit no es exactamente un dígito. Lo que viajó desde la ciudad sitiada hasta su destino no es un 0 ni un 1. Es **un bit de información**, aquella unidad de información que permite tomar una decisión entre dos alternativas. Sin embargo, la identificación de los bits con los dígitos binarios es útil para todo lo que tiene que ver con las computadoras.

Byte

Como el bit es una medida tan pequeña de información, resulta necesario definir unidades más grandes. En particular, y debido a la forma como se organiza la memoria de las computadoras, es útil tener como unidad al **byte** (abreviado **B** mayúscula), que es una secuencia de **8 bits**. Podemos imaginarnos la memoria de las computadoras como una estantería muy alta, compuesta por estantes que contienen ocho casilleros. Cada uno de estos estantes es una **posición o celda de memoria**, y contiene exactamente ocho bits (un byte) de información.

Como los valores de los bits que forman un byte son independientes entre sí, existen 2^8 diferentes valores para esos ocho bits. Si los asociamos con números en el sistema binario, esos valores serán **00000000**, **00000001**, **00000010**, ..., etc., hasta el **11111111**. En decimal, esos valores corresponden a los números **0, 1, 2, ..., 255**.

Cada byte de la memoria de una computadora, entonces, puede alojar un número entre 0 y 255. Esos números representarán diferentes piezas de información: si los vemos como bytes independientes, pueden representar **caracteres** como letras y otros símbolos, pero también pueden estar formando parte de otras estructuras de información más complejas, y tener otros significados.

Representando datos con bytes

Para poder tratar y comunicar la información, que está organizada en bytes, es necesario que exista una asignación fija de valores binarios a caracteres. Es decir, se necesita una **tabla de caracteres** que asigne un símbolo a cada valor posible entre 0 y 255.

La memoria de la computadora es como un espacio donde se almacenan temporariamente contenidos del tamaño de un byte. Si pudiéramos ver el contenido de una sección de la memoria mientras la computadora está trabajando, veríamos que cada byte tiene determinados contenidos binarios. Esos contenidos pueden codificar los caracteres de un mensaje, carácter por carácter.

Sabiendo que la memoria está organizada en bytes, es interesante saber qué capacidad tendrá la memoria de una computadora dada y qué tamaño tendrán las piezas de información que caben en esta memoria. Como la memoria de una computadora, y la cantidad de información que compone un mensaje, un programa, una imagen, etc., suelen ser muy grandes, utilizamos **múltiplos** de la unidad de memoria, que es el byte.

Existen en realidad dos sistemas diferentes de múltiplos: el **Sistema Internacional** y el **Sistema de Prefijos Binarios**. Las unidades de ambos sistemas son parecidas, pero no exactamente iguales.

Los dos sistemas difieren esencialmente en el factor de la unidad en los sucesivos múltiplos. En el caso del Sistema Internacional, todos los factores son alguna potencia de 1000. En el caso del Sistema de Prefijos Binarios, todos los factores son potencias de 1024.

Sistema Internacional

En el llamado Sistema Internacional, la unidad básica, el byte, se multiplica por potencias de 1000. Así, tenemos:

- El **kilobyte** (1000 bytes)
- El **megabyte** (1000 $\times$ 1000 bytes = 1000 kilobytes = un millón de bytes)
- El **gigabyte** (1000 $\times$ 1000 $\times$ 1000 bytes = mil megabytes = mil millones de bytes)
- El **terabyte** (1000 $\times$ 1000 $\times$ 1000 $\times$ 1000 bytes = mil gigabytes = un billón de bytes), y otros múltiplos mayores como **petabyte, exabyte, zettabyte, yottabyte**.

Como puede verse, cada unidad se forma multiplicando la anterior por 1000.

Los símbolos de cada múltiplo, utilizados al especificar las unidades, son **k minúscula** para **kilo**, **M mayúscula** para **mega**, **G mayúscula** para **giga**, **T mayúscula** para **tera**, etc.

Sistema de Prefijos Binarios

En el llamado Sistema de Prefijos Binarios, el byte se multiplica por potencias de 2^{10}, que es 1024. Así, tenemos:

- El **kibibyte** (1024 bytes)
- El **mebibyte** (1024 $\times$ 1024 bytes, **aproximadamente** un millón de bytes, pero exactamente 1048576 bytes)
- El **gibibyte** (1024 $\times$ 1024 $\times$ 1024 bytes, **aproximadamente** mil millones de bytes)
- El **tebibyte** (1024 $\times$ 1024 $\times$ 1024 $\times$ 1024 bytes, aproximadamente un millón de mebibytes, o aproximadamente un billón de bytes), y otros múltiplos mayores como **pebibyte, exbibyte, zebibyte, yobibyte**.

Como puede verse, cada unidad se forma multiplicando la anterior por 1024.

Notemos que los prefijos **kilo, mega, giga, tera**, del Sistema Internacional, cambian a **kibi, mebi, gibi, tebi**, del sistema de Prefijos Binarios.

Los símbolos de cada múltiplo, utilizados al especificar las unidades, son **Ki**, con K mayúscula, para **kibi**, **Mi** para **mebi**, **Gi** para **gibi**, **Ti** para **tebi**, etc.

¿Por qué existen **dos** sistemas en lugar de uno? En realidad la adopción del Sistema de Prefijos Binarios se debe a las características de la memoria de las computadoras:

- Cada posición o celda de la memoria tiene su dirección, que es el número de la posición de esa celda dentro del conjunto de toda la memoria de la computadora.
- Cuando la computadora accede a una posición o celda de su memoria, para leer o escribir un contenido en esa posición, debe especificar la dirección de la celda.
- Como la computadora usa exclusivamente números binarios, al especificar la dirección de la celda usa una cantidad de dígitos binarios.
- Por lo tanto, la cantidad de posiciones que puede acceder usando direcciones es una potencia de 2: si usa 8 bits para especificar cada dirección, accederá a 2^8 bytes, cuyas direcciones estarán entre 0 y 255. Si usa 10 bits, accederá a 2^{10} bytes, cuyas direcciones serán 0 a 1023.
- Entonces, tener una memoria de, por ejemplo, exactamente **mil bytes**, complicaría técnicamente las cosas porque las direcciones 1000 a 1023 no existirían. Si un programa quisiera acceder a la posición 1020 habría un grave problema. Habría que tener en cuenta excepciones por todos lados y la vida de los diseñadores de computadoras y de los programadores sería lamentable.
- En consecuencia, todas las memorias se fabrican en tamaños que son potencias de 2 y el Sistema de Prefijos Binarios se adapta perfectamente a medir esos tamaños.

En computación se utilizan, en diferentes situaciones, ambos sistemas de unidades. Es costumbre usar el Sistema Internacional para hablar de velocidades de transmisión de datos o tamaños de archivos, pero usar Prefijos Binarios al hablar de almacenamiento de memoria, o en unidades de almacenamiento permanente, como los discos.

- Entonces, cuando un proveedor de servicios de Internet ofrece **un enlace de 1 Mbps**, nos está diciendo que por ese enlace podremos transferir **exactamente 1 millón de bits por segundo**. El proveedor utiliza el Sistema Internacional.
- Los textos, imágenes, sonido, video, programas, etc., se guardan en **archivos**, que son sucesiones de bytes. Encontramos archivos en el disco de nuestra computadora, y podemos descargar archivos desde las redes. Cuando nos interesa saber cuánto mide un archivo, en términos de bytes, usamos el Sistema Internacional porque el archivo no tiene por qué tener un tamaño que sea potencia de 2.
- Por el contrario, los fabricantes de medios de almacenamiento, como memorias, discos rígidos o pendrives, deberían (aunque a veces no lo hacen) utilizar Prefijos

Binarios para expresar las capacidades de almacenamiento de esos medios. Así, un *"pendrive de dieciséis gigabytes"*, si tiene una capacidad de 16×2^{30} bytes, debería publicitarse en realidad como *"pendrive de dieciséis gibibytes"*.

Representación de datos numéricos

Veremos de qué manera puede ser tratada mediante computadoras la información correspondiente a números, textos, imágenes y otros datos. Necesitaremos conocer las formas de representación de datos, y comenzaremos por los datos numéricos.

Representación de datos numéricos

Hemos visto ejemplos de sistemas de numeración: en base 6, en base 10, o decimal, en base 2, o binario, en base 16, o hexadecimal, y en base 8, u octal; y sabemos convertir la representación de un número en cada una de estas bases, a los sistemas en las demás bases. Sin embargo, aún nos falta considerar la representación numérica de varios casos importantes:

- Hemos utilizado estos sistemas para representar únicamente números **enteros**. Nos falta ver de qué manera representar números racionales, es decir aquellos que tienen una parte fraccionaria (los "decimales").
- Además estos enteros han sido siempre **no negativos**, es decir, sabemos representar únicamente el 0 y los naturales. Nos falta considerar los negativos.
- Por otra parte, no nos hemos planteado el problema de la **cantidad de dígitos**. Idealmente, un sistema de numeración puede usar infinitos dígitos para representar números arbitrariamente grandes. Si bien esto es matemáticamente correcto, las computadoras son objetos físicos que tienen unas ciertas limitaciones, y con ellas no es posible representar números de infinita cantidad de dígitos.

En esta parte de la unidad mostraremos sistemas de representación utilizados en computación que permiten tratar estos problemas.

Clasificación de los números

Es conveniente repasar la clasificación de los diferentes conjuntos de números y conocer las diferencias importantes entre éstos. Los títulos en el cuadro (tomado de Wikipedia) son referencias a los artículos enciclopédicos sobre cada uno de esos conjuntos.

- Números complejos

- Complejos
- Reales
- Racionales
- Enteros
- Naturales
- Uno
- Naturales primos
- Naturales compuestos
- Cero
- Enteros negativos
- Fraccionarios
- Fracción propia
- Fracción impropia
- Irracionales
- Irracionales algebraicos
- Trascendentes
- Imaginarios

Preguntas

- El **cero**, ¿es un natural?
- ¿Existen números naturales negativos? ¿Y racionales negativos?
- ¿Es correcto decir que un racional tiene una parte decimal que es, o bien finita, o bien periódica?
- ¿Puede haber dos expresiones diferentes para el mismo número, en el mismo sistema de numeración decimal?
- El número $0.9999\ldots$ con 9 periódico, y el número 1, ¿son dos números diferentes o el mismo número? Si son diferentes, ¿qué número se encuentra entre ellos dos?
- El número 1 es a la vez natural y entero. ¿Por qué no puede haber un número que sea a la vez racional e irracional?
- ¿Por qué jamás podremos computar la sucesión completa de decimales de π?

Datos enteros

Veremos un sistema de representación de datos no negativos, llamado **sin signo**, y tres sistemas de representación de datos numéricos enteros, llamados **signo-magnitud**, **complemento a 2** y **notación en exceso**.

Datos fraccionarios

Para representar fraccionarios consideraremos los sistemas de **punto fijo** y **punto flotante**.

Rango de representación

Cada sistema de representación de datos numéricos tiene su propio **rango de representación** (que podemos abreviar **RR**), o intervalo de números representables. Ningún número fuera de este rango puede ser representado en dicho sistema. Conocer este intervalo es importante para saber con qué limitaciones puede enfrentarse un programa que utilice alguno de esos sistemas.

El rango de los números representados bajo un sistema está dado por sus **límites inferior y superior**, que definen qué zona de la recta numérica puede ser representada. Como ocurre con todo intervalo numérico cerrado, el rango de representación puede ser escrito como $[a, b]$, donde a y b son sus límites inferior y superior, respectivamente.

Por la forma en que están diseñados, algunos sistemas de representación sólo pueden representar números muy pequeños, o sólo positivos, o tanto negativos como positivos. En general, el RR **será más grande cuantos más dígitos binarios**, o bits, tenga el sistema. Sin embargo, el RR depende también de la forma como el sistema **utilice** esos dígitos binarios, ya que un sistema puede ser más o menos **eficiente** que otro en el uso de esos dígitos, aunque la cantidad de dígitos sea la misma en ambos sistemas.

Por lo tanto, decimos que el rango de representación depende a la vez de la **cantidad de dígitos** y de la **forma de funcionamiento** del sistema de representación.

Representación sin signo SS(k)

Consideremos primero qué ocurre cuando queremos representar números enteros **no negativos** (es decir, **positivos o cero**) sobre una cantidad fija de bits.

En el sistema **sin signo**, simplemente usamos el sistema binario de numeración, tal como lo conocemos, **pero limitándonos a una cantidad fija** de dígitos binarios o bits. Podemos entonces abreviar el nombre de este sistema como **SS(k)**, donde k es la cantidad fija de bits, o ancho, de cada número representado.

Rango de representación de SS(k)

¿Cuál será el rango de representación? El **cero** puede representarse, así que el límite inferior del rango de representación será 0. Pero ¿cuál será el límite superior? Es decir, si la cantidad de dígitos binarios en este sistema es k, ¿cuál es el número más grande que podremos representar?

Podemos estudiarlo de dos maneras.

1. Usando combinatoria

Contemos cuántos números diferentes podemos escribir con k dígitos binarios. Imaginemos un número binario cualquiera con k dígitos. El dígito de más a la derecha tiene únicamente dos posibilidades (0 o 1). Por cada una de éstas hay

nuevamente dos posibilidades para el siguiente hacia la izquierda (lo que da las cuatro posibilidades 00, 01, 10, 11). Por cada una de éstas, hay dos posibilidades para el siguiente (dando las ocho posibilidades 000, 001, 010, 011, 100, 101, 110, 111), etc., y así hasta la posición k. No hay más posibilidades. Como hemos multiplicado 2 por sí mismo k veces, la cantidad de números que se pueden escribir es 2^k. Luego, el número más grande posible es $2^k - 1$. (**Pregunta**: ¿Por qué $2^k - 1$ y no 2^k?).

2. Usando álgebra

El número más grande que podemos representar en un sistema sin signo a k dígitos es, seguramente, aquel donde todos los k dígitos valen **1**. La Expresión General que hemos visto nos dice que si un número n está escrito en base 2, **con k dígitos**, entonces

$$n = x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0$$

y, si queremos escribir el más grande de todos, deberán ser todos los x_i iguales a 1. (**Pregunta**: ¿Por qué si el número n tiene k dígitos binarios, el índice del más significativo es $k - 1$ y no k?)

Esta suma vale entonces

$$\begin{aligned} x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0 = \\ = 1 \times 2^{k-1} + ... + 1 \times 2^1 + 1 \times 2^0 = \\ = 2^{k-1} + ... + 2^1 + 2^0 = \\ = 2^k - 1 \end{aligned}$$

Usando ambos argumentos hemos llegado a que el número más grande que podemos representar con k dígitos binarios es $2^k - 1$. Por lo tanto, **el rango de representación de un sistema sin signo a k dígitos, o SS(k), es** $[0, 2^k - 1]$. Todos los números representables en esta clase de sistemas son **positivos o cero**.

Ejemplos

- Para un sistema de representación sin signo a 8 bits: $[0, 2^8 - 1] = [0, 255]$
- Con 16 bits: $[0, 2^{16} - 1] = [0, 65{,}535]$
- Con 32 bits: $[0, 2^{32} - 1] = [0, 4{,}294{,}967{,}295]$

Representación con signo

En la vida diaria manejamos continuamente números negativos, y los distinguimos de los positivos simplemente agregando un signo "menos". Representar esos datos en la memoria de la computadora no es tan directo, porque, como hemos visto, la memoria **solamente puede alojar ceros y unos**. Es decir, ¡no podemos simplemente guardar un signo "menos"! Lo único que podemos hacer es almacenar secuencias de ceros y unos.

Esto no era un problema cuando los números eran no negativos. Para poder representar, ahora, tanto números **positivos como negativos**, necesitamos cambiar la forma de representación. Esto quiere decir que una secuencia particular de dígitos binarios, que en un sistema sin signo tiene un cierto significado, ahora tendrá un significado diferente. Algunas secuencias, que antes representaban números positivos, ahora representarán negativos.

Veremos los **sistemas de representación con signo** llamados **Signo-magnitud (SM)**, **Complemento a 2 (C2)** y **Notación en exceso**.

Es importante tener en cuenta que **solamente se puede operar entre datos representados con el mismo sistema de representación**, y que el **resultado** de toda operación **vuelve a estar representado en el mismo sistema**.

Preguntas

- ¿Cuáles son los límites del rango de representación de un sistema de representación numérica?
- Un número escrito en un sistema de representación **con signo**, ¿es siempre negativo?
- ¿Para qué querríamos escribir un número positivo en un sistema de representación con signo?

Sistema de Signo-magnitud SM(k)

El sistema de **Signo-magnitud** no es el más utilizado en la práctica, pero es el más sencillo de comprender. Se trata simplemente de utilizar un bit (el de más a la izquierda) para representar el **signo**. Si este bit tiene valor 0, el número representado es positivo; si es 1, es negativo. Los demás bits se utilizan para representar la **magnitud**, es decir, el **valor absoluto** del número en cuestión.

Ejemplos

- $7_{(10} = 00000111_{(2}$
- $-7_{(10} = 10000111_{(2}$

Como estamos reservando un bit para expresar el signo, ese bit ya no se puede usar para representar magnitud; y como el sistema tiene una cantidad de bits fija, el RR ya no podrá representar el número máximo que era posible con el sistema **sin signo**.

Rango de representación de SM(k)

- En todo número escrito en el sistema de signo-magnitud a k bits, ya sea positivo o negativo, hay un bit reservado para el signo, lo que implica que quedan $k - 1$ bits para representar su valor absoluto.

- Siendo un valor absoluto, estos $k-1$ bits representan un número **no negativo**. Además este número está representado con el sistema **sin signo** sobre $k-1$ bits, es decir, SS(k-1).
- Este número no negativo en SS(k-1) tendrá un valor máximo representable que coincide con el **límite superior** del rango de representación **de SS(k-1)**.
- Sabemos que el rango de representación de SS(k) es $[0, 2^k - 1]$. Por lo tanto, el rango de SS(k-1), reemplazando, será $[0, 2^{k-1} - 1]$.
- Esto quiere decir que el número representable en SM(k) cuyo valor absoluto es máximo, es $2^{k-1} - 1$. Por lo tanto éste es el límite superior del rango de representación de SM(k).
- Pero en SM(k) también se puede representar su opuesto negativo, simplemente cambiando el bit más alto por 1. El opuesto del máximo positivo representable es a su vez el número más pequeño, negativo, representable: $-(2^{k-1} - 1)$.

Con lo cual hemos calculado tanto el límite inferior como el superior del rango de representación de SM(k), que, finalmente, es $[-(2^{k-1} - 1), 2^{k-1} - 1]$.

Limitaciones de Signo-Magnitud

Si bien **SM(k)** es simple, no es tan efectivo, por varias razones:

- Existen dos representaciones del 0 ("positiva" y "negativa"), lo cual desperdicia un representante.
- Esto acorta el rango de representación.
- La aritmética en SM no es fácil, ya que cada operación debe comenzar por averiguar si los operandos son positivos o negativos, operar con los valores absolutos y ajustar el resultado de acuerdo al signo reconocido anteriormente.
- El problema aritmético se agrava con la existencia de las dos representaciones del cero: cada vez que un programa quisiera comparar un valor resultado de un cómputo con 0, debería hacer **dos** comparaciones.

Por estos motivos, el sistema de SM dejó de usarse y se diseñó un sistema que eliminó estos problemas, el sistema de **complemento a 2**.

Sistema de Complemento a 2

Para comprender el sistema de complemento a 2 es necesario primero conocer la **operación** de complementar a 2.

Operación de Complemento a 2

La **operación** de complementar a 2 consiste aritméticamente en obtener el **opuesto** de un número (el que tiene el mismo valor absoluto pero signo opuesto).

Para obtener el complemento a 2 de un número escrito en base 2, **se invierte cada uno de los bits (reemplazando 0 por 1 y viceversa) y al resultado se le suma 1**.

Otra forma

Otro modo de calcular el complemento a 2 de un número en base 2 es **copiar los bits, desde la derecha, hasta el primer 1 inclusive; e invertir todos los demás a la izquierda**.

Propiedad fundamental

El resultado de esta operación, C2(a), es el opuesto del número original a, y por lo tanto tiene la propiedad de que a y C2(a) suman 0:

$$C2(a) + a = 0$$

Comprobación

Podemos comprobar si la complementación fue bien hecha aplicando la **propiedad fundamental** del complemento. Si, al sumar nuestro resultado con el número original, no obtenemos 0, corresponde revisar la operación.

Ejemplos

- Busquemos el complemento a 2 de 111010. Invirtiendo todos los bits, obtenemos 000101. Sumando 1, queda 000110.
- Busquemos el complemento a 2 de 0011. Invirtiendo todos los bits, obtenemos 1100. Sumando 1, queda 1101.
- Comprobemos que el resultado obtenido en el último caso, 1101, es efectivamente el opuesto de 0011: $0011 + 1101 = 0$.

Representación en Complemento a 2

Ahora que contamos con la **operación de complementar a 2**, podemos ver cómo se construye el **sistema de representación en Complemento a 2**.

Para representar un número a en complemento a 2 a k bits, comenzamos por considerar su signo:

- Si a es positivo o cero, lo representamos como en SM(k), es decir, lo escribimos en base 2 a k bits.
- Si a es negativo, tomamos su valor absoluto y lo complementamos a 2.

Ejemplos

- Representemos el número 17 en complemento a 2 con 8 bits. Como es positivo, lo escribimos en base 2, obteniendo 00010001, que es 17 en notación complemento a 2 con 8 bits.
- Representemos el número -17 en complemento a 2 con 8 bits. Como es negativo, escribimos su valor absoluto en base 2, que es 00010001, y lo complementamos a 2. El resultado final es 11101111 que es -17 en notación complemento a 2 con 8 bits.

Conversión de C2 a base 10

Para convertir un número n, escrito en el sistema de complemento a 2, a decimal, lo primero es determinar el signo. Si el bit más alto es 1, n es negativo. En otro caso, n es positivo. Utilizaremos esta información enseguida.

- Si n es positivo, se interpreta el número como en el sistema sin signo, es decir, se utiliza la Expresión General para hacer la conversión de base como normalmente.
- Si n es negativo, se lo complementa a 2, obteniendo el opuesto de n. Este número, que ahora es positivo, se convierte a base 10 como en el caso anterior; y finalmente se le agrega el signo "-" para reflejar el hecho de que es negativo.

Ejemplos

- Convertir a decimal $n = 00010001$. Es positivo, luego, aplicamos la Expresión General dando $17_{(10}$.
- Convertir a decimal $n = 11101111$. Es negativo; luego, lo complementamos a 2 obteniendo 00010001. Aplicamos la Expresión General obteniendo $17_{(10}$. Como n era negativo, agregamos el signo menos y obtenemos el resultado final $-17_{(10}$.

RR de C2 con k bits

La forma de utilizar los bits en el sistema de complemento a 2 permite recuperar un representante que estaba desperdiciado en SM.

El rango de representación del sistema complemento a 2 sobre k bits es $[-(2^{k-1}), 2^{k-1} - 1]$. El límite superior del RR de C2 es el mismo que el de SM, pero el **límite inferior** es menor; luego el RR de C2 es mayor que el de SM.

El sistema de complemento a 2 tiene otras ventajas sobre SM:

- El cero tiene una única representación, lo que facilita las comparaciones.
- Las cuentas se hacen bit a bit, en lugar de requerir comprobaciones de signo.
- El mecanismo de cálculo es eficiente y fácil de implementar en hardware.
- Solamente se requiere diseñar un algoritmo para **sumar**, no uno para sumar y otro para restar.

Comparando rangos de representación

Diferentes sistemas, entonces, tienen diferentes rangos de representación. Si construimos un cuadro donde podamos comparar los rangos de representación **sin signo, signo-magnitud y complemento a 2** para una misma cantidad de bits, veremos que todas las combinaciones de bits están utilizadas, sólo que de diferente forma.

El cuadro comparativo para cuatro bits mostrará que las combinaciones 0000...1111 representan los primeros 16 números no negativos para el sistema sin signo, mientras que esas mismas combinaciones tienen otro significado en los sistemas con signo. En éstos últimos, una misma combinación con el bit más significativo en 1 siempre es negativa, pero el orden en que aparecen esas combinaciones es diferente entre SM y C2.

Por otro lado, los números positivos quedan representados por combinaciones idénticas en los tres sistemas, hasta donde lo permite el rango de representación de cada uno.

Si descartamos el bit de signo y consideramos sólo las magnitudes, los números negativos en SM aparecen con sus magnitudes crecientes alejándose del 0, mientras que en C2 esas magnitudes comienzan en cero al representar el negativo más pequeño posible y crecen a medida que se acercan al cero.

Aritmética en C2

Una gran ventaja que aporta el sistema en Complemento a 2 es que los diseñadores de hardware no necesitan implementar algoritmos de resta además de los de la suma. Cuando se necesita efectuar una resta, **se complementa el sustraendo** y luego se lo **suma** al minuendo. Las computadoras no restan: siempre suman.

Por ejemplo, la operación $9-8$ se realiza como $9+(-8)$, donde (-8) es el complemento a 2 de 8.

Preguntas

- Un número en complemento a 2, ¿tiene siempre su bit más a la izquierda en 1?
- El complemento a 2 de un número, es decir, **C2(x)**, ¿es siempre un número negativo?
- ¿Quién es **C2(0)**?
- ¿Cuánto vale **C2(C2(x))**? Es decir, ¿qué pasa si complemento a 2 el complemento a 2 de x?
- ¿Cuánto vale **x + C2(x)**? Es decir, ¿qué pasa si sumo a x su propio complemento a 2?
- ¿Cómo puedo verificar si calculé correctamente un complemento a 2?

Overflow o desbordamiento en C2

En todo sistema de ancho fijo, la suma de **dos números positivos, o de dos números negativos** puede dar un resultado que sea imposible de representar debido a las limitaciones del rango de representación. Este problema se conoce como desbordamiento, u *overflow*. Cuando ocurre una situación de overflow, el resultado de la operación **no es válido** y debe ser descartado.

Si conocemos los valores en decimal de dos números que queremos sumar, usando nuestro conocimiento del rango de representación del sistema podemos saber si el resultado quedará dentro de ese rango, y así sabemos, de antemano, si ese resultado será válido. Pero las computadoras no tienen forma de conocer a priori esta condición, ya que todo lo que tienen es la representación en C2 de ambos números. Por eso necesitan alguna forma de detectar las situaciones de overflow, y el modo más fácil para ellas es comprobar los dos últimos bits de la fila de bits de acarreo o *carry*.

El último bit de la fila de carry, el que se posiciona en la última de las k columnas de la representación, se llama *carry-in*. El siguiente bit de carry, el que ya no puede acarrearse sobre ningún dígito válido porque se han rebasado los k dígitos de la representación, se llama el *carry-out*.

- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **iguales**, entonces la computadora detecta que el resultado no ha desbordado y que **la suma es válida**. La operación de suma se ha efectuado exitosamente.
- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **diferentes**, entonces la computadora detecta que el resultado ha desbordado y que **la suma no es válida**. La operación de suma no se ha llevado a cabo exitosamente, y el resultado debe ser descartado.

Suma sin overflow

Siguiendo atentamente la secuencia de bits de carry podemos detectar, igual que lo hace la computadora, si se producirá un desbordamiento. En el caso de la operación $23 + (-9)$, el resultado (que es 14) cae dentro del rango de representación, y esto se refleja en los bits de *carry-in* y de *carry-out*, cuyos valores son iguales.

Suma con overflow

En el caso de la operación $123 + 9$ en C2 a 8 bits, el resultado (que es 132) cae fuera del rango de representación. Esto se refleja en los bits de *carry-in* y de *carry-out*, que son diferentes. El resultado no es válido y debe ser descartado.

Preguntas

- ¿Qué condición sobre los bits de carry permite asegurar que **no habrá** overflow?
- ¿Para qué sistemas de representación numérica usamos la condición de detección de overflow?

- ¿Puede existir overflow al sumar dos números de diferente signo?
- ¿Qué condición sobre los bits **de signo** de los operandos permite asegurar que **no habrá** overflow?
- ¿Puede haber casos de overflow al sumar dos números negativos?
- ¿Puede haber casos de overflow al restar dos números?

Extensión de signo en C2

Para poder efectuar una suma de dos números, ambos operandos deben estar representados en el mismo sistema de representación.

- Una suma de dos operandos donde uno esté, por ejemplo, en SM y el otro en C2, no tiene sentido aritmético.
- Además, la cantidad de bits de representación debe ser la misma.

En una suma en C2, si uno de los operandos estuviera expresado en un sistema con menos bits que el otro, será necesario convertirlo al sistema del otro (**extenderlo**) y operar con ambos en ese sistema de mayor ancho.

Si el operando en el sistema de menor ancho es positivo, la extensión se realiza simplemente **completando con ceros a la izquierda** hasta obtener la cantidad de dígitos del otro sistema. Si el operando del menor ancho es negativo, la extensión de signo se hace **agregando unos**.

Ejemplos

- $A + B = 00101011_{(2} + 00101_{(2}$
  - A está en C_2^8 y B en C_2^5
  - Se completa B (positivo) como $00000101_{(2}$
- $A + B = 1010_{(2} + 0110100_{(2}$
  - A está en C_2^4 y B en C_2^7
  - Se completa A (negativo) como $1111010_{(2}$

Notación en exceso o *bias*

En un sistema de notación en exceso, se elige un intervalo $[a, b]$ de enteros a representar, y todos los valores dentro del intervalo se representan con una secuencia de bits de la misma longitud.

La cantidad de bits deberá ser la necesaria para representar todos los enteros del intervalo, inclusive los límites, y por lo tanto estará en función de la longitud del intervalo. Un intervalo $[a, b]$ de enteros, con sus límites incluidos, comprende exactamente $n = b - a + 1$ valores. Esta longitud del intervalo debe ser cubierta

con una cantidad k de bits suficiente, lo cual obliga a que $2^k \geq n$. Supongamos que n sea una potencia de 2 para facilitar las ideas, de forma que $2^k = n$.

Las 2^k secuencias de k bits, ordenadas como de costumbre según su valor aritmético, se aplican a los enteros en $[a, b]$, uno por uno. Es decir, si usamos 3 bits, las secuencias serán 000, 001, 010, ... hasta 111; y los valores representados serán respectivamente:

- 000 = a
- 001 = $a + 1$
- 010 = $a + 2$
- ...
- 111 = b

Notemos que tanto a como b pueden ser **negativos**. Así podemos representar intervalos de enteros arbitrarios con secuencias de k bits, lo que nos vuelve a dar un sistema de representación con signo.

Con este método no es necesario que el bit de orden más alto represente el signo. Tampoco que el intervalo contenga la misma cantidad de números negativos que positivos o cero, aunque para la mayoría de las aplicaciones es lo más razonable.

El sistema en exceso se utiliza como componente de otro sistema de representación más complejo, la representación en punto flotante.

Conversión entre exceso y decimal

Una vez establecido un sistema en exceso que representa el intervalo $[a, b]$ en k bits:

- Para calcular la secuencia binaria que corresponde a un valor decimal d, a d **le restamos** a y luego convertimos el resultado (que será **no negativo**) a **SS(k)**, es decir, a binario sin signo sobre k bits.
- Para calcular el valor decimal d representado por una secuencia binaria, convertimos la secuencia a decimal como en **SS(k)**, y al resultado (que será **no negativo**) le **sumamos** el valor de a.

Ejemplos

Representemos en sistema en exceso el intervalo $[10, 25]$ (que contiene $25-10+1 = 16$ enteros). Como necesitamos 16 secuencias binarias, usaremos 4 bits que producirán las secuencias 0000, 0001, ..., 1111.

- Para calcular la secuencia que corresponde al número 20, hacemos $20-10 = 10$ y el resultado será la secuencia **1010**.
- Para calcular el valor decimal que está representando la secuencia **1011**, convertimos 1011 a decimal, que es 11, y le sumamos 10; el resultado es 21.

Representemos en sistema en exceso el intervalo $[-3, 4]$ (que contiene $4-(-3)+1 = 8$ enteros). Como necesitamos 8 secuencias binarias, usaremos 3 bits que producirán las secuencias 000, 001, . . . , 111.

- Para calcular la secuencia que corresponde al número 2, hacemos $2-(-3) = 5$ y el resultado será la secuencia **101**.
- Para calcular el valor decimal que está representando la secuencia **011**, convertimos 011 a decimal, que es 3, y le sumamos -3; el resultado es 0.

Preguntas sobre Notación en Exceso

- Dado un valor decimal a representar, ¿cómo calculamos el binario?
- Dado un binario, ¿cómo calculamos el valor decimal representado?
- El sistema en exceso ¿destina un bit para representar el signo?
- ¿Se puede representar un intervalo que no contenga el cero?
- ¿Cómo se comparan dos números en exceso para saber cuál es el mayor?

Decimal a PF(n,k)

Para representar un decimal fraccionario a, positivo o negativo, en notación de punto fijo en n lugares con k fraccionarios ($PF(n, k)$), necesitamos obtener su parte entera y su parte fraccionaria, y expresar cada una de ellas en la cantidad de bits adecuada a la notación. Para esto completaremos la parte entera con ceros a la izquierda hasta obtener $n - k$ dígitos, y completaremos la parte fraccionaria con ceros por la derecha, hasta obtener k dígitos. Una vez expresado así, lo tratamos como si en realidad fuera $a \times 2^k$, y por lo tanto, un entero.

- Si es positivo, calculamos la secuencia de dígitos binarios que expresan su parte entera y su parte fraccionaria, y escribimos ambas sobre la cantidad de bits adecuada.
- Si es negativo, consideramos su valor absoluto y procedemos como en el punto anterior. Luego complementamos a 2 como si se tratara de un entero.

Truncamiento

Al escribir la parte fraccionaria de un número a en k bits (porque ésta es la capacidad del sistema de representación de punto fijo con k dígitos fraccionarios), en el caso general estaremos **truncando** el desarrollo fraccionario. El número a podría tener otros dígitos diferentes de cero más allá de la posición k. Sin embargo, el sistema no permite representarlos, y esa información se perderá.

La consecuencia del truncamiento es la aparición de un **error de truncamiento** o pérdida de precisión. El número almacenado en el sistema PF(n,k) será una aproximación con k dígitos fraccionarios al número original a, y no estará representándolo con todos sus dígitos fraccionarios.

¿Cuál es el valor de este error de truncamiento, es decir, cuál es, cuantitativamente, la diferencia entre a y la representación en PF(n,k)? Si los primeros k dígitos del desarrollo fraccionario real de a se han conservado, entonces la diferencia es menor que 2^{-k}.

Ejemplo

Representemos 3.1459 en notación PF(8,3). Parte entera: 00011. Parte fraccionaria: 001. Representación obtenida: 00011001. Reconvirtiendo 00011001 a decimal, obtenemos parte entera 3 y parte fraccionaria 0.125; de modo que el número representado en PF(8,3) como 00011001 es en realidad **3.125** y no 3.1459.

El error de truncamiento es $3{,}1459 - 3{,}1250 = 0{,}0209$, que es menor que $2^{-3} = 0{,}125$.

PF(n,k) a decimal

Para convertir un binario en notación de punto fijo en n lugares con k fraccionarios (PF(n,k)) a decimal:

- Si es positivo, aplicamos la Expresión General extendida, utilizando los exponentes negativos para la parte fraccionaria.
- O bien, lo consideramos como un entero, convertimos a decimal y finalmente lo dividimos por 2^k.
- Si es negativo, lo complementamos a 2 y terminamos operando como en el caso positivo.
- Finalmente agregamos el signo $-$ para expresar que se trata de un número negativo.

Preguntas

- ¿A qué número decimal corresponde...
  - $0011{,}0000$?
  - $0001{,}1000$?
  - $0000{,}1100$?
- ¿Cómo se representan en $PF(8,4)$...
  - $0{,}5$?
  - $-7{,}5$?
- ¿Cuál es el RR de $PF(8,3)$? ¿Y de $PF(8,k)$?

La representación de punto fijo es adecuada para cierta clase de problemas donde los datos que se manejan son de magnitudes y precisiones comparables.

- En la situación contraria, cuando las magnitudes de los datos son muy variadas, habrá datos de valor absoluto muy grande, lo que hará que sea necesario elegir una representación de una gran cantidad de bits de ancho. Pero esta cantidad de bits quedará desperdiciada al representar los datos de magnitud pequeña.

- Otro tanto ocurre con los bits destinados a la parte fraccionaria. Si los requerimientos de precisión de los diferentes datos son muy altos, será necesario reservar una gran cantidad de bits para la parte fraccionaria. Esto permitirá almacenar los datos con mayor cantidad de dígitos fraccionarios, pero esos bits quedarán desperdiciados al almacenar otros datos.

Las ventajas de la representación en punto fijo provienen, sobre todo, de que permite reutilizar completamente la lógica ya implementada para tratar enteros en complemento a 2, sin introducir nuevos problemas ni necesidad de nuevos recursos. Como la lógica para C2 es sencilla y rápida, la representación de punto fijo es adecuada para sistemas que deben ofrecer una determinada *performance*:

- Los sistemas que deben ofrecer un tiempo de respuesta corto, especialmente aquellos interactivos, como los juegos.
- Los de tiempo real, donde la respuesta a un cómputo debe estar disponible en un tiempo menor a un plazo límite, generalmente muy corto.
- Los sistemas empotrados o embebidos, que suelen enfrentar restricciones de espacio de memoria y de potencia de procesamiento.

Por el contrario, algunas clases de programas suelen manipular datos de otra naturaleza. No es raro que aparezcan en el mismo programa, e incluso en la misma instrucción de programa, datos o variables de magnitud o precisión extremadamente diferentes.

Por ejemplo, si un programa de cómputo científico necesita calcular el **tiempo en que la luz recorre una millonésima de milímetro**, la fórmula a aplicar relacionará la velocidad de la luz en metros por segundo (unos $300{,}000{,}000 m/s$) con el tamaño en metros de un nanómetro ($0{,}000000001 m$).

Estos dos datos son extremadamente diferentes en magnitud y cantidad de dígitos fraccionarios. La velocidad de la luz es un número astronómicamente grande en comparación a la cantidad de metros en un nanómetro; y la precisión con que necesitamos representar al nanómetro no es para nada necesaria al representar la velocidad de la luz.

Notación Científica

En Matemática, la respuesta al problema del cálculo con variables tan diferentes existe desde hace mucho tiempo, y es la llamada **Notación Científica**. En Notación

Científica, los números se expresan en una forma estandarizada que consiste de un **coeficiente, significando o mantisa** multiplicado por **una potencia de 10**. Es decir, la forma general de la notación es $m \times 10^e$, donde m, el coeficiente, **es un número positivo o negativo**, y e, el **exponente**, es un entero positivo o negativo.

La notación científica puede representar entonces números muy pequeños y muy grandes, todos en el mismo formato, con economía de signos y permitiendo operar entre ellos con facilidad. Al operar con cantidades en esta notación podemos aprovechar las reglas del Álgebra para calcular m y e separadamente, y evitar cuentas con muchos dígitos.

Ejemplo

Los números mencionados hace instantes, la velocidad de la luz en metros por segundo, y la longitud en metros de un nanómetro, se representarán en notación científica como 3×10^8 y 1×10^{-9}, respectivamente.

El tiempo en que la luz recorre una millonésima de milímetro se computará con la fórmula $t = e/v$, con los datos expresados en notación científica, como:

$$e = 1 \times 10^{-9} m$$
$$v = 3 \times 10^8 m/s$$
$$t = e/v = (1 \times 10^{-9}\ m)/(3 \times 10^8\ m/s) =$$
$$t = 1/3 \times 10^{-9-8}\ s =$$
$$t = 0{,}333 \times 10^{-17}\ s$$

Normalización

El resultado que hemos obtenido en el ejemplo anterior debe quedar **normalizado** llevando el coeficiente m a un valor **mayor o igual que 1 y menor que 10**. Si modificamos el coeficiente al normalizar, para no cambiar el resultado debemos ajustar el exponente.

Ejemplo

El resultado que obtuvimos anteriormente al computar $t = 1/3 \times 10^{-9-8}\ s$ fue $0{,}333 \times 10^{-17}\ s$. Este coeficiente $0{,}333$ no cumple la regla de normalización porque no es **mayor o igual que 1**.

- Para normalizarlo, lo multiplicamos por 10, convirtiéndolo en $3{,}33$.
- Para no cambiar el resultado, dividimos todo por 10 afectando el exponente, que de -17 pasa a ser -18.
- El resultado queda normalizado como $0{,}333 \times 10^{-18}$.

Normalización en base 2

Es perfectamente posible definir una notación científica en otras bases. En base 2, podemos escribir números con parte fraccionaria en notación científica normalizada desplazando la coma o punto fraccionario hasta dejar una parte entera **igual a 1** (ya que es el único valor binario que cumple la condición de normalización) y ajustando el exponente de base 2, de manera de no modificar el resultado.

Ejemplos

- $100{,}111_{(2} = 1{,}00111_{(2} \times 2^2$
- $0{,}0001101_{(2} = 1{,}101_{(2} \times 2^{-4}$

Representación en Punto Flotante

La herramienta matemática de la Notación Científica ha sido adaptada al dominio de la computación definiendo métodos de **representación en punto flotante**. Estos métodos resuelven los problemas de los sistemas de punto fijo, abandonando la idea de una cantidad fija de bits para parte entera y parte fraccionaria. En su lugar, inspirándose en la notación científica, los formatos de punto flotante permiten escribir números de un gran rango de magnitudes y precisiones en un campo de tamaño fijo.

Actualmente se utilizan los estándares de cómputo en punto flotante definidos por la organización de estándares **IEEE** (Instituto de Ingeniería Eléctrica y Electrónica, o "I triple E").

Estos estándares son dos, llamados **IEEE 754 en precisión simple y en precisión doble**.

- IEEE 754 precisión simple
  - Se define sobre un campo de 32 bits
  - Cuenta con **1 bit de signo**
  - Reserva **8 bits para el exponente**
  - Reserva **23 bits para la mantisa**
- IEEE 754 precisión doble
  - Se define sobre un campo de 64 bits
  - Cuenta con **1 bit de signo** igual que en precisión simple
  - Reserva **11 bits para el exponente**
  - Reserva **52 bits para la mantisa**

La definición de los formatos está acompañada por la especificación de mecanismos de cálculo para usarlos, manejo de errores y otra información importante.

Conversión de decimal a punto flotante

Para convertir manualmente un número decimal n a punto flotante necesitamos calcular los tres elementos del formato de punto flotante: **signo** (que llamaremos s), **exponente** (que llamaremos e) y **mantisa** (que llamaremos m), en la cantidad de bits correcta según el formato de precisión simple o doble que utilicemos.

En el curso utilizaremos siempre el formato de precisión simple. Una vez conocidos s, e y m, sólo resta escribirlos como secuencias de bits de la longitud que especifica el formato.

1. Separar el **signo** y escribir el valor absoluto de n en base 2.
    - Si n es positivo (respectivamente, negativo), s será 0 (respectivamente, 1). Separado el signo, consideramos únicamente el **valor absoluto** de n y lo representamos en base 2 como se vio al convertir un decimal fraccionario a base 2.
2. Escribir el valor binario de n en notación científica ** en base 2 normalizada**.
    - Para convertir n a notación científica lo multiplicamos por una potencia de 2 de modo que la parte entera sea 1 (condición para la normalización). El resto de la expresión binaria se convierte en parte fraccionaria. Para no cambiar el valor de n, lo multiplicamos por una potencia de 2 inversa a aquella que utilizamos.
3. El exponente, positivo o negativo, que aplicamos en el paso anterior debe ser expresado en notación en exceso a 127.
    - Al exponente se le suma 127 para representar valores en el intervalo $[-127, 128]$ con 8 bits. Esta representación se elige para poder hacer comparables directamente dos números expresados en punto flotante.
4. El coeficiente calculado se guarda **sin su parte entera** en la parte de mantisa.
    - Como la normalización obliga a que la parte entera de la mantisa sea 1, no tiene mayor sentido utilizar un bit para guardarlo en el formato de punto flotante: guardarlo no aportaría ninguna información. Por eso basta con almacenar la parte fraccionaria de la mantisa, hasta los 23 bits disponibles (o completando con ceros).

Ejemplo de Punto Flotante

Recorramos los pasos para la conversión manual a punto flotante precisión simple, partiendo del decimal $n = -5{,}5$. Recordemos que necesitamos averiguar s, e y m.

- n es negativo, luego $s = 1$.
- $|n| = 5{,}5$. Convirtiendo el valor absoluto a binario obtenemos $101{,}1_{(2}$.

- Normalizando, queda $101{,}1_{(2} = 1{,}011_{(2} \times 2^2$.
- Del paso anterior, el exponente 2 se representa en exceso a 127 como $e = 2 + 127 = 129$. En base 2, $129 = 10000001_{(2}$.
- Del mismo paso anterior extraemos la mantisa quitando la parte entera: $1{,}011 - 1 = 0{,}011$. Los bits de m son $011000000...$ con ceros hasta la posición 23.
- Finalmente, $s, e, m = 1, 10000001, 011000000000....$

Lo que significa que la representación en punto flotante de $-5{,}5$ es igual a $11000000101100000000...$ (con ceros hasta completar los 32 bits de ancho total).

Expresión de punto flotante en hexadecimal

Para facilitar la escritura y comprobación de los resultados, es conveniente leer los 32 bits de la representación en punto flotante precisión simple como si se tratara de 8 dígitos hexadecimales. Se aplica la regla, que ya conocemos, de sustituir directamente cada grupo de 4 bits por un dígito hexadecimal.

Así, en el ejemplo anterior, la conversión del decimal $-5{,}5$ resultó en la secuencia de bits $11000000101100000...$ (con más ceros).

Es fácil equivocarse al transcribir este resultado. Pero sustituyendo los bits, de a grupos de 4, por dígitos hexadecimales, obtenemos la secuencia equivalente $C0B00000$, que es más simple de leer y de comunicar.

Conversión de punto flotante a decimal

Teniendo un número expresado en punto flotante precisión simple, queremos saber a qué número decimal equivale. Separamos la representación en sus componentes s, e y m, que tienen **1, 8 y 23 bits** respectivamente, y "deshacemos" la transformación que llevó a esos datos a ocupar esos lugares. De cada componente obtendremos un factor de la fórmula final.

- Signo
  - El valor de s nos dice si el decimal es positivo o negativo.
  - La fórmula $(-1)^s$ da -1 si $s = 1$, y 1 si $s = 0$.
- Exponente
  - El exponente está almacenado en la representación IEEE 754 como ocho bits en exceso a 127. Corresponde **restar 127** para volver a obtener el exponente de 2 que afectaba al número originalmente en notación científica normalizada.
  - La fórmula $2^{(e-127)}$ dice cuál es la potencia de 2 que debemos usar para ajustar la mantisa.

- Mantisa
  - La mantisa está almacenada sin su parte entera, que en la notación científica normalizada en base 2 **siempre es 1**. Para recuperar el coeficiente o mantisa original hay que restituir esa parte entera igual a 1.
  - La fórmula $1 + m$ nos da la mantisa binaria original.

Reuniendo las fórmulas aplicadas a los tres elementos de la representación, hacemos el cálculo multiplicando los tres factores:

$$n = (-1)^s \times 2^{(e-127)} \times (1 + m)$$

obteniendo finalmente el valor decimal representado.

Ejemplo

Para el valor de punto flotante IEEE 754 precisión simple representado por la secuencia hexadecimal $C0B00000$, encontramos que $s = 1$, $e = 129$, $m = 011000....$

- Signo
  - $(-1)^s = (-1)^1 = -1$
- Exponente
  - $e = 129 \rightarrow 2^{(e-127)} = 2^2$
- Mantisa
  - $m = 0110000... \rightarrow (1 + m) = 1{,}011000....$

Ajustando la mantisa $1{,}011000...$ por el factor 2^2 obtenemos $101{,}1$. Convirtiendo a decimal obtenemos $5{,}5$. Aplicando el signo recuperamos finalmente el valor $-5{,}5$, que es lo que está representando la secuencia $C0B00000$.

Error de truncamiento

Aunque los 23 bits de mantisa del formato de punto flotante en precisión simple son suficientes para la mayoría de las aplicaciones, existen números que no pueden ser representados, ni aun en doble precisión. El caso más evidente es el de aquellos números que por su magnitud caen fuera del rango de representación del sistema. Sin embargo, el formato IEEE 754 también encuentra limitaciones al tratar con números aparentemente tan pequeños como 0.1 o 0.2. ¿Cuál es el problema en este caso?

Si hacemos manualmente el cálculo de la parte fraccionaria binaria de 0.1 (o de 0.2) encontraremos que esta parte fraccionaria es **periódica**. Esto ocurre porque $0{,}1 = 1/10$, y el denominador 10 contiene factores que no dividen a la base (es decir, el 5, que no divide a 2). Lo mismo ocurre en base 10 cuando computamos $1/3$, que tiene infinitos decimales periódicos porque el denominador 3 no divide a 10, la base.

Cuando un lenguaje de computación reconoce una cadena de caracteres como "0.1", introducida por el programador o el usuario, advierte que se está haciendo referencia a un número con decimales, e intenta representarlo en la memoria como un número en punto flotante. La parte fraccionaria debe ser forzosamente **truncada**, ya sea a los 23 bits, porque se utiliza precisión simple, o a los 52 bits, cuando se utiliza precisión doble. En ambos casos, el número representado es una aproximación al 0.1 original, y esta aproximación será mejor cuantos más bits se utilicen; pero en cualquier caso, esta parte fraccionaria almacenada en la representación en punto flotante es **finita**, de manera que nunca refleja el verdadero valor que le atribuimos al número original.

A partir del momento en que ese número queda representado en forma aproximada, todos los cómputos realizados con esa representación adolecen de un **error de truncamiento**, que va agravándose a medida que se opera con ese número representado.

En precisión simple, se considera que tan sólo **los primeros siete decimales** de un número en base 10 son representados en forma correcta. En precisión doble, sólo los primeros quince decimales son correctos.

Casos especiales en punto flotante

En el estándar IEEE 754, no todas las combinaciones de s, e y m dan representaciones con sentido, o con el sentido esperable.

Por ejemplo, con las fórmulas presentadas, no es posible representar el **cero**, ya que toda mantisa normalizada lleva una parte entera igual a 1, y los demás factores nunca pueden ser iguales a 0. Entonces, para representar el 0 en IEEE 754 se recurre a una **convención**, que se ha definido como la combinación de **exponente 0 y mantisa 0**, cualquiera sea el signo.

Los números **normalizados** en IEEE 754 son aquellos que provienen de una expresión en notación científica normalizada con exponente diferente de -127, y son la gran mayoría de los representables. Sin embargo, el estándar permite la representación de una clase de números muy pequeños, con parte entera 0 en la notación científica, que son los llamados **desnormalizados**.

Otros números especiales son aquellos donde el exponente consiste en ocho **unos** binarios con mantisa 0. Estos casos están reservados para representar los valores **infinito** positivo y negativo (que aparecen cuando una operación arroja un resultado de **overflow** del formato de punto flotante).

Similarmente, cuando el exponente vale ocho unos, y la mantisa es diferente de 0, se está representando un caso de **NaN** (**Not a Number**, "no es un número"). Estos casos patológicos sólo ocurren cuando un proceso de cálculo lleva a una condición de error (por intentar realizar una operación sin sentido en el campo real, como obtener una raíz cuadrada de un real negativo).

Representación de Texto y multimedia

En esta parte de la unidad veremos la forma de representar otras clases de información no numérica, como los textos y las imágenes.

Codificación de texto

Cuando escribimos texto en nuestra computadora, estamos almacenando temporariamente en la memoria una cierta secuencia de números que corresponden a los **caracteres**, o símbolos que tipeamos en nuestro teclado.

Estos caracteres tienen una **representación gráfica** en nuestro teclado, en la pantalla o en la impresora, pero mientras están en la memoria no pueden ser otra cosa que **bytes**, es decir, conjuntos de ocho dígitos binarios.

Para lograr almacenar caracteres de texto necesitamos adoptar una **codificación**, es decir, una tabla que asigne a cada carácter un patrón de bits fijo.

Esta codificación debe ser universal: para poder compartir información entre usuarios, o entre diferentes aplicaciones, se requiere algún estándar que sea comprendido y respetado por todos los usuarios y las aplicaciones. Hacia la mitad del siglo XX no existía un único estándar, y cada fabricante de computadoras definía el suyo propio. La comunicación entre diferentes computadoras y sistemas era complicada y llevaba mucho trabajo improductivo.

Códigos de caracteres

Inicialmente se estableció con este fin el **código ASCII**, que durante algún tiempo fue una buena solución. El código ASCII relaciona cada secuencia de **siete bits** con un carácter (o **grafema**) específico de la **Tabla ASCII**. Es decir que hay $2^7 = 128$ posibles caracteres codificados por el código ASCII.

Sin embargo, el código ASCII es insuficiente para muchas aplicaciones: no contempla las necesidades de diversos idiomas. Por ejemplo, nuestra letra Ñ no figura en la tabla ASCII. Tampoco las vocales acentuadas, ni con diéresis, como tampoco decenas de otros caracteres de varios idiomas europeos. Peor aún, con solamente 128 posibles patrones de bits, es imposible representar algunos idiomas orientales como el chino, que utilizan miles de ideogramas.

Por este motivo se estableció más tarde una familia de nuevos estándares, llamada Unicode. Uno de los estándares o esquemas de codificación definidos por Unicode, el más utilizado actualmente, se llama **UTF-8**. Este estándar mantiene la codificación que ya empleaba el código ASCII para su conjunto de caracteres, pero agrega códigos de dos, tres y cuatro bytes para otros símbolos. El resultado es que hoy, con UTF-8, se pueden representar todos los caracteres de cualquier idioma conocido. Más aún, con UTF-8 pueden codificarse textos multilingües.

Otro estándar utilizado, **ISO/IEC 8851**, codifica los caracteres de la mayoría de los idiomas de Europa occidental.

El código ASCII, los diferentes esquemas de Unicode, y el estándar ISO/IEC 8851, coinciden en la codificación de las letras del alfabeto inglés, que son comunes a la mayoría de los idiomas occidentales, y en la codificación de símbolos usuales como los dígitos, símbolos matemáticos, y otros. Por este motivo son relativamente compatibles, aunque cuando el texto utiliza otros caracteres aparecen diferencias.

Tabla de códigos ASCII

El código ASCII asigna patrones de siete bits a un conjunto de caracteres que incluye:

- Las 25 letras del alfabeto inglés, mayúsculas y minúsculas;
- Los dígitos del 0 al 9,
- Varios símbolos matemáticos, de puntuación, etc.,
- El espacio en blanco,
- Y 32 caracteres no imprimibles. Estos caracteres no imprimibles son combinaciones de bits que no tienen una representación gráfica o grafema, sino que sirven para diversas funciones de comunicación de las computadoras con otros dispositivos. Suelen ser llamados **caracteres de control**.

En general, prácticamente todos los símbolos que figuran en nuestro teclado tienen un código ASCII asignado. Como sólo se usan siete bits, el bit de mayor orden (el de más a la izquierda) de cada byte siempre es cero, y por lo tanto los códigos ASCII toman valores de 0 a 127.

Textos y documentos

Un archivo de texto es una sucesión de caracteres codificados bajo algún estándar. Puede manipularse con programas básicos como los **editores de texto** u otras herramientas que ofrece el ambiente del sistema operativo. Un archivo de texto es directamente legible por humanos porque contiene únicamente los caracteres que constituyen las palabras, espacios en blanco o saltos de línea.

Otra clase de archivos, los que son creados y manipulados por **procesadores de texto**, además de esa información tienen una estructura compleja que permite definir características de presentación y organización del texto. Esto incluye los diferentes tipos, tamaños o colores de los caracteres, las dimensiones de la página, la organización en secciones o capítulos, etc. La estructura de los archivos generados por los procesadores de texto es específica de cada programa y convierte al documento en algo que sólo puede ser leído con el procesador de texto correspondiente.

Archivos de hipertexto

Una página HTML servida por un servidor Web es un archivo de texto que suele estar codificado en el estándar UTF-8. El contenido de este texto es directamente legible, pero no es exactamente lo que muestra el navegador, sino que esa representación gráfica está indicada por el lenguaje HTML en el que está escrito el documento. Las propiedades de navegación del documento también están determinadas por elementos del lenguaje HTML.

Las primeras líneas del documento HTML definen cuestiones relativas a la presentación que hará el navegador, como el idioma en el cual está escrita la página, el conjunto de caracteres que la codifica, el título que debe presentarse en la ventana de visualización, etc. Estas líneas se especifican en el lenguaje especial de la Web, el lenguaje de marcado de hipertexto, o HTML.

Con el navegador podemos visualizar el texto de esa página pulsando las teclas CTRL+U. Lo mismo si descargamos la página hacia un archivo y usamos el comando **head**. Lo que se ve es diferente de lo que muestra el navegador: se trata del **código fuente** de la página HTML.

Presentemos otras vistas del mismo archivo de texto, a fin de mostrar que se compone simplemente de una secuencia de bytes.

Con diferentes comandos o programas de visualización podemos ver, carácter por carácter, cómo está construido este texto. El comando **hexdump -bc** nos da la lista de los caracteres que componen el texto, con la notación en octal de su código, que aparece encima de cada uno de ellos.

Las letras acentuadas se representan con una serie de caracteres UTF-8 especiales, no pertenecientes a la zona visible del ASCII. El comando separa el carácter en los bytes que lo componen y los muestra individualmente.

Los caracteres de control, como el tabulador y el fin de línea, no tienen un grafema asociado, sino que se representan por las secuencias **\t** y **\n** respectivamente. Estos caracteres desplazan el cursor de posición que escribe los caracteres en pantalla (o en una impresora) para organizar visualmente la presentación del texto, y también son parte del código fuente de la página.

Del mismo modo, el comando **hexdump -C** muestra cada uno de los grafemas de los caracteres acompañado de su codificación en hexadecimal. Esta vista no muestra los caracteres acentuados ni los de control, sino que los reemplaza por puntos.

Pregunta

- Estos comandos aplicados a un documento HTML muestran información legible porque se trata, esencialmente, de un archivo de texto. ¿Qué ocurre si los mismos comandos se aplican a un archivo creado por un procesador de texto?

Imagen digital

Otras clases de datos, diferentes del texto, también requieren codificación (porque siempre deben ser almacenados en la memoria en forma de bits y bytes), pero su tratamiento es diferente.

Introducir en la computadora, por ejemplo, una imagen analógica (tal como un dibujo o una pintura hecha a mano), o un fragmento de sonido tomado del ambiente, requiere un proceso previo de **digitalización**. Digitalizar es convertir en digital la información que es analógica, es decir, convertir un rango **continuo** de valores (lo que está en la naturaleza) a un conjunto **discreto** de valores numéricos.

Si partimos de una imagen analógica, el proceso de digitalización involucra la división de la imagen en una fina cuadrícula, donde cada elemento de la cuadrícula abarca un pequeño sector cuadrangular de la imagen. A cada elemento de la cuadrícula se le asignan valores discretos que codifican el color de la imagen en ese lugar.

Estos elementos o puntos se llaman **pixels** (del inglés, **picture element**). La imagen queda constituida por una sucesión de valores de color para cada pixel de los que forman la imagen.

En general, mientras más elementos de cuadrícula (más pixels) podamos representar, mejor será la aproximación a nuestra pieza de información original. Mientras más fina la cuadrícula (es decir, mientras mayor sea la **resolución** de la imagen digitalizada), y mientras más valores discretos usemos para representar los colores, más se parecerá nuestra versión digital al original analógico.

Notemos que la digitalización de una imagen implica la discretización de **dos** variables analógicas:

- Por un lado, los infinitos puntos de la imagen analógica, bidimensional, deben reducirse a unos pocos rectángulos discretos.
- Por otro lado, los infinitos valores de color deben reducirse a unos pocos valores discretos, en el rango de nuestro esquema de codificación.

Este proceso de digitalización es el que hacen automáticamente una cámara de fotos digital o un celular, almacenando luego los bytes que representan la imagen tomada.

Color

Hay varias maneras de representar el color en las imágenes digitales. Una forma es definir, para cada pixel o punto de la imagen, tres coordenadas que describen las intensidades de luz **roja, verde y azul** que conforman cada color.

Cuando se crea una mezcla de rayos de luz de colores con diferentes intensidades, usando un proyector o una pantalla como los displays LED, las ondas luminosas individuales del rojo, verde y azul se suman formando otros colores. Este esquema

de representación del color se llama **RGB** por las iniciales de los colores rojo, verde y azul en inglés.

Para cada punto, esas tres coordenadas son números en un cierto intervalo. El valor mínimo de una coordenada, el 0, representa la ausencia de ese color. El valor máximo, la intensidad máxima de ese color que se puede reproducir con el dispositivo de salida que lo está visualizando. Cuando las coordenadas se representan en un byte, cada coordenada puede ir entre 0 y 255.

Así, la terna (0, 0, 0) representa el negro (ausencia de los tres colores), la terna (255, 255, 255) el blanco (valores máximos de los tres colores, sumados), etc.

Profundidad de color

Con este esquema de representación de color, cada pixel o elemento de la imagen quedaría representado por tres bytes, o 24 bits. Sin embargo, las cámaras fotográficas digitales modernas utilizan un esquema de codificación con mucha mayor **profundidad de color** (es decir, más bits por cada coordenada de color) que en el ejemplo anterior.

Formato de imagen

Lógicamente, para las imágenes con muchos colores (como las escenas de la naturaleza donde hay gradaciones de colores) es conveniente contar con muchos bits de profundidad de color. Sin embargo, cuando una imagen se compone de pocos colores, la imagen digital es innecesariamente grande, costosa de almacenar y de transmitir. En estos casos es útil definir un **formato de imagen** que represente esos pocos colores utilizando menos bits.

Una forma de hacerlo es definir una **paleta de colores**, que es una lista de los diferentes colores utilizados en la imagen, codificados con la mayor economía de bits posible. Si conocemos la cantidad de colores en la imagen, podemos determinar la cantidad mínima de bits que permite codificarlos a todos.

Así, cada pixel de la imagen, en lugar de quedar representado por una terna de valores, puede representarse por un número de color en la paleta.

Queda por especificar **cuál color es el que está codificado por cada número de color** de la paleta. Si una imagen tiene dos bits de profundidad de color, los colores serán cuatro, y sus códigos serán **00, 01, 10, 11**. Pero, ¿cuáles exactamente son estos colores? Tal vez, blanco, negro, rojo y azul. Pero tal vez sean cuatro niveles de gris. O cuatro diferentes tonos de verde.

Para simplificar nuestro trabajo asumiremos que esto no es importante, sino que el problema consiste únicamente en que nuestro formato determine los códigos de colores de cada uno de los pixels. El problema de cuáles son los colores asignados a

esos códigos puede resolverse de otras maneras: por ejemplo, suponiendo que existe una hipotética tabla universal de colores y códigos, conocida por todos.

El programa que reciba esta sucesión de bits debe conocer además cómo se disponen en el espacio los pixels, es decir, cuál es el ancho y el alto de la imagen; y exactamente cuántos bits codifican un pixel. Si esta información no está presente en el archivo que representa la imagen, su reconstrucción puede ser errónea o imposible.

Nuestro formato de imagen digital debe contener información **de dimensiones y de profundidad de color**, para poder ser comunicado efectivamente hacia otros programas o computadoras.

Un formato de imagen

Teniendo en cuenta todo lo anterior, podemos definir un formato de imagen como sigue. El formato de archivo de imagen tendrá una primera sección o **cabecera** con datos acerca de la imagen, o **metadatos**, y una segunda sección con los bits o datos de la imagen propiamente dichos.

- El primer byte de la cabecera del archivo se reserva para especificar el **ancho** de la imagen, es decir, cuántos pixels hay en cada fila.
- El segundo byte se reserva para especificar la **altura** o cantidad de filas de pixels de la imagen.
- El tercer byte especifica la profundidad de color, o cantidad de **bits por pixel**. Esta cantidad de bits por pixel define la cantidad de colores que se pueden codificar en la imagen. Si la imagen tiene n bits por pixel, hay 2^n posibilidades para el código de color y por lo tanto 2^n colores representables.
- Finalmente, el resto del archivo contiene los bits que representan a cada uno de los pixels por su color. Éstos son los datos de la imagen propiamente dicha.

Ejemplo

Un archivo que define una imagen de **cinco por cinco pixels, a cuatro colores**, comienza con los bytes 00000101, 00000101, 00000010, y sigue con los datos de la imagen.

Como la cantidad de datos binarios de un archivo en este formato es muy grande, para hacerlo más manejable usaremos notación hexadecimal. Entonces el archivo del ejemplo se representa por el hexadecimal 050502... y a continuación siguen en hexadecimal los códigos de color de los pixels.

Reconstruyendo una imagen

Para interpretar qué imagen describe un archivo dado, consideramos primero su cabecera y buscamos cuál es el ancho y el alto (indicados por los primeros dos bytes),

y cuántos bits por pixel están codificados en el resto del archivo (indicados por el tercer byte). De esta manera no es difícil dibujar la imagen.

Ejemplo

- Una imagen dada por la cadena hexadecimal **070401AEBF3...** tendrá 7×4 pixels, y un solo bit de paleta.
- Como la paleta se codifica con un solo bit, esta imagen es en blanco y negro (no puede haber más que dos valores de color).
- Los dígitos hexadecimales a partir de la cadena **AEBF3...** se analizan como grupos de cuatro bits y nos dicen cuáles pixels individuales están en negro (bits en 1) y en blanco (bits en 0).

Compresión de datos

Muchas veces es interesante reducir el tamaño de un archivo, para que ocupe menos espacio de almacenamiento o para que su transferencia a través de una red sea más rápida. Al ser todo archivo una secuencia de bytes, y por lo tanto de números, disponemos de métodos y herramientas matemáticas que permiten, en ciertas condiciones, reducir ese tamaño. La manipulación de los bytes de un archivo con este fin se conoce como **compresión**.

La compresión de un archivo se ejecuta mediante un programa que utiliza un algoritmo especial de compresión. Este algoritmo puede ser de **compresión sin pérdida**, o de **compresión con pérdida**.

Compresión sin pérdida

Decimos que la compresión ha sido **sin pérdida** cuando puede extraerse del archivo comprimido exactamente la misma información que antes de la compresión, utilizando otro algoritmo que ejecuta el trabajo inverso al de compresión. En otras palabras, la compresión sin pérdida es reversible: aplicando el algoritmo inverso, o de descompresión, siempre puede volverse a la información de partida. Esto es un requisito indispensable cuando necesitamos recuperar exactamente la secuencia de bytes original, como en el caso de un archivo de texto, una base de datos, una planilla de cálculo.

Como usuarios de computadoras, es muy probable que hayamos utilizado más de una vez la compresión sin pérdida, al tener que comprimir un documento de texto, utilizando un programa utilitario como ZIP, RAR u otros. Si la compresión no fuera reversible, no podríamos recuperar el archivo de texto tal cual fue escrito.

También somos usuarios de la compresión, muchas veces sin sospecharlo, al consultar páginas de Internet. Muchos sitios populares utilizan compresión para acelerar la descarga de sus contenidos. Los navegadores cuentan con el conocimiento para

identificar cuándo una página está comprimida, y saben descomprimirla en forma transparente, es decir, sin que el usuario necesite hacer ni saber nada.

Compresión con pérdida

Cuando la compresión se hace con una técnica **con pérdida**, no existe un algoritmo de descompresión que recupere la información original; es decir, no existe un algoritmo inverso.

El resultado de la compresión con pérdida de un archivo es otro archivo del cual ya no puede recuperarse la misma información original, pero que de alguna manera sigue sirviendo a los fines del usuario. La pérdida de información **es intencional**, y es el usuario quien ha elegido descartar esa información porque no es necesaria.

En el mundo analógico es frecuente la compresión con pérdida, por ejemplo en el caso de la compresión de audio, al descartar componentes del sonido con frecuencias muy bajas o muy altas, inaudibles para los humanos (como en la tecnología de grabación de CDs), con lo cual la diferencia entre el material original y el comprimido no es perceptible al oído. También es útil, para algunos fines, reducir la calidad del audio quitando algunos componentes audibles (lo que hacen, por ejemplo, los sistemas telefónicos, o algunos grabadores "de periodista" para lograr archivos más pequeños, con audio de menor fidelidad, pero donde el diálogo sigue siendo comprensible).

Al utilizar un servicio de *streaming* de video o audio, muchas veces se nos da la oportunidad de elegir una "calidad" menor del audio o del video, lo que quiere decir que el audio o la imagen se representarán con menos bits por segundo, y se transferirán a través de la red más rápidamente. Estas diferentes "calidades" menores son formas de compresión con pérdida.

Los estándares MP3 y MP4 son ejemplos de formatos de archivos digitales comprimidos con pérdida. Para comprimir con pérdida imágenes, se reduce su calidad, ya sea disminuyendo la resolución o utilizando menos colores.

Reducción de color

Si la imagen tiene $ancho \times alto$ pixels, y la información de color es de n bits por pixel, el archivo sin su cabecera mide $ancho \times alto \times n$ bits. Una forma sencilla de compresión con pérdida, que no modifica la resolución, es la reducción de la profundidad de color de una imagen. Si la imagen puede seguir siendo útil con menos colores, comprimiendo la paleta de colores puede obtenerse un archivo de menor tamaño.

Comprimir la paleta de colores consiste en reescribir la imagen con una cantidad menor de bits por pixel. Cada vez que la cantidad de bits por pixel decrece en uno, la profundidad de color, es decir, la cantidad de colores diferentes, se divide por dos. De esta forma se puede reducir la cantidad de bits utilizados para expresar cada pixel, claro está, al costo de perder información de color de la imagen.

Ejemplo

Sea una imagen a cuatro colores; luego la cantidad de bits por pixel es 2. Al reducir la profundidad de color, los colores 00 y 10 pasan a ser el único color 0; y los colores 10 y 11 pasan a ser el único color 1. Todos los pixels quedan expresados por un único bit 0 o 1, reduciendo efectivamente el tamaño de la imagen.

- La información ha sido **comprimida con pérdida** porque el archivo original no puede ser reconstruido a partir de este nuevo archivo.
- El nuevo archivo, sin su cabecera, mide $ancho \times alto \times (n-1)$, o sea, es $ancho \times alto$ bits más corto que el original.

Un método para reducir a la mitad la profundidad de color puede ser como sigue:

1. Escribir la tabla de códigos de color.
2. Retirar el bit más alto de cada código de color en la paleta.
3. Eliminar de la paleta los códigos duplicados.
4. Reescribir la cabecera del archivo manteniendo ancho y alto pero con la nueva cantidad de bits por pixel.
5. Reescribir los datos de la imagen reemplazando el código original de color de cada pixel por el nuevo código, es decir, quitando el bit más alto de cada pixel.

Dos pixels cuyos códigos de color diferían sólo en el bit de orden más alto ahora tendrán el mismo código, y por lo tanto se "pintarán" del mismo color. El archivo ya no contiene la información necesaria para saber cuál era el color original de cada pixel.

Ejemplo

Si el archivo está dado por la cadena hexadecimal **050502AEAFFAE8A600A8** (ancho: 5, alto: 5, bits por pixel: 2, pixels: 10 10 11 10 10 10 11 11 11 11 10 10 11 10 10 00 10 10 01 10 00 00 10 10 10), los pasos del procedimiento anterior son:

1. La tabla de códigos de color es {00 01 10 11}.
2. Sin su bit más alto, estos códigos son {0 1 0 1}.
3. Sin duplicados, quedan los códigos {0 1}.
4. La cabecera del nuevo archivo es {ancho: 5, alto: 5, bits por pixel: 1}.
5. Los bits que describen los pixels de la nueva imagen son {0 0 1 0 0 0 1 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 }.

La imagen comprimida queda como **05050123C82000** (ancho: 5, alto: 5, bits por pixel: 1, pixels: 0 0 1 0 0 0 1 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0).

Pregunta

- Se ha visto cómo reducir la profundidad de color en exactamente 1 bit. ¿Cómo podemos generalizar el método, para reducir la información de color en una cantidad de bits cualquiera?

Algoritmos de compresión sin pérdida

Aunque los programas que aplican algoritmos de compresión sin pérdida pueden ser muy sofisticados, algunas ideas básicas son muy sencillas.

Run Length Encoding o RLE

Supongamos tener una imagen en el formato que ya hemos descripto, y supongamos además que los datos de la imagen, es decir, la sucesión de bits que codifican los pixels, presentan grandes zonas de pixels con el mismo valor (muchos "1" seguidos, y muchos "0" seguidos). Si quisiéramos transmitir esta información por teléfono a alguien más, para que la imagen pudiera ser dibujada del otro lado, tarde o temprano la conversación incluiría frases como "...ahora cinco unos, ahora doce ceros...". Esta forma de descripción es mucho más económica, y menos propensa a errores.

Resulta natural abreviar la descripción de la imagen usando este tipo de expresiones, donde las cantidades funcionan como **coeficientes**. Un método inspirado directamente en esta idea se llama **Run Length Encoding (RLE)** o **Codificación por longitud de secuencia**. Una **secuencia** es una subsucesión de elementos del mismo valor. El método RLE identifica secuencias de elementos de un mismo valor, computa su longitud, y emite, en lugar de la secuencia, el coeficiente de longitud y el valor que corresponde.

Por supuesto, la efectividad de este método de compresión depende de la **redundancia** presente en el material original. Si no hay secuencias largas, el método no logrará compresión aceptable, e inclusive puede resultar contraproducente (el archivo final puede ser más largo que el original).

Para comprimir sin pérdida una pieza de información cualquiera con la técnica RLE o de Run Length Encoding, primeramente fijamos la cantidad de bits que ocuparán los coeficientes de longitud de secuencias. Esta cantidad de bits, ya que define el tamaño máximo de los coeficientes, debe ser elegida con cuidado:

- Si los coeficientes son pequeños, y la redundancia del archivo es muy alta, no aprovecharemos la capacidad de compresión del método.
- Si los coeficientes son muy grandes, y la imagen tiene poca redundancia, se desperdiciarán bits en los coeficientes y no lograremos buena compresión.

Ejemplo

- Si quisiéramos almacenar o transmitir un patrón de 253 "unos" seguidos de 119 "ceros" y luego 87 "unos", sin ninguna compresión, deberíamos manejar 458 bits. Si nuestra compresión utilizara la técnica RLE con ocho bits para el "coeficiente" y un bit para el valor repetido, bastaría con la secuencia binaria (11111101, 1, 01110111, 0, 01010111, 1) (en decimal 253, 1, 119, 0, 87, 1) que ocuparía tan sólo 27 bits.

Códigos de Huffmann o de longitud variable

La compresión sin pérdida por el método de Huffmann utiliza códigos de longitud variable. El método consiste esencialmente en examinar el archivo completo buscando subsecuencias de bits repetidas. Se computa la frecuencia, o cantidad de veces que aparece, para cada una de estas subsecuencias. Las subsecuencias se ordenan descendentemente por frecuencia, y cada una se reemplaza por un código instantáneo de bits de longitud creciente.

Por ejemplo, el carácter más frecuente será reemplazado por el código 1; el siguiente en frecuencia, por el código 01; el siguiente, por 001; etc. Así, los caracteres que aparecen más veces serán codificados por patrones de bits más cortos. De esta forma el archivo comprimido ocupará menos espacio que con un código de longitud uniforme.

Ejemplo

- El texto de once caracteres "ABRACADABRA" contiene cinco "A", dos "B", dos "R", una "C" y una "D". Si no utilizamos compresión, se necesitan $11 \times 8 = 88$ bits para representarlo. Si utilizamos compresión por códigos de longitud variable, crearemos un pequeño diccionario de la forma { A $\rightarrow$ 1, B $\rightarrow$ 01, R $\rightarrow$ 001, C $\rightarrow$ 0001, D $\rightarrow$ 00001 }. Con este diccionario, el texto se podrá representar como "1 01 001 1 0001 1 00001 1 01 001 1", en tan sólo 23 bits.

Compresión de imágenes con RLE

Fijada la cantidad de bits para coeficientes, la imagen se comprime indicando, para cada secuencia de pixels iguales, qué factor de repetición corresponde y qué valor de color llevan los pixels repetidos.

Ejemplo

La imagen con profundidad de color 2, cuyos datos de imagen son {10 10 11 11 11 11 11 11 11 10 10 10 10 10 10 00 ...}, tiene una secuencia de **dos pixels con valor 10**, **siete pixels con valor 11**, **seis pixels con valor 10**, etc.

Si utilizamos **tres bits para el coeficiente**, los coeficientes RLE **2, 7 y 6** se expresarán como **010, 111 y 110**. Los datos de la imagen se comprimirán como { 010 10 111 11 110 10... }. Los primeros treinta bits de los datos de imagen han quedado comprimidos a **quince** bits.

Compresión con pérdida y pérdida de información

Es importante insistir en el punto siguiente, que con frecuencia es mal comprendido.

Si un archivo es comprimido **sin pérdida** y luego transferido a través de la red, llega a destino un cierto conjunto de bits que, en algunos casos, puede contener errores.

El conjunto de bits puede tener valores intercambiados (ceros por unos) o estar incompleto. En estas condiciones, la descompresión o reconstrucción del archivo original no será posible, por pérdida de información. El programa que intente la descompresión fallará o entrará en una condición de error.

Se ha perdido información. Sin embargo, éste **no es un caso de compresión con pérdida**.

- La compresión **con pérdida** implica una pérdida de información que es **intencional**. La información ha sido quitada a propósito porque estaba de más, y no existe la intención de reconstruir el archivo original.
- Al comprimir **sin pérdida**, si existe pérdida de información, ésta ha sido **accidental**. La idea al comprimir era reconstruir el archivo en un momento posterior.

Arquitectura y Organización de Computadoras

¿Cómo **definimos** una computadora? ¿Cuándo un dispositivo es una computadora y cuándo decimos que no lo es? En esta unidad, vemos estos temas y estudiamos los diferentes componentes que tienen las computadoras.

Componentes de una computadora simple

En primer lugar, una **memoria principal**, que es donde se almacenan todos los datos y las instrucciones de programa. Todo lo que puede hacer la computadora, lo hace únicamente con contenidos que estén en la memoria. Para poder procesar un dato, primero hay que hacerlo llegar a la memoria principal, no importa de dónde venga. Un conjunto de datos puede estar en disco, en un pendrive, o ser introducido por el teclado, pero sólo cuando llega a la memoria principal es que puede ser procesado por la CPU.

La **CPU, o Unidad Central de Procesamiento**, es el componente que realmente lleva a cabo los cómputos. Trabaja leyendo instrucciones y datos de la memoria; y ejecuta esas instrucciones que operan sobre esos datos. Vamos a hablar mucho más sobre la CPU en esta unidad.

Los **dispositivos de Entrada y Salida** son todos aquellos dispositivos que conectamos a la computadora para hacer que el conjunto CPU + Memoria se comunique con el ambiente.

- Con dispositivos como el teclado, mouse, o tableta digitalizadora, podemos introducir datos. Son dispositivos de **entrada**.
- Con dispositivos como pantalla o impresora, podemos hacer que la computadora presente los resultados de los cómputos y los entregue al usuario. Son dispositivos de **salida**.

- Algunos dispositivos son de entrada y salida a la vez, como la tarjeta de red.

Todos estos componentes, y muchos otros que pueden estar o no presentes, dependiendo de la **arquitectura**, o modo de construcción, de la computadora, están conectados entre sí mediante **buses** o líneas de interconexión.

Memoria

La memoria es un componente fundamental de la computadora. Está implementada con circuitos que pueden estar en uno de dos estados eléctricos, y por esto los llamamos **biestables**.

Cada circuito biestable puede almacenar la información correspondiente a un **bit**. Los bits están agrupados de a ocho formando los **bytes**. Estos circuitos, con las tecnologías de hoy, están super miniaturizados y contienen muchos millones de posiciones donde se pueden almacenar temporariamente los datos y las instrucciones de programa.

Para poder utilizar la memoria es imprescindible conocer el número, o **dirección**, de la posición de memoria donde está el dato o instrucción que se necesita acceder. Con esta dirección podemos **recuperar**, es decir, **leer**, el valor que está alojado en ese byte de la memoria, o escribir sobre ese byte un contenido de ocho bits.

CPU

La CPU está implementada como un circuito sumamente complejo que contiene **registros**. Éstos son lugares de almacenamiento temporario de datos e instrucciones que se utilizan durante el cómputo.

Si, en un momento dado, sacamos una foto instantánea de una CPU, sus registros tendrán un cierto conjunto de valores. Ese conjunto de valores se llama el **estado** de la CPU. La CPU, mientras opera, va cambiando de estado, es decir, paso a paso va modificando los valores de sus registros hasta llegar a un resultado de cada instrucción.

Por atravesar esta sucesión de cambios de estado, se dice que una CPU es un circuito **secuencial**. Los cambios de estado son disparados por un **reloj**, que es un circuito auxiliar que produce pulsos o impulsos eléctricos que hacen marchar a la CPU.

La CPU puede interpretar un conjunto determinado de instrucciones. Estas instrucciones han sido definidas en su arquitectura; y son las únicas que puede ejecutar. El funcionamiento de la CPU está limitado a estas instrucciones, que son muy básicas, como sumar dos datos, o leer el dato que está en una determinada dirección de la memoria.

Sin embargo, cuando escribimos una secuencia de instrucciones en la memoria, es decir, un **programa**, podemos hacer que la CPU desarrolle otras tareas que no

estaban previstas en su arquitectura. Por ejemplo, podemos tener una CPU que no sepa multiplicar o dividir, pero si cuenta con un programa con las instrucciones adecuadas, puede ejecutar operaciones de multiplicación o división de datos.

La CPU está constituida por varios circuitos componentes o unidades funcionales, como la **Unidad de Control** y la **Unidad Lógico-Aritmética**.

- La Unidad de Control es la que contiene la lógica necesaria para **leer** cada instrucción del programa que está en memoria, **ejecutarla**, y pasar a la **siguiente** instrucción. Esta lógica se llama el **ciclo de instrucción** y se repite **continuamente** mientras la CPU está funcionando.
- La Unidad Lógico-Aritmética es la que efectivamente realiza los cómputos con los datos.

Arquitectura de Von Neumann

Máquina de programa almacenado

Por supuesto, además de todos esos componentes que hemos nombrado, hay muchas otras cosas que físicamente forman parte de la computadora; pero la descripción de la computadora que hemos hecho hasta el momento dice, por lo menos en líneas generales, los componentes fundamentales que tiene cualquier computadora actual. Esta descripción puede resumirse diciendo que la computadora es una **máquina de Von Neumann o máquina de programa almacenado**.

En esta clase de máquinas, existe una memoria; que contiene instrucciones y datos; que como contenidos de esa memoria, no se diferencian, salvo por la forma como son utilizados.

Estas máquinas ejecutan las instrucciones almacenadas en memoria secuencialmente, es decir, procediendo desde las direcciones inferiores de la memoria hacia las superiores, leyendo y ejecutando cada instrucción y pasando a la siguiente.

CPU y Memoria

En una **máquina de Von Neumann**, entonces, aparecen dos componentes básicos fundamentales que son la CPU y la memoria,

la primera conteniendo una **Unidad de Control, o UC**, para realizar el **ciclo de instrucción**,

y una **Unidad Lógico-Aritmética, o ALU**, para el cómputo.

Buses

En la máquina existen diferentes clases de buses para interconectar los componentes:

- **buses internos** de la CPU para comunicar la UC y la ALU,
- **buses de sistema** que relacionan la CPU y la memoria,
- y otros **buses de Entrada/Salida** para comunicar todo el sistema con los dispositivos de entrada o de salida.

¿En qué momento se utilizará cada clase de bus?

- Cuando la UC disponga que se debe ejecutar una instrucción, tal como una suma, enviará los datos de partida, y la instrucción, a la ALU a través de un bus interno.
- Si la ALU necesita más datos, los obtendrá de la memoria a través de un bus de sistema.
- Si la CPU encuentra instrucciones que ordenan presentar el resultado del cómputo al usuario, usará un bus de Entrada/Salida para emitir ese resultado por pantalla o por impresora.

Modelo Computacional Binario Elemental

Para comprender desde lo más básico cómo opera la computadora, recurrimos al **MCBE o Modelo Computacional Binario Elemental**, que es una máquina teórica. El MCBE es una computadora extremadamente simple, pero que podría ser implementada físicamente, y funcionaría como la mayoría de las computadoras actuales. Bueno, con muchas menos capacidades, claro, pero manteniendo lo esencial de las computadoras de programa almacenado.

Esquema del MCBE

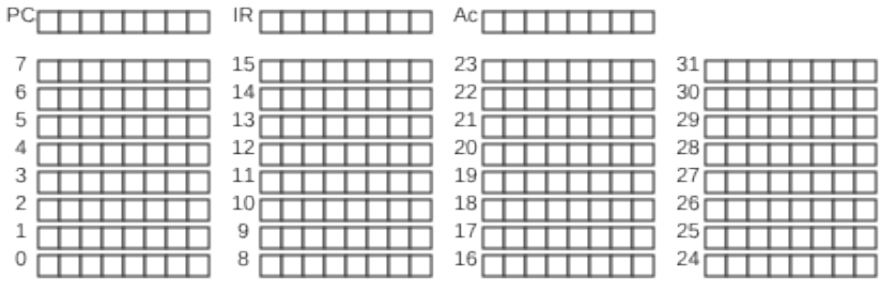

Figura 1: Esquema del MCBE

En este esquema del MCBE vemos los tres registros de la CPU: el **PC o contador de programa**, el **IR o registro de instrucción**, situados en la Unidad de Control,

y el **Acumulador**, situado en la Unidad Lógico-aritmética. Los tres son registros de ocho bits.

Además se representa **la memoria**, compuesta por 32 bytes de ocho bits. Las direcciones de los bytes van, entonces, **de 0 a 31**. Aquí hemos descompuesto la memoria en trozos solamente para poder representarla completa en el esquema, pero conviene pensar en la memoria como una única sucesión de bytes, numerados de 0 a 31. Es costumbre, al representar los diagramas de memoria, ubicar las posiciones con direcciones menores en la parte inferior del diagrama, y las direcciones mayores arriba; como si la memoria fuera una escalera con posiciones numeradas a partir de 0 y ascendiendo.

Estado de la máquina

Cada combinación posible de ceros y unos en los bits de cualquiera de los registros o posiciones de memoria representa un **estado** de la máquina, porque define los valores de la memoria y de los registros en un momento dado. Es decir, el estado de la máquina en cada momento es el conjunto de valores de los tres registros y de los 32 bytes de la memoria.

El estado de la máquina en cada momento define cuál será el estado siguiente. Ninguna otra cosa interviene en el comportamiento de la máquina. En particular, la máquina no tiene voluntad ni toma decisiones propias: solamente cumple el **ciclo de instrucción**, que la hace ejecutar las instrucciones del programa que tenga almacenado.

Memoria del MCBE

Las 32 posiciones de memoria, cada una de 8 bits, son casi todas iguales, con dos excepciones.

- La posición 30 (casi al final de la memoria) está reservada para comunicación con dispositivos de entrada. Es sólo de lectura, es decir, no se pueden escribir contenidos en esa dirección. Cuando leemos esa posición de memoria, la máquina detiene su programa y espera que el usuario introduzca un dato. Una vez que el usuario ha terminado de escribirlo, el MCBE continúa la operación del programa con el dato recién leído.
- La posición 31 es solamente de escritura. Al escribir un dato en la dirección 31, hacemos que el MCBE escriba ese valor por pantalla, y solamente así podemos saber el resultado de un cómputo.

Registros del MCBE

Como hemos dicho, los registros son lugares de almacenamiento temporario para varios usos. Los registros funcionan en forma parecida a la memoria, en el sentido de

que se pueden leer o escribir datos en ellos, pero normalmente en una computadora física su velocidad de acceso es mayor.

Los registros del MCBE tienen diferentes funciones. El registro **PC, o contador de programa,** contiene en cada momento la dirección de la próxima instrucción que se va a ejecutar; es decir, contiene un número que es la dirección de la posición de memoria donde está almacenada la instrucción que está por ejecutar el MCBE.

Antes de ejecutar cada instrucción, la CPU va a **copiarla** en el registro **IR, o registro de instrucción**; y mientras está almacenada allí la instrucción, va a ser decodificada, es decir, la CPU va a interpretar de qué instrucción se trata, y la va a ejecutar.

El registro **acumulador**, que pertenece a la ALU, es un registro que interviene en casi todas las operaciones del MCBE; sobre todo para las operaciones aritméticas.

CPU del MCBE

Entonces la CPU del MCBE queda definida como el conjunto de **Unidad de Control** con dos registros **PC e IR**, más **Unidad lógico-aritmética** con un registro **acumulador**.

Esta CPU va a ser muy limitada y solamente va a ejecutar operaciones de suma y resta en complemento a 2, con ocho bits. No va a ejecutar **multiplicaciones**, ni **divisiones**, ni operaciones en **punto flotante**.

Formato de instrucciones del MCBE

Código de instrucción Las instrucciones del MCBE se codifican en ocho bits y por lo tanto pueden ser almacenadas en un byte de la memoria, o en el registro IR. Cada instrucción se divide en dos partes: los tres bits de más a la izquierda se destinan a representar el **código de instrucción**.

Argumentos u operandos Los cinco bits restantes representan el **argumento u operando** de esa instrucción, es decir, el dato con el cual tiene que operar esa instrucción.

Direcciones y desplazamientos Por otro lado, los operandos pueden ser de dos clases: o bien **direcciones**, o bien **desplazamientos**. Si el operando es una dirección, es porque la CPU necesita conocer la dirección de un dato. Si es un desplazamiento, ese desplazamiento es **una cantidad de posiciones** que hay que trasladarse en la memoria, para encontrar la siguiente instrucción que hay que procesar.

- Cuando el operando es una dirección, los cinco bits del operando representan una cantidad sin signo (porque no pueden existir direcciones negativas).
- Cuando es un desplazamiento, esos cinco bits son **con signo**, y más precisamente, en complemento a 2; porque el desplazamiento puede ser **negativo**, indicando que hay que **volver hacia atrás, a una cierta dirección de la memoria** a ejecutar una instrucción que tal vez ya fue ejecutada.

Notemos que al representar las direcciones con cinco bits, sin signo, tenemos un rango de representación de 0 a 31, justo lo que necesitamos para alcanzar todas las posiciones de memoria.

Para los desplazamientos, como estamos usando un sistema con signo, tenemos un rango de -16 a 15. Lo que quiere decir que al trasladarnos de un lugar a otro de la memoria, vamos a poder hacerlo en saltos de a lo sumo 16 bytes hacia atrás o 15 bytes hacia adelante.

Conjunto de instrucciones del MCBE

Las instrucciones del MCBE se dividen en cuatro grupos.

- Las instrucciones de transferencia de datos son las que leen o escriben datos en la memoria.
- Las aritméticas son las que operan entre esos datos de la memoria y el valor presente en el registro acumulador.
- Las de salto o transferencia de control, las que desvían, derivan o trasladan la ejecución a otra posición de memoria.
- Y las de control, completan el funcionamiento de la máquina, por ejemplo controlando cuándo va a detenerse el programa.

Notemos que, como tenemos un campo de **tres bits** para definir el código de instrucción, no vamos a poder tener más que **ocho instrucciones**. Precisamente hay dos instrucciones en cada grupo de estos cuatro que hemos definido.

Instrucciones de transferencia de datos

Las instrucciones de transferencia de datos son dos. El código 010 copia un byte de la memoria hacia el acumulador. Para esto se necesita la **dirección** de ese byte, y esa dirección es precisamente el argumento u operando de la instrucción, y por lo tanto esa dirección está codificada en los cinco bits de operando de la instrucción.

El código 011 es la operación inversa, es decir, copia el contenido del registro acumulador en una posición de memoria. La dirección de esa posición está, también, determinada por los cinco bits de operando.

En cualquiera de los dos casos, luego de ejecutarse la instrucción, el valor del PC queda valiendo 1 más de lo que valía antes, es decir, se incrementa en 1. Esto permite que el ciclo de instrucción pase a la instrucción siguiente.

El efecto sobre el estado de la máquina es exactamente lo que se describe aquí: cambia el valor del acumulador, en el caso de la instrucción 010, o el valor de una posición de memoria, en el caso del código 011, y el valor del PC se incrementa en

1. No ocurre ningún otro cambio en el estado de la máquina, ni en los registros ni en la memoria.

Instrucciones aritméticas

Las instrucciones aritméticas también son dos. Si el código es 100, la CPU va a buscar el valor contenido en la dirección dada por el operando, y lo suma al acumulador. Es decir, se suman el valor que tuviera anteriormente el acumulador, con el valor que proviene de esa posición de memoria. El resultado queda en el acumulador. El valor de la posición de memoria no varía.

Si el código es 101, la operación es una resta, que como sabemos consiste en complementar a 2 el operando y luego sumar. El resultado, igual que en la instrucción de suma, queda en el acumulador, y la posición de memoria queda sin cambios. Como en las instrucciones de transferencia de datos, el registro PC se incrementa en 1. Decimos que el PC **queda apuntando** a la siguiente instrucción.

Instrucciones de salto

Las dos instrucciones **de salto, o de transferencia de control**, tienen un efecto diferente. Funcionan modificando exclusivamente el valor del registro PC. En ambos casos, el operando es un valor con signo a cinco bits. En el caso del código 110, ese valor se suma algebraicamente al valor que tuviera hasta el momento el registro PC. Con lo cual el PC queda apuntando a alguna posición de memoria por delante o por detrás de donde estaba antes. Así, el ciclo de instrucción siguiente va a leer la instrucción ubicada en esa nueva dirección que ahora está contenida en el PC.

El código 110 es una instrucción de salto **incondicional**, es decir, **siempre** provoca un **salto** en la ejecución. En el caso del código 111, el salto es **condicional, y depende del valor del registro acumulador**. Si el acumulador **tiene un valor 0**, se produce el salto, sumando el valor del desplazamiento al registro PC. Pero si el acumulador no vale cero, simplemente se incrementa el PC en 1 como en el resto de las instrucciones; y el control sigue secuencialmente como es normal.

Otras instrucciones

Hasta el momento no hemos explicado cómo se detiene la máquina. El ciclo de instrucción continuamente va a ejecutar la instrucción siguiente, sin parar nunca. Para terminar la ejecución usamos la instrucción 001. El programa se detiene, y el estado final de la máquina queda con los valores que recibieron por última vez los registros y la memoria. El valor del PC no cambia.

La operación 000 no tiene ningún efecto sobre el estado del MCBE, salvo incrementar el PC. Ningún otro registro ni posición de memoria cambia su valor.

Ciclo de instrucción

Ahora podemos definir con más rigurosidad lo que se entiende por **ciclo de instrucción**. El MCBE inicia su operación con todos los contenidos de la memoria y

registros en 0, y se pone a ejecutar continuamente el ciclo de instrucción. Para esto repite continuamente las fases siguientes.

1. Copia en el registro IR la instrucción cuya dirección está en el PC. Como el PC comienza con un valor 0, esto significa copiar la instrucción almacenada en la dirección 0 hacia el IR.
2. Decodifica la instrucción, lo que significa separar la instrucción en sus dos componentes, que son el código de operación, de tres bits, y el operando o argumento, de cinco bits.
3. Se ejecuta la instrucción, lo que significa que va a haber algún efecto sobre el estado de la máquina. Si es una instrucción de transferencia de datos, cambiará el registro acumulador o alguna posición de memoria; si es una instrucción aritmética, cambiará el valor del registro acumulador; si es de transferencia de control, cambiará el valor del PC, etc.
4. Una vez ejecutada la instrucción, se vuelve a repetir el ciclo, leyendo la siguiente instrucción que haya que ejecutar, que será aquella cuya dirección esté contenida en el PC.

Programación del MCBE

¿Cómo es, entonces, un programa para esta máquina teórica? Es una sucesión de bytes, que representan instrucciones y datos, contenidos en la memoria a partir de la dirección 0, y donde cada byte va a ser interpretado como instrucción o como dato según lo diga el programa. Como el estado inicial de la máquina es **con todos los valores en 0**, lo único que puede decirse con seguridad es que **la primera posición de la memoria contiene una instrucción**. Pero a partir de allí, el desarrollo de la ejecución va a ser dado por las instrucciones particulares que contenga el programa.

Dirección	Contenido
00000	01000110
00001	10000111
00010	01101000
00011	00100000
00100	
00101	
00110	00001100
00111	00000001
01000	

- En este programa en particular, la primera instrucción es 010 00110, que es una instrucción de transferencia de datos de la posición 6 al acumulador.
- La segunda instrucción es 100 00111, que es una suma del valor que haya en la posición 7 al acumulador.
- La tercera instrucción es 011 01000, que significa transferir el valor que haya en el acumulador a la posición 8.
- Y la cuarta instrucción es 001 00000, que es la instrucción de parada, con lo cual termina el programa.
- Todas éstas eran las instrucciones del programa. En las posiciones 6 y 7 de la memoria tenemos datos almacenados en complemento a 2 sobre ocho bits. Estos datos son el número 12, en la posición 6, y el número 1 en la posición 7.
- En las restantes posiciones de memoria hay contenidos nulos, o sea, todos los bits en 0, y no los escribimos para no complicar más el diagrama.

Traza de ejecución

¿Qué es realmente lo que hace este programa, y cuál es el resultado de ejecutarlo? Para poder saberlo, lo más conveniente es hacer una **traza del programa**. Una traza es un diagrama o planilla donde preparamos **columnas** con los nombres de los **registros, la memoria y la salida**, para poder ir simulando manualmente la ejecución, e ir anotando qué valores toman esos registros; es decir, cuáles son los sucesivos estados del MCBE.

Para cada instrucción que se ejecute usaremos un renglón de la planilla.

En la traza solamente escribiremos los elementos del estado **que se modifiquen** en cada paso. En la columna de MEMORIA anotaremos cuándo hay una operación de escritura en memoria, en la columna de SALIDA anotaremos cuando haya un contenido que se escriba en pantalla, etc.

Este programa en particular presenta la traza que veremos a continuación.

Queda como ejercicio seguir la traza e interpretar qué está ocurriendo en cada momento con cada uno de los registros y las posiciones de memoria.

Dirección	Contenido
00000	01000110
00001	10000111
00010	01101000
00011	00100000
00100	
00101	
00110	00001100

Dirección	Contenido
00111	00000001
01000	00001101

Ayuda

- 000 No operación
- 001 Parada
- 010 Mem $\rightarrow$ Ac
- 011 Ac $\rightarrow$ Mem
- 100 Sumar al Ac
- 101 Restar al Ac
- 110 Salto
- 111 Salto cond

Preguntas

1. El MCBE, ¿puede encontrar una instrucción que no sea capaz de decodificar?
2. Supongamos que hemos almacenado en la posición 14 un dato numérico que representa la edad de una persona. ¿Qué pasa si en algún momento de la ejecución el PC contiene el número 14? ¿Qué pasa si esa persona tiene 33 años?
3. ¿Podría aumentarse la capacidad de memoria del MCBE? ¿Esto requeriría algún cambio adicional a la máquina?

Proponemos como ejercicio **examinar** las siguientes frases, tomadas de exámenes de la materia, a ver si descubrimos qué está mal en cada una de ellas.

1. El primer paso del ciclo de instrucción es cargar el IR en el PC.
2. Lo que hacen las instrucciones de salto es cambiar el efecto de las instrucciones en los registros del MCBE.
3. Las instrucciones de salto sirven como desplazamiento de instrucciones y cambian el orden de los registros.
4. La instrucción de salto incondicional es un desplazamiento sin signo, la de salto condicional es un desplazamiento con signo.
5. Las instrucciones de salto copian el contenido de la dirección en el acumulador.

El Software

En esta parte de la unidad, **El Software**, nos interesa conocer el proceso de desarrollo de software, desde el punto de vista de la organización de computadoras. Explicaremos cómo se llega desde un programa, en un lenguaje de alto o bajo nivel, a obtener una sucesión de instrucciones de máquina para un procesador.

Lenguajes de bajo nivel

Lenguaje de máquina o código máquina

Hemos visto un conjunto de instrucciones y convenciones sobre cómo se utilizan los datos en el MCBE, que es el llamado **lenguaje de máquina** del MCBE. En este lenguaje, las operaciones y los datos se escriben como secuencias de dígitos binarios.

Por supuesto, escribir un programa para el MCBE y **depurarlo**, es decir, identificar y corregir sus errores, es una tarea muy dificultosa, porque los códigos de operación, las direcciones y los datos, fácilmente terminan confundiéndonos. Para facilitar la programación, se ha definido un lenguaje alternativo llamado el **ensamblador** del MCBE.

Lenguaje ensamblador

Cuando escribimos un programa en el lenguaje **ensamblador** del MCBE, las instrucciones se corresponden una a una con las del programa en lenguaje de máquina. Pero en el lenguaje ensamblador del MCBE:

- En lugar de códigos de tres bits usamos unas abreviaturas un poco más significativas (llamadas los **mnemónicos** de las instrucciones).
- En lugar de direcciones de cinco bits para los datos, usamos unos nombres simbólicos (**rótulos o etiquetas**) que hacen referencia a esas direcciones.
- Para las instrucciones de salto, en lugar de desplazamientos, también usamos rótulos o etiquetas para indicar la instrucción del programa adonde deseamos saltar.

Cada CPU del mundo real tiene su propio lenguaje de máquina, y aunque mucho más poderosos y de instrucciones más complejas, se parecen bastante, en líneas generales, al lenguaje de máquina del MCBE. Igual que ocurre con el lenguaje de máquina, cada CPU del mundo real tiene su propio lenguaje ensamblador, basado en los mismos principios que el que mostramos aquí.

El lenguaje de máquina de cualquier CPU, y su lenguaje ensamblador (o *Assembler*), son llamados en general **lenguajes de bajo nivel**.

Mnemónicos

Los **mnemónicos** o nombres simbólicos de las instrucciones se basan en los nombres en inglés de las operaciones correspondientes. Disponemos de los mnemónicos:

- LD para la operación de cargar el Acumulador con un contenido de memoria (código 010), y ST para la operación inversa (código 011).
- ADD para la operación de suma (código 100) y SUB para la resta (código 101).
- JMP y JZ para los saltos incondicional y condicional (códigos 110 y 111), respectivamente.
- HLT para la instrucción de parada (código 001) y NOP para la operación nula o no operación (código 000).

Rótulos

Cuando necesitamos hacer referencia a una dirección, como en las operaciones de transferencia o en las aritméticas, el ensamblador nos permite independizarnos del valor de esa dirección y simplemente indicar un **nombre simbólico o rótulo** para esa dirección. Así, un rótulo equivale en lenguaje ensamblador a la **dirección de un dato**.

Para que el programa quede completo, ese nombre simbólico debe aparecer en algún lugar del programa, al principio de la instrucción, y separado por un carácter ":" del resto de la línea.

Ejemplo

Dirección	Instrucción	Rótulo	Mnemónico	Argumento
00000	01000111		LD	CANT
00001	11100100	SIGUE:	JZ	FIN
00010	01111111		ST	OUT
00011	10100110		SUB	UNO
00100	11011101		JMP	SIGUE
00101	00100000	FIN:	HLT	
00110	00000001	UNO:	1	
00111	00000011	CANT:	3	

En este ejemplo, SIGUE, FIN, UNO y CANT son rótulos. El rótulo CANT, por ejemplo, nos permite referirnos en la primera instrucción, LD CANT, a un dato

declarado más adelante con ese nombre. Del mismo modo, cuando la instrucción es de salto, podemos hacer referencia a la posición de memoria donde se hará el salto usando un rótulo, como en la quinta instrucción, JMP SIGUE.

Es importante recordar que, de todas maneras, en la traducción de ensamblador a lenguaje de máquina **para las instrucciones de salto**, el rótulo se sustituye por un **desplazamiento**, y no por una dirección.

Ejemplo

- En el ejemplo existe un rótulo SIGUE que identifica a la instrucción en la posición 1. La instrucción del ejemplo JMP SIGUE, al ser ejecutada, deriva el control a la instrucción 1. Es decir, almacena un 1 en el registro PC, para que la siguiente iteración del ciclo de instrucción ejecute la instrucción en la dirección 1 de la memoria. Sin embargo, el argumento para la instrucción JMP **no vale 1** sino **-3**, como podemos corroborar en la columna "Instrucción" de la tabla.

Rótulos predefinidos

Los rótulos IN y OUT vienen predefinidos en el lenguaje ensamblador de MCBE y corresponden a las posiciones de memoria 30 (para entrada) y 31 (para salida) respectivamente.

Ejemplo

La instrucción en línea 2, ST OUT, almacena el contenido del acumulador en la posición 31, lo que equivale a escribir ese contenido en la salida del MCBE.

Traductores

Uno puede pensar en un programa cualquiera como si se tratara de una máquina, cuyo funcionamiento es, en principio, desconocido. Todo lo que vemos es que, si introducimos ciertos datos, de alguna forma esta "máquina" devolverá un resultado.

Si pensamos en una clase especial de estos programas, donde los datos que ingresan son a su vez un programa, y donde la salida devuelta por la máquina es, a su vez, un programa, entonces esa clase especial de programas son los **traductores**.

Ensambladores

Como hemos dicho anteriormente, una CPU como el MCBE sólo sabe ejecutar instrucciones de código máquina expresadas con unos y ceros. Cuando vimos el lenguaje ensamblador del MCBE lo propusimos simplemente como una forma de abreviar las instrucciones de máquina, o como una forma de facilitar la escritura,

porque los mnemónicos y rótulos eran más fáciles de memorizar y de leer que las instrucciones con unos y ceros.

Sin embargo, un programa escrito en ensamblador del MCBE podría ser traducido automáticamente, por un traductor, a código de máquina MCBE, ahorrándonos mucho trabajo y errores.

Esta clase de traductores, que reciben un programa en lenguaje ensamblador y devuelven un programa en código de máquina, son los llamados **ensambladores** o **assemblers**.

Ensamblador x86

Cada CPU tiene su propio lenguaje ensamblador, y existen programas traductores (ensambladores) para cada una de ellas. Por ejemplo, la familia de procesadores de Intel para computadoras personales comparte el mismo ISA, o arquitectura y conjunto de instrucciones. Cualquiera de estos procesadores puede ser programado usando un ensamblador para la familia **x86**.

Según la tradición, el primer programa que uno debe intentar escribir cuando comienza a aprender un lenguaje de programación nuevo es "Hola mundo". Es un programa que simplemente escribe esas palabras por pantalla. Aquí mostramos el clásico ejemplo de "Hola mundo" en el lenguaje ensamblador de la familia x86.

```
.globl  _start
.text               # seccion de codigo
_start:
        movl    $len, %edx  # carga parametros longitud
        movl    $msg, %ecx  # y direccion del mensaje
        movl    $1, %ebx        # parametro 1: stdout
        movl    $4, %eax        # servicio 4: write
        int     $0x80       # syscall

        movl    $0, %ebx        # retorna 0
        movl    $1, %eax        # servicio 1: retorno de llamada
        int     $0x80       # syscall
.data               # seccion de datos
msg:
        .ascii   "Hola, mundo!\n"
        len =   . - msg     # longitud del mensaje
```

Los procesadores de la familia x86 se encuentran en casi todas las computadoras personales y notebooks.

Ensamblador ARM

Por supuesto, los procesadores de familias diferentes tienen conjuntos de instrucciones diferentes. Así, un lenguaje y un programa ensamblador están ligados a un procesador determinado. El código máquina producido por un ensamblador no puede ser trasladado sin cambios a otro procesador que no sea aquel para el cual fue ensamblado. Las instrucciones de máquina tendrán sentidos completamente diferentes para uno y otro.

Por eso, el código máquina producido por un ensamblador para x86 no puede ser trasladado directamente a una computadora basada en un procesador como, por ejemplo, ARM; sino que el programa original, en ensamblador, debería ser **portado** o traducido al ensamblador propio de ARM, por un programador, y luego ensamblado con un ensamblador para ARM.

```
.global main
main:
    @ Guarda la direccion de retorno lr
    @ mas 8 bytes para alineacion
    push    {ip, lr}
    @ Carga la direccion de la cadena y llama syscall
    ldr     r0, =hola
    bl      printf
    @ Retorna 0
    mov     r0, #0
    @ Desapila el registro ip y guarda
    @ el siguiente valor desapilado en el pc
    pop     {ip, pc}
hola:
    .asciz "Hola, mundo!\n"
```

El ARM es un procesador que suele encontrarse en plataformas móviles como *tablets* o teléfonos celulares, porque ha sido diseñado para minimizar el consumo de energía, una característica que lo hace ideal para construir esos productos portátiles. Su arquitectura, y por lo tanto, su conjunto de instrucciones, están basados en esos principios de diseño.

Ensamblador Power PC

Lo mismo ocurre con otras familias de procesadores como el Power PC, un procesador que fue utilizado para algunas generaciones de consolas de juegos, como la PlayStation 3.

```
.data             # seccion de variables
msg:
```

```
    .string "Hola, mundo!\n"
    len = . - msg     # longitud de cadena
.text             # seccion de codigo
    .global _start
_start:
    li  0,4        # syscall sys_write
    li  3,1    # 1er arg: desc archivo (stdout)
               # 2do arg: puntero a mensaje
    lis     4,msg@ha   # carga 16b mas altos de &msg
    addi    4,4, msg@l # carga 16b mas bajos de &msg
    li  5,len      # 3er arg: longitud de mensaje
    sc         # llamada al kernel
#
    li  0,1    # syscall sys_exit
    li  3,1    # 1er arg: exit code
    sc         # llamada al kernel
```

Lenguajes de programación

Lenguajes de bajo nivel

Como vemos, tanto el lenguaje de máquina como el ensamblador o **Assembler** son lenguajes **orientados a la máquina**. Ofrecen control total sobre lo que puede hacerse con un procesador o con el sistema construido alrededor de ese procesador. Por este motivo son elegidos para proyectos de software donde se necesita dialogar estrechamente con el hardware, como ocurre con los sistemas operativos.

Sin embargo, como están ligados a un procesador determinado, requieren conocimiento profundo de dicho procesador y resultan poco **portables**. Escribir un programa para resolver un problema complejo en un lenguaje de bajo nivel suele ser muy costoso en tiempo y esfuerzo.

Lenguajes de alto nivel

Otros lenguajes, los de **alto nivel**, ocultan al usuario los detalles de la arquitectura de las computadoras y le facilitan la programación de problemas de software complejos. Son más **orientados al problema**, lo que quiere decir que nos aíslan de cómo funcionan los procesadores o de cómo se escriben las instrucciones de máquina, y nos permiten especificar las operaciones que necesitamos para resolver nuestro problema en forma más parecida al lenguaje natural, matemático, o humano.

Una ventaja adicional de los lenguajes de alto nivel es que resultan más portables, y su **depuración** (el proceso de corregir errores de programación) es normalmente más fácil.

Niveles de lenguajes

Se han diseñado muchísimos lenguajes de programación. Cada uno de ellos es más apto para alguna clase de tareas de programación y cada uno tiene sus aplicaciones.

- Entre estos lenguajes, que podemos organizar en una jerarquía, encontramos los de bajo nivel u orientados a la máquina, los de alto nivel, u orientados al problema, y algunos en una zona intermedia.
- Pero también pueden clasificarse por el **paradigma de programación** al cual pertenecen.
- Además, algunos son habitualmente lenguajes **compilables**, y otros, **interpretables**.

Paradigmas de programación

La programación en lenguajes de alto nivel puede adoptar varias formas. Existen diferentes modos de diseñar un lenguaje, y varios modos de trabajar para obtener los resultados que necesita el programador. Esos modos de pensar o trabajar se llaman **paradigmas de lenguajes de programación**.

Hay al menos cuatro paradigmas reconocidos, que son, aproximadamente en orden histórico de aparición, **imperativo** o procedural, **lógico o declarativo**, **funcional** y **orientado a objetos**. Los paradigmas lógico y funcional son los más asociados a la disciplina de la Inteligencia Artificial.

Paradigma imperativo o procedural

Bajo el paradigma imperativo, los programas consisten en una sucesión de instrucciones o comandos, como si el programador diera órdenes a alguien que las cumpliera. El ejemplo en lenguaje **C** explica cuáles son las órdenes que deben ejecutarse, una por una.

```
int factorial(int n)
{
        int f = 1;
        while (n > 1) {
                f *= n;
                n--;
        }
        return f;
}
```

Paradigma lógico o declarativo

El lenguaje Prolog representa el paradigma lógico y con frecuencia constituye el corazón de los sistemas de Inteligencia Artificial que realizan **razonamiento**.

La definición de **factorial** en lenguaje Prolog que mostramos se compone de un hecho y dos reglas. El hecho consiste en que el **factorial** de 0 vale 1. La primera regla expresa que el factorial de un número **N** se calcula como el factorial de **N-1** multiplicado por N. Es una definición **recursiva** porque la definición de la regla se utiliza a sí misma.

```
factorial(0,X):- X=1.
factorial(N,X):- N1=N-1, factorial(N1,X1), X=X1*N.
factorial(N):- factorial(N,X), write(X).
```

El usuario de este programa puede usarlo de dos maneras. Podría preguntar el valor del factorial de un número N, o consultar si es cierto que el factorial de N es otro número dado Y.

Paradigma funcional

En el lenguaje Lisp, perteneciente al paradigma funcional, una función es un enunciado entre paréntesis que puede contener a otras funciones. En particular la definición de **factorial** presentada aquí contiene a su vez una invocación de la misma función, volviéndola una función **recursiva**.

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))) ) )
```

El lenguaje Lisp utiliza notación prefija para los operadores.

Orientación a objetos

En un lenguaje **orientado a objetos**, definimos una **clase** que funciona como un molde para crear múltiples instancias de objetos que se parecen entre sí, ya que tienen los mismos datos que los componen y la misma funcionalidad. Los objetos creados se comunican entre sí por **mensajes**, disparando **métodos** o conductas de otros objetos.

```
class Combinatoria():
    def factorial(self,n):
```

```
        num = 1
        while n > 1:
            num = num * n
            n = n - 1
        return num

c = Combinatoria()
a = c.factorial(3)
print a
```

En el ejemplo de programación orientada a objetos en Python, definimos una clase **Combinatoria** que producirá objetos con la conducta **factorial**. El programa crea un objeto, instancia de la clase Combinatoria, llamado **c**, al cual se le envía el mensaje **factorial**, que dispara la conducta correspondiente especificada en el método del mismo nombre. Finalmente se imprime su valor.

Compiladores e intérpretes

Los traductores de lenguajes de alto nivel pueden funcionar de dos maneras: o bien producen una versión en código máquina del programa fuente (**compiladores**) o bien analizan instrucción por instrucción del programa fuente y además de generar una traducción a código máquina de cada línea, la ejecutan (**intérpretes**).

Luego de la compilación, el programa en código máquina obtenido puede ser ejecutado muchas veces. En cambio, el programa interpretado debe ser traducido cada vez que se ejecute.

Una ventaja comparativa de la compilación respecto de la interpretación es la mayor velocidad de ejecución. Al separar las fases de traducción y ejecución, un compilador alcanza la máxima velocidad de ejecución posible en un procesador dado. Por el contrario, un intérprete alterna las fases de traducción y ejecución, por lo cual la ejecución completa del mismo programa tardará algo más de tiempo.

Inversamente, el código interpretado presenta la ventaja de ser directamente portable. Dos plataformas diferentes podrán ejecutar el mismo programa interpretable, siempre que cuenten con intérpretes para el mismo lenguaje. Por el contrario, un programa compilado está en código máquina para alguna arquitectura específica, así que no será compatible con otras.

Ciclo de compilación

Terminología

- Cuando utilizamos un **compilador** para obtener un programa **ejecutable**, el programa que nosotros escribimos, en algún lenguaje, se llama **programa fuente**, y estará generalmente contenido en algún **archivo fuente**.

- El resultado de la traducción será un archivo llamado **objeto** conteniendo las instrucciones de código máquina equivalentes.
    - Sin embargo, este archivo objeto puede no estar completo, ya que el programador puede hacer uso de rutinas o funciones que vienen provistas con el sistema, y no necesita especificar cómo se realizan esas funciones.
    - Al no aparecer en el programa fuente, esas funciones no aparecerán en el archivo objeto.

    - Por ejemplo, cualquier programa "Hola Mundo", en cualquier lenguaje, imprime en pantalla un mensaje; pero la acción de imprimir algo en pantalla no es trivial ni sencilla, y la explicación de cómo se hace esta acción **no está contenida en esos programas**. En su lugar, existe una llamada a una función de impresión cuya definición reside en algún otro lugar.
- Ese otro lugar donde están definidas funciones disponibles para el programador son las **bibliotecas**. Las bibliotecas son archivos conteniendo grupos o familias de funciones.
- El proceso de **vinculación**, que es posterior a la traducción, debe buscar en esas bibliotecas la definición de las funciones faltantes en el archivo objeto.
- Si la vinculación resulta exitosa, el resultado final es un programa **ejecutable**.

Fases del ciclo de compilación

- El desarrollador que necesita producir un archivo ejecutable utilizará varios programas de sistema como editores, traductores, vinculadores, etc.
- En algún momento anterior, alguien habrá creado una biblioteca de funciones para uso futuro. Esa biblioteca consiste en versiones objeto de varias funciones, compiladas, y reunidas con un programa bibliotecario, en un archivo.
- Esa biblioteca es consultada por el vinculador para completar las referencias pendientes del archivo objeto.
- En resumen, la primera fase del ciclo de compilación es necesariamente la **edición del programa fuente**.
- Luego, la traducción para generar un **archivo objeto** con referencias pendientes.
- Luego, la vinculación con **bibliotecas** para resolver esas referencias pendientes.
- El resultado final del ciclo de compilación es un **ejecutable**.

Entornos de desarrollo o IDE

Muchos desarrolladores utilizan algún **ambiente integrado de desarrollo (IDE)**, que es un programa que actúa como intermediario entre el usuario y los componentes del ciclo de compilación (editor, compilador, vinculador, bibliotecas).

El entorno integrado facilita el trabajo al desarrollador automatizando el proceso. Sin embargo, aunque el ambiente integrado lo oculte, el sistema de desarrollo **sigue trabajando como se ha descrito**, con fases separadas y sucesivas para la edición, traducción, vinculación y ejecución de los programas.