Universidad Nacional del Comahue

Facultad de Informática

Introducción a la computación

Apunte de la materia

Versión 2.0 - Revisión IC 2023

Autor: Esp. Eduardo Grosclaude

Índice general

Índice de figuras

Capítulo 1

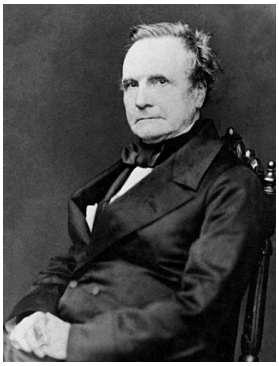
Historia de la Computación

1.1. Antecedentes

Las computadoras actuales tienen una genealogía muy extensa. En el periodo posterior a la Edad Media y anterior a la Edad Moderna, se sentaron las bases para la búsqueda de máquinas de computación más sofisticadas. Unos cuantos inventores comenzaron a experimentar con la tecnología de los engranajes. Entre ellos estaban Blaise Pascal (1623–1662) en Francia, Gottfried Wilhelm Leibniz (1646–1716) en Alemania y Charles Babbage (1792–1871) en Inglaterra. Estas máquinas representaban los datos mediante posicionamiento con engranajes, introduciéndose los datos mecánicamente por el procedimiento de establecer las posiciones iniciales de esos engranajes. En las máquinas de Pascal y Leibniz, la salida se conseguía observando las posiciones finales de los engranajes. Babbage, por su parte, concibió máquinas que imprimían los resultados de los cálculos en papel, con el fin de poder eliminar la posibilidad de que se produjeran errores de transcripción.

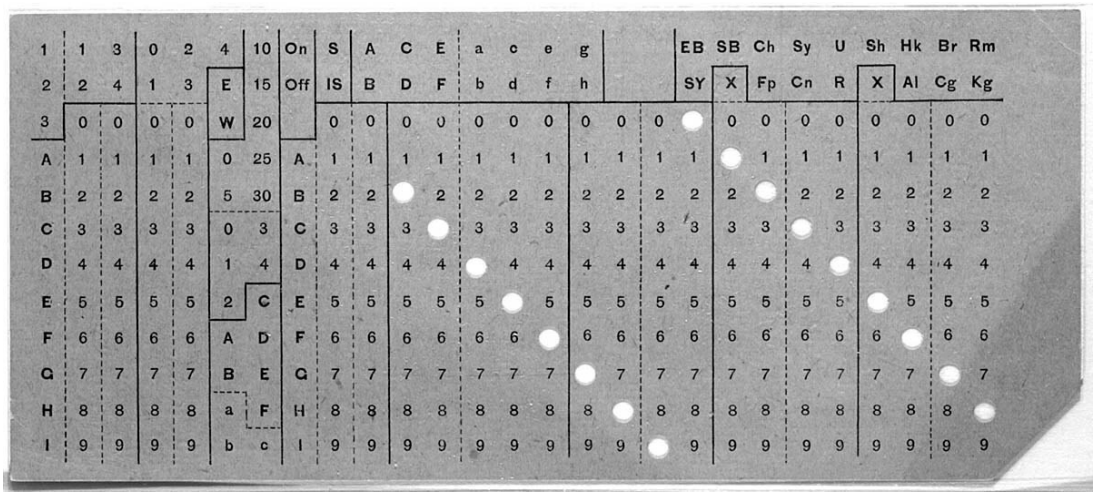
Por lo que respecta a la capacidad de seguir un algoritmo, podemos ver una cierta progresión en la flexibilidad de estas máquinas. La máquina de Pascal se construyó para realizar únicamente sumas. En consecuencia, la secuencia apropiada de pasos estaba integrada dentro de la propia estructura de la máquina. De forma similar, la maquina de Leibniz tenía los algoritmos firmemente integrados en su arquitectura, aunque ofrecía diversas operaciones aritméticas entre las que el operador podría seleccionar una.

La máquina diferencial de Babbage (de la que solo se construyó un modelo de demostración) podía modificarse para realizar diversos cálculos, pero su máquina analítica (para cuya construcción nunca consiguió financiación) estaba diseñada para leer las instrucciones en forma de agujeros realizados en una tarjeta de cartón. Por tanto, la máquina analítica de Babbage era programable. De hecho, se considera a Augusta Ada Byron (Ada Lovelace), que publicó un artículo en el que ilustraba cómo podría programarse la máquina analítica de Babbage para realizar diversos cálculos, como la primera programadora del mundo.



Charles Babbage
(1792–1871)

Herman Hollerith (1860–1929) también aplicó el concepto de representar la información mediante agujeros en tarjetas de cartón para acelerar el proceso de tabulación de resultados en el censo de Estados Unidos de 189 (fue este trabajo de Hollerith el que condujo a la creación de la empresa IBM). Dichas tarjetas terminaron siendo conocidas con el nombre de **tarjetas perforadas** y sobrevivieron como método popular de comunicación con las computadoras hasta bien avanzada la década de 1970.



Tarjeta Perforada

Ada Lovelace
(1815–1852)

La tecnología disponible en aquella época no permitía producir las complejas máquinas basadas en engranajes de Pascal, Leibniz y Babbage de manera económica. Pero los avances experimentados por la electrónica a principios del siglo XX permitieron eliminar dichos obstáculos. Aparecen las computadoras construidas con tubos de vacío.

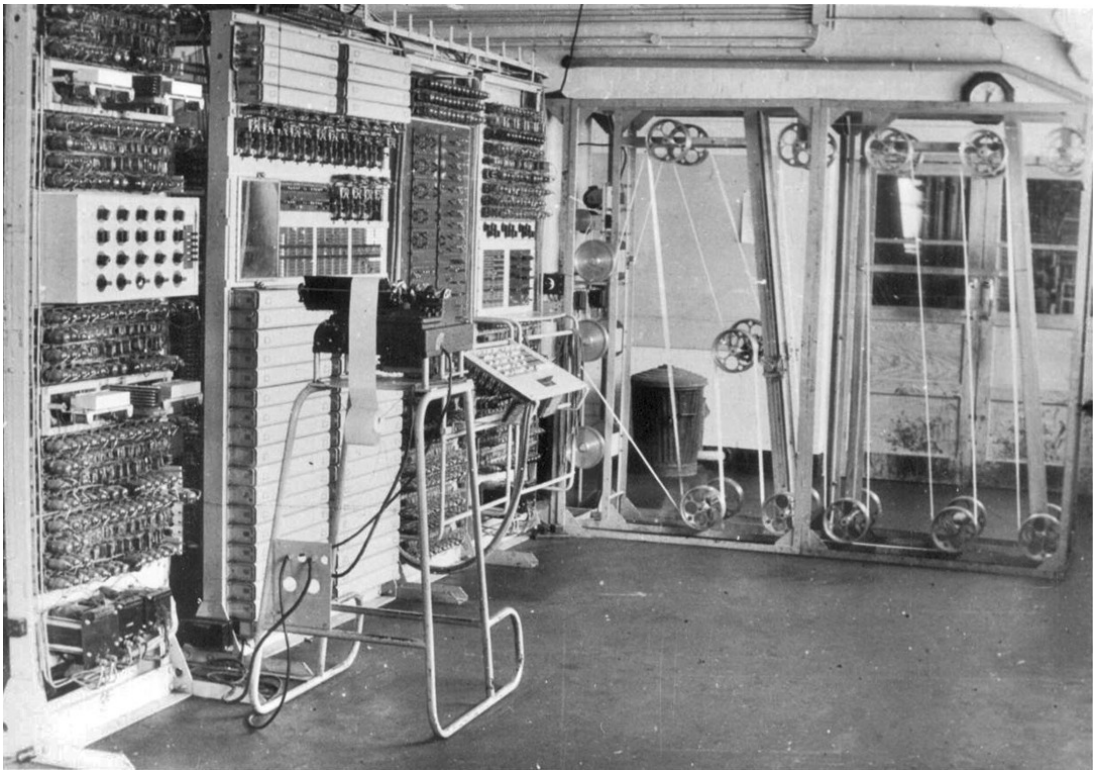
1.2. Primera Generación (1938-1950)

El **tubo de vacío o válvula termoiónica** fue patentado por Edison y fue sucesivamente modificado para diferentes usos en electrónica hasta llegar a ser usado en las computadoras de la primera generación. Una de sus variedades, el **triodo**, tiene tres electrodos o terminales conectados al resto del circuito, llamados **cátodo, ánodo y rejilla o grilla de control**. En éstos, la corriente eléctrica se dirige siempre desde el cátodo al ánodo, pero únicamente circula cuando existe una determinada carga negativa en la grilla, que funciona como un interruptor.

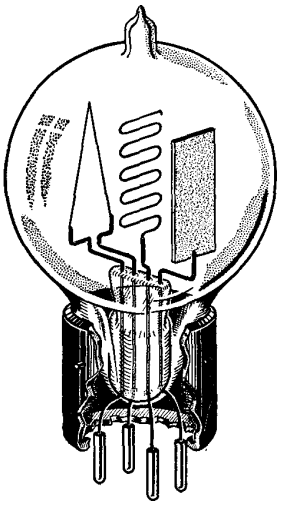
De esta manera se puede controlar el flujo de corriente por un circuito y construir dispositivos que implementen funciones lógicas. Así, dos válvulas de este tipo, conectadas en serie, simulan una función lógica de conjunción o **AND**; dos válvulas conectadas en paralelo, simulan una disyunción u **OR**, etc. Con válvulas termoiónicas es posible además crear un dispositivo que mantenga permanentemente un cierto estado eléctrico, y que por lo tanto **puede almacenar un bit de información**.

Las primeras computadoras electrónicas usaban **tubos de vacíos** como interruptores, implementando dispositivos que realizaban operaciones aritméticas y lógicas. La grilla de las válvulas necesita alcanzar una alta temperatura para poder gobernar el flujo de electrones. De ahí que el consumo de electricidad fuera altísimo y su funcionamiento sumamente lento, unido esto a una alta tasa de fallos.

Dado el momento histórico en el cual aparecieron estos equipos, los objetivos con los cuales se creaban eran, con frecuencia, los usos militares. Las máquinas de esta generación eran grandes instalaciones que ocupaban una habitación, y sus miles de válvulas disipaban una gran cantidad de calor, que debía combatirse con sistemas de aire acondicionado.



Colossus (1943,1944)



Tubo de Vacío

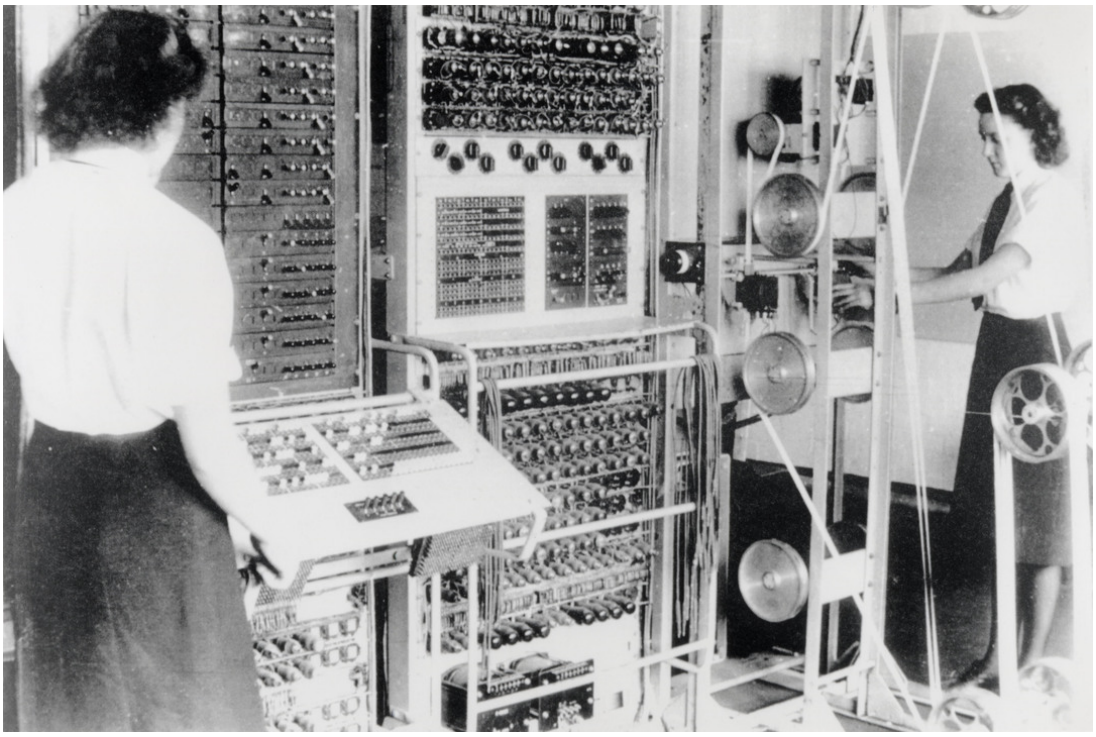
Colossus (1943,1944) fue construida en Inglaterra para decodificar los mensajes alemanes durante los últimos años de la Segunda Guerra Mundial, y se programaba activando y desactivando interruptores físicos. Cada computadora Colossus utilizaba entre 1.600 y 2.400 tubos de vacío. La existencia de la máquina se mantuvo en secreto y el público desconocía su aplicación hasta la década de 1970.

En los Estados Unidos, el trabajo de la computadora ENIAC comenzó a fines de la Segunda Guerra Mundial y la máquina fue terminada en 1945. Esta computadora se programó con paneles de conexión e interruptores.

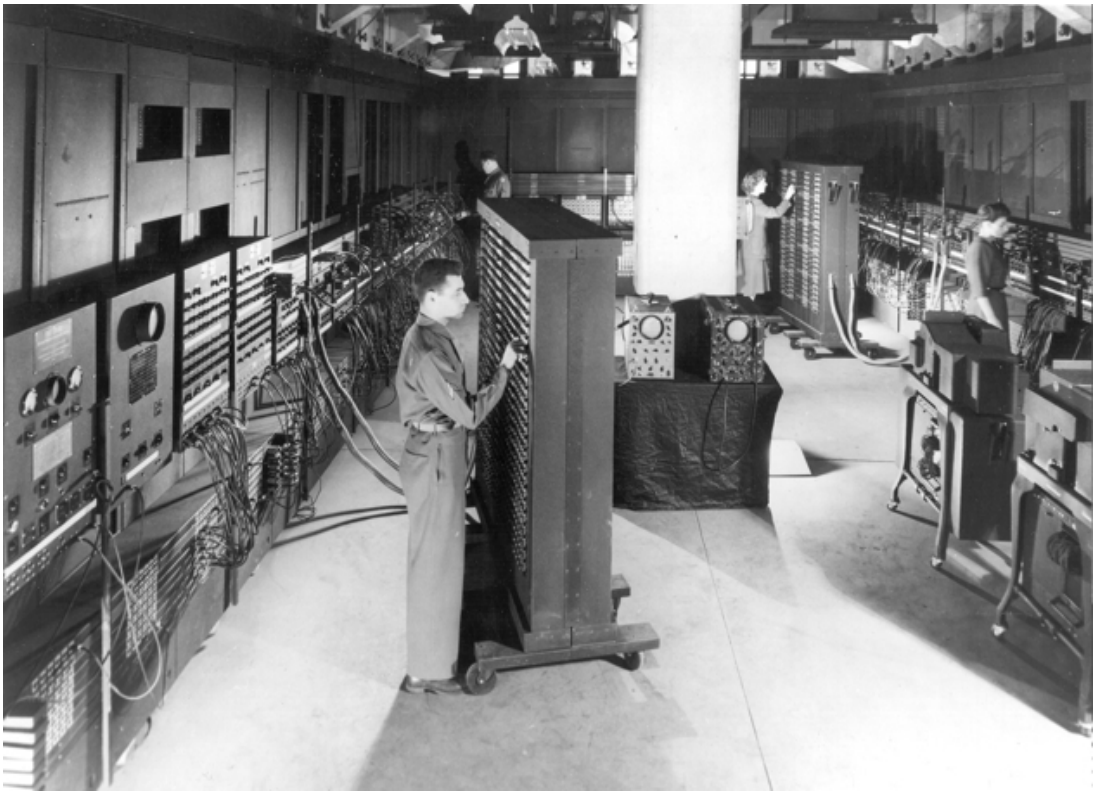
La tecnología de tubo de vacío requería una gran cantidad de electricidad. La computadora ENIAC (1946) tenía más de 17,000 tubos y sufría fallas de tubo (que tardaría aproximadamente 15 minutos en ubicarse) en promedio cada dos días. Debido a que la falla de cualquiera de los miles de tubos en una computadora podría dar lugar a errores, la confiabilidad del tubo era de gran importancia.

Clementina

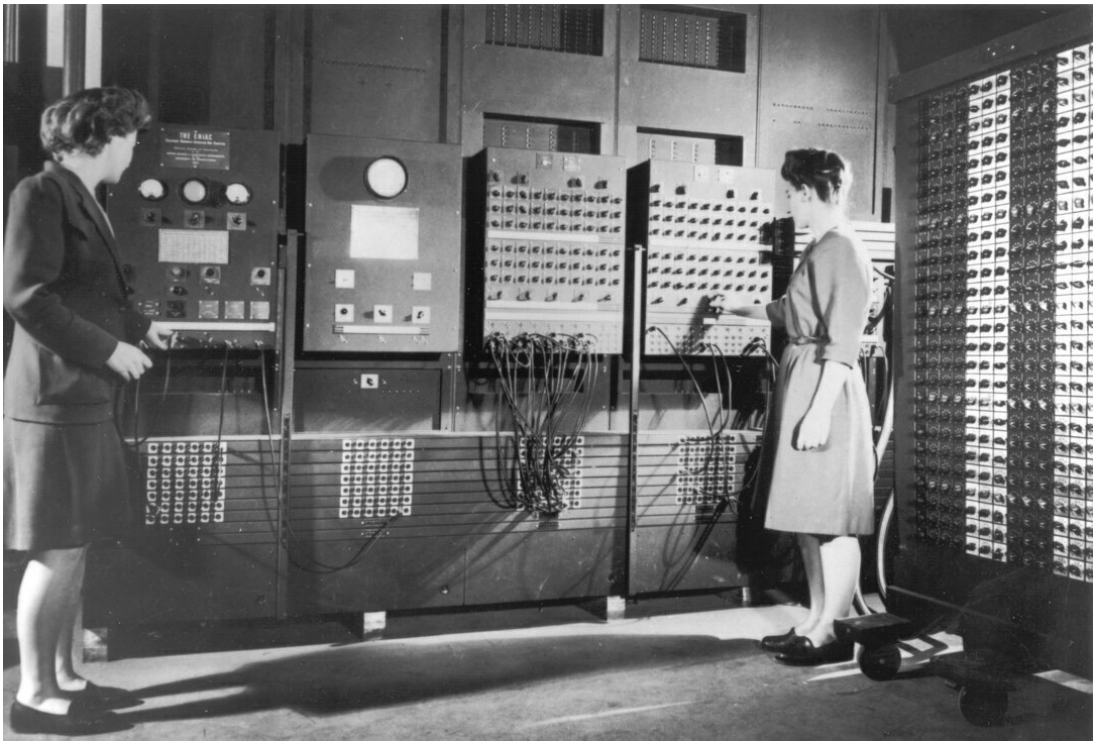
¿Qué pasaba en nuestro país durante estas épocas? La actividad de la computación aquí no había comenzado. Recién a principios de los años 60 la universidad argentina decidió hacer una importante inversión, que fue la compra de una computadora de primera generación, bautizada aquí **Clementina**.



Colossus (1943,1944)



ENIAC (1945)



ENIAC (1945)

1.3. Segunda Generación (1951-1964)

En 1948 se descrubre que combinando elementos que eran vecinos en la tabla periódica, se creaban nuevos materiales con un desbalance de electrones; y que de esta manera se podía controlar el sentido de las corrientes eléctricas que atravesaban esos materiales. Así fue inventado un componente electrónico revolucionario, el **transistor**, que era básicamente un **triodo de estado sólido**, es decir, podía cumplir el mismo papel en un circuito que la válvula termoiónica de tres electrodos, pero era construido de una forma completamente diferente. Esto significa que las mismas funciones lógicas de los interruptores, que en las computadoras de primera generación eran cumplidas por las válvulas termoiónicas, podían ser resueltas con dispositivos mucho más pequeños, de mucho menor consumo, con tiempos de reacción mucho menores y mucho más confiables.

Transistor (1947)

Con estos desarrollos, las máquinas de la década de 1940, que tenían el tamaño de una habitación, se redujeron a lo largo de las décadas siguientes hasta el tamaño de un armario. Algunas de las desarrolladas en esta época recibieron el nombre de **minicomputadoras**. El PDP-1 fue uno de los primeros computadores que pudieron ser accedidos masivamente por los estudiantes de computación. Tenía un **sistema de tiempo compartido (time-sharing)** que hacía posible la utilización de la máquina por varios usuarios a la vez. Tenía 144 KB de memoria principal y ejecutaba 100.000 instrucciones por segundo.

1.4. Tercera Generación (1965-1971)

A mediados de los 60 se desarrollaron los **circuitos integrados o microchips**, que empaquetaban una gran cantidad de transistores

PDP-1 (1959)

en un solo componente, con importantes mejoras en el aspecto funcional y en la economía de la producción de computadoras. Aparecieron computadoras más baratas que llegaron a empresas y establecimientos educativos más pequeños, popularizándose el uso de la computación.

Con estos circuitos era mucho más fácil montar aparatos complicados: receptores de radio o televisión y computadoras. A medida que fue progresando el desarrollo de los **circuitos integrados**, muchos de los circuitos que forman parte de una computadora pasaron a estar disponibles comercialmente encapsulados en unos bloques de plástico diminutos denominados chips. En 1964, IBM anunció el primer grupo de máquinas construidas con **circuitos integrados**. Estas computadoras de *tercera generación* cambiaron totalmente la computación como se conocía hasta el momento, introduciendo una nueva forma de programar que aún se mantiene en las grandes computadoras actuales. También aparecieron las primeras **supercomputadoras**, como el Cray-1, en 1976, que ejecutaba 160 millones de instrucciones por segundo y tenía 8 MiB de memoria principal.

Esto son las principales ventajas de la tercera generación de computadoras:

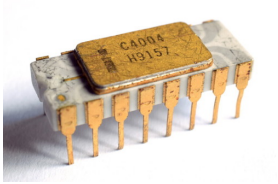
- Menor consumo de energía eléctrica.
- Apreciable reducción del espacio que ocupaba el aparato.
- Aumento de fiabilidad y flexibilidad.

El **microprocesador** desarrollado por Intel reunió la mayor parte de las funciones de las computadoras en un solo microchip. La existencia del microprocesador favoreció la creación de una industria de las computadoras personales. En 1982 IBM propuso el PC (Personal Computer), un **computador personal o microcomputador** del cual descienden la mayoría de las computadoras domésticas y de oficina que se usan hoy. Al contrario que las computadoras de hasta entonces, construidas con procedimientos y componentes propios del fabricante, y a

veces secretos, la **arquitectura abierta** del PC utilizaba componentes existentes y conocidos, y estaba públicamente documentada; de manera que otras empresas podían libremente fabricar componentes compatibles con esta computadora.

1.5. Cuarta Generación (1971-actualidad)

Gracias a **nuevos procesos de fabricación de circuitos integrados**, se logró cada vez mayor miniaturización de componentes, logrando la integración de miles de transistores en un solo componente. El Intel 4004, un CPU de 4 bits, fue el primer microprocesador en un simple chip, así como el primero disponible comercialmente.



Intel 4004 (1971)

Contiene 2300 transistores y una velocidad máxima del reloj 740 kHz, El conjunto de instrucciones está formado por 46 instrucciones (de las cuales 41 son de 8 bits de ancho y 5 de 16 bits de ancho), y contiene 16 registros de 4 bits cada uno.

En 1981, IBM presentó su primera computadora de escritorio. Se le decía *de escritorio o de mesa* en contraste con las computadoras anteriores que ocupaban mucho espacio (como un armario por ejemplo). IBM la denominó computadora personal o PC (Personal Computer), y cuyo software subyacente había sido desarrollado por una empresa de reciente creación de nombre Microsoft.

La PC tuvo un éxito instantáneo y dio legitimidad a la computadora de escritorio como producto de consumo en la mente de la comunidad empresarial. Hoy día, se emplea ampliamente el término PC para hacer referencia a todas esas máquinas de diversos fabricantes cuyo diseño ha evolucionado a partir de la computadora personal inicial de IBM.

IBM PC 5150 (1981)

Capítulo 2

Sistemas de Numeración

En este capítulo veremos la definición de sistema de numeración y la diferencia entre sistema posicional y no posicional. Veremos qué es la base de un sistema de numeración posicional, y cómo hacemos el cambio de una base a otra.

2.1. Introducción

Un **Sistema de Numeración** es un conjunto de símbolos y reglas. Las reglas permiten construir todos los numerales válidos en el sistema, utilizando los símbolos. Es decir, un sistema de numeración es un conjunto de símbolos y de normas a través del cual pueden expresarse elementos válidos dentro de ese sistema, los cuales llamamos numerales. Por lo tanto, todo sistema de numeración contiene un conjunto finito de símbolos, además de un conjunto finito de reglas, mediante las cuales se combinan los símbolos para obtener los numerales del sistema. Cada numeral representa un **número**.

En nuestra vida cotidiana usamos el sistema de numeración **decimal**, el cual nos resulta familiar. Diferentes culturas han desarrollado otros sistemas de numeración y escriben los numerales de otra manera. La figura 2.1 muestra un ejemplo de un sistema de numeración no posicional. Es decir, donde no importa la posición de cada símbolo para conocer el número representado por un determinado numeral.

Figura 2.1: Sistema de numeración egipcio.

Pero, ¿y qué son los numerales? ¿y los números?. Veamos:

- El **numeral** es lo que escribimos.
- El **número** es la cantidad que representa el numeral en cuestión.

Para comprender esto veamos la figura 2.2. Es esa figura vemos cómo distinos numerales pueden representar el mismo número. Ahora bien, si observamos el numeral del hueso atentamente, ¿qué podemos decir de sus tres *rayitas*? ¿Cuánto *pesa* cada rayita? Cada rayita suma una unidad al total, es decir que tienen el mismo peso. A este sistema se le dice que es *no posicional*, en contraposición al **sistema posicional**, que definiremos a continuación.

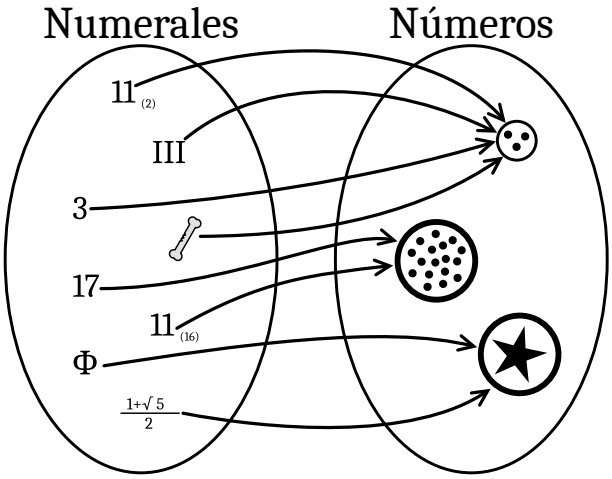


Figura 2.2: Diferencia entre numeral y número

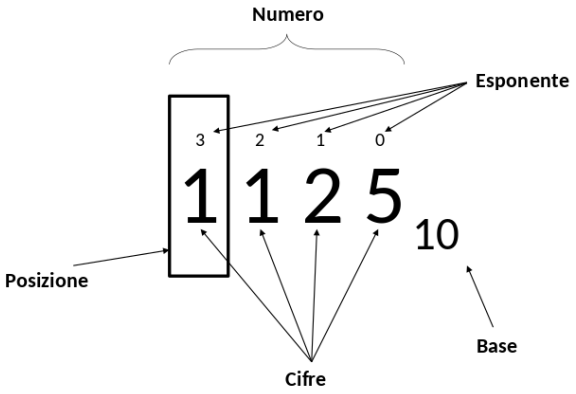


Figura 2.3: Notación del Sistema Posicional (ejemplo con base 10)

2.2. Sistema posicional y base de un sistema de numeración

Se denomina **sistema posicional**, al sistema de numeración en el cual la contribución de un dígito al valor del número, es el producto del valor del dígito por un factor determinado por la posición del dígito. Antes de ver un ejemplo, definamos **base de un sistema de numeración**: La base de un sistema de numeración es la **cantidad de dígitos de que dispone el sistema**. Entonces, el sistema decimal habitual es de base 10, mientras que el sistema binario es de base 2, el sistema octal es de base 8, y así.

Veamos un ejemplo. El sistema decimal es un sistema posicional de base 10. Este sistema cuenta con diez dígitos (de ahí el nombre de *decimal*. Cuando escribimos **15** en el sistema decimal, esta expresión equivale a decir: *para saber de qué cantidad estoy hablando, tome el 5 multiplicado por 1 y luego sume el 1 y multiplicado por 10.*

Como vemos, la forma de calcular el valor de un numeral en base 10, es multiplicando los dígitos del numeral por **potencias de base 10**. ¿Qué potencias? Las potencias arrancan en 0 para la primera posición (la de mas a la derecha) y van creciendo de derecha a izquierda. La figura 2.3 indica una notación posible para saber de qué estamos hablando.

Vemos otro ejemplo. El numeral **2017** (dos cero uno siete) en base 10 es la suma de:

$$2 \times 1000 + 0 \times 100 + 1 \times 10 + 7 \times 1$$

Los dígitos 2, 0, 1 y 7 se multiplican, respectivamente, por 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.

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**.

2.2.1. Expresión General

La forma de calcular el valor de un numeral en una base b genérica, es igual a la forma vista anteriormente, sólo que con potencias **de la base correspondiente**. Las cifras de un **numeral** escrito en cualquier base son los **factores por los cuales hay que multiplicar las sucesivas potencias de la base** para saber a qué **número** nos estamos refiriendo.

Veamos la **expresión general** que nos permite escribir un número n (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.

2.3. Conversión de base

Cuando necesitamos expresar un numeral en otra base se hace lo que denominamos **conversión 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.

2.3.1. Conversión de otras bases a base 10

La conversión de un numeral expresado en una base b cualquiera, a su expresión en base 10, se realiza aplicando la Expresión General vista anteriormente (ver sección 2.2.1).

Es importante 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. La Fig. 2.4 muestra un ejemplo.

Ejemplos:

- $60_{(7)} = 6\times7^1 + 0\times7^0 = 42_{(10)}$
- $2A_{(16)} = 2\times(16)^1 + (10)\times(16)^0 = 42_{(10)}$
- $1120_{(3)} = 1\times3^3 + 1\times3^2 + 2\times3^1 + 0\times3^0 = 42_{(10)}$

Figura 2.4: Ejemplo de conversión de cualquier base a base 10

2.3.2. 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). Procedimiento:

- 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.

La Fig. 2.5 muestra algunos ejemplos.

Ejemplo:

	C	R
$1961 \div 16$	122	9
$122 \div 16$	7	10
$7 \div 16$	0	7

Entonces: $1961_{(10)} = 7A9_{(16)}$

Figura 2.5: Ejemplo de conversión de base 10 a otra base.

Algunas consideraciones a tener en cuenta:

- 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, **son dígitos válidos de un sistema en la base destino** al ser todos menores que ella.

Caso particular: Conversión de base 2 a decimal

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**. Como indica

la Expresión General, los numerales se escriben como suma de dígitos del sistema multiplicados por potencias de la base. Por ejemplo,

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

y

$$1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 = 1111_{(2)} = 15_{(10)}$$

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 desde y hacia el sistema binario. Una manera de facilitar esto es memorizar los valores de algunas potencias iniciales de la base 2:

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

¿Qué utilidad tiene memorizar esta tabla? Que nos permite convertir mentalmente algunos casos simples de números en sistema decimal, a base 2. Por ejemplo, veamos los pasos para convertir el número **12** a binario:

- Escribimos el número como suma de potencias de 2. En este caso 12 equivale a **8 + 4**.
- Convertimos a binario cada número:
  $8 = 2^3 = 2^3 + 0^2 + 0^1 + 0^0 ==> 1000_{(2)}$
  $4 = 2^2 = 2^2 + 0^1 + 0^0 ==> 100_{(2)}$
- Sumamos los números ya convertidos en binario. En este caso $1000_{(2)} + 100_{(2)} = 1100_{(2)}$

Luego, la expresión de 12 decimal convertida a binario será **1100**.

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: $1010000_{(2)} = 80_{(10)}$.

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?

2.3.3. 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 es la base 10. La buena noticia es

0 0 1	1 1 0	0 1 1	1 0 0	0 1 0	1 0 1	1 1 0	0 0 1
1	6	3	4	2	5	6	1

Conversión de binario a octal agrupando de a 3 dígitos binarios

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** y luego **de base 10 a** b_2, aplicando los procedimientos ya vistos. Eso es todo.

2.3.4. Conversión entre sistemas binario y octal o hexadecimal

Pero en algunos casos especiales podemos aprovechar cierta relación existente entre las bases a convertir: por ejemplo, cuando son **2 y 16** (binario y hexadecimal), o **2 y 8** (binario y octal). 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. 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.

- 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

Para convertir entre bases 2 y 8 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. La figura 2.6 muestra un ejemplo.

- 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

Para convertir de base 2 a base 16 simplemente hay que 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. Para convertirde base 16 a base 2, reemplazamos cada dígito hexadecimal por los cuatro dígitos binarios que lo representan.

Capítulo 3

Unidades de Información

En este capítulo veremos qué es la información y cómo podemos cuantificarla, es decir, como podemos medir la cantidad de información que, por ejemplo, puede guardar un dispositivo. Veremos además las relaciones entre las diferentes unidades de información.

3.1. ¿Qué es la 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 números, textos, 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.

3.2. El 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**. La memoria de las computadoras está diseñada de forma que le premite almacenar **dos estados** en cada celda. 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. Todo lo que puede guardar, procesar, o emitir una computadora digital, está 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.

3.2.1. El viaje de un bit

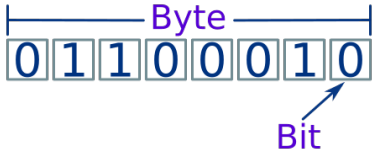
En una famosa película de aventuras hay una ciudad en problemas. Uno de las héroes enciende una pila de leña porque se prepara un terrible ataque sobre la ciudad. La pila de leña es el dispositivo preestablecido que tiene la ciudad para pedir ayuda en caso de emergencia.

En la cima de la montaña que está cruzando el valle existe un puesto similar, con su propio montón de leña, y un vigía. Quien 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. 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**. 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.

Notemos que en los manuales de lógica o de informática, encontraremos siempre asociados los **bits** con los valores **0 y 1**. Aunque lo que viajó desde la ciudad sitiada hasta su destino no es un 0 ni un 1, es **un bit de información**. Sin embargo, la identificación de los bits con los dígitos binarios es útil para todo lo que tiene que ver con las computadoras.

3.3. El 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** (ver figura 3.3). 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.



Un byte son 8 bits

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.

3.4. Sistemas de medición

Como tanto el tamaño de la memoria de una computadora, como el tamaño de un archivo (cantidad de bytes que lo componen), suelen ser grandes (es decir, de muchos bytes), utilizamos las unidades mas apropiadas a cada caso. De la misma manera que para indicar la distancia que hay entre la ciudad de Cipolletti y Las Grutas usamos kilómetros y no metros, para indicar el tamaño de un archivo podemos usar kilobytes en vez de bytes.

Para esto, existen dos sistemas diferentes: el **Sistema Internacional** y el **Sistema de Prefijos Binarios**. Las unidades de ambos sistemas son parecidas, pero no 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.

3.4.1. Sistema Internacional

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

- El **kilobyte (KB)**: 1000 bytes
- El **megabyte (MB)**: 1000 × 1000 bytes = 1000 kilobytes = un millón de bytes
- El **gigabyte (GB)**: 1000 × 1000 × 1000 bytes = mil megabytes = mil millones de bytes
- El **terabyte (TB)**: 1000 × 1000 × 1000 × 1000 bytes = mil gigabytes = un billón de bytes
- Y así siguen otros múltiplos mayores como **petabyte, exabyte, zettabyte, yottabyte**.

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

3.4.2. 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 (KiB)**: 1024 bytes
- El **mebibyte (MiB)**: 1024 × 1024 bytes = 1048576 bytes
- El **gibibyte (GiB)**: 1024 × 1024 × 1024 bytes
- El **tebibyte (TiB)**: 1024 × 1024 × 1024 ×
- Y así siguen otros múltiplos mayores como **pebibyte, exbibyte, zebibyte, yobibyte**.

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

3.4.3. ¿Por qué dos sistemas?

¿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 quienes diseñan y programa las computadoras 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.

- 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*.

Capítulo 4

Representación de datos numéricos

Veremos en este capítulo cómo pueden representarse datos numéricos (enteros y fraccionarios) mediante patrones de bits. Idealmente, un sistema de numeración puede usar infinitos dígitos para representar números arbitrariamente grandes. Si bien esto es matemáticamente correcto, las computadoras tienen limitaciones físicas, y con ellas no es posible representar números de infinita cantidad de dígitos. Es por esto en siempre que trabajemos con un sistema de representación de datos debemos conocer la cantidad de bits de que disponemos.

Es importante recordar que solo se puede operar entre datos representados **con el mismo** sistema de representación, y que el resultado de esta operación estará representado en ese sistema. Por ejemplo, si tenemos dos valores A y B, y queremos sumarlos, ambos valores deben estar representados en el mismo sistema.

4.1. Rango de Representación

Cada **sistema de representación de datos numéricos** tiene su propio **rango de representación** (que podemos abreviar RR).

EL RR de un sistema de representación de datos numéricos es el intervalo de números representables por dicho sistema.

Este intervalo puede ser escrito como $[a, b]$, donde a y b son sus límites inferior y superior, respectivamente. Estos límites definen el intervalo de la recta numérica puede ser representada. Ningún número fuera del RR de un sistema de representación de datos numéricos puede ser representado en dicho sistema. Conocer este intervalo es importante para saber las limitaciones que tendremos al programar.

¿De qué depende que un RR sea mayor que otro? En general, mientras más bits utilice el sistema, mayor será el RR. Sin embargo, el RR también depende de la forma en que el sistema **utilice** esos bits. Un sistema puede ser más o menos **eficiente** que otro en el uso de esos dígitos. Por lo tanto, decimos que el rango de representación depende tanto de la **cantidad de bits** como de la **forma de funcionamiento** del sistema de representación.

4.1.1. Valores representables con k bits: Cuántos y cuáles

Cuántos

Veamos cómo calcular cuántos valores diferentes podemos representar con una cantidad fija de bits. Veamos un ejemplo con pocos bits. Si tenemos 3 bits, todos los números representables son los siguientes:

Decimal	Binario
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Si observamos esta tabla, vemos que, usando 3 bits, podemos representar ocho números diferentes (del 0 al 7), y ocho es justamente 2^3, donde 2 es la base del sistema (estamos trabajando con sistema binario) y 3 es la cantidad de bits. Recordemos que como estamos estudianto sistemas de representación de datos en la computadora, debemos atenernos a una cantidad fija de bits. A esa **cantidad fija de bits** la nombraremos k.

¿Cuántos valores diferentes podemos representar con k bits? Con k bits, podemos representar 2^k valores diferentes.

Cuáles

¿Cómo interpretar esos 2^k valores diferentes? Dependerá del sistema de representación que estemos utilizando. En nuestro ejemplo de 3 bits, el sistema de representación que usemos definirá cómo interpretar esos 8 valores diferentes. Por ejemplo, en un sistema de representación ."A", el "111"puede significar $7_{(10)}$, mientras que en otro sistema de representación "B", el "111"puede representar $-1_{(10)}$, y así. Entonces, cuando nos preguntemos ¿qué valor representa el "111¿ La respuesta es: Depende! ¿De qué depende? Del sistema de representación con el que estemos trabajando.

¿Cuáles valores podemos representar con k bits? Depende del sistema de representación que usemos.

Preguntas de repaso

- ¿Cuáles son los límites del rango de representación de un sistema de representación numérica?
- ¿Cuántos valores diferentes podemos representar con k bits?

4.2. Sistema de 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 de Representación Sin Signo**, simplemente usamos el sistema binario de numeración, tal como lo conocemos. Podemos entonces abreviar el nombre de este sistema como **SS(k)**, donde k es la cantidad fija de bits.

4.2.1. Rango de representación de SS(k)

¿Cuál será el rango de representación de este sistema? 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? La respuesta es: $2^k - 1$

Por lo tanto, **el rango de representación de un sistema sin signo con k dígitos es** $[0, 2^k - 1]$. Todos los números representables en esta clase de sistemas son **positivos o cero**.

Ejemplos

- k = 8 bits: $[0, 2^8 - 1] = [0, 255]$
- k = 16 bits: $[0, 2^{16} - 1] = [0, 65{,}535]$
- k = 32 bits: $[0, 2^{32} - 1] = [0, 4{,}294{,}967{,}295]$

¿Cuándo usaremos este sistema de representación? Cuando lo que necesitemos representar siempre utilice números positivos o cero. Por ejemplo, la edad de una persona siempre será un número entero igual o mayor a cero. Entonces en ese caso podríamos utilizar este sistema de representación.

4.3. Sistema de Representación con Signo

En la vida diaria manejamos continuamente números negativos, y los distinguimos de los positivos simplemente agregando un signo − adelante. 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, pero para poder representar tanto números **positivos como negativos**, necesitamos cambiar la interpretación de una misma secuencia de bits. 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 (para entender esto leer subsección 4.1.1).

Preguntas para pensar

- 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?

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

4.4. Sistema de Representación Signo-magnitud: SM(k)

El sistema **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. En este sistema, el bit reservador para expresar el signo no se puede usar para representar magnitud.

Ejemplos con k=8

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

4.4.1. Rango de Representación de SM(k)

- En todo número escrito en el sistema de signo-magnitud con 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**.
- Este valor absoluto se representa con el sistema **sin signo** sobre $k-1$ bits, es decir, SS(k-1).
- Sabemos que el rango de representación de SS(k) es $[0, 2^k - 1]$. Por lo tanto, el rango de representación de SS(k-1), reemplazando, será $[0, 2^{k-1} - 1]$.
- Esto quiere decir que el mayor número representable en SM(k) es $2^{k-1} - 1$.
- Como en este sistema también se puede representar el opuesto negativo de cualquier número, en particular el opuesto negativo del máximo número representable será el menor número representable. Este es $-(2^{k-1} - 1)$.
- Con lo cual hemos calculado tanto el límite inferior como el superior del rango de representación de SM(k), que, finalmente, es $[-(2^{k-1} - 1), 2^{k-1} - 1]$.

4.4.2. Limitaciones de Signo-Magnitud

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

- Existen dos representaciones del 0 (una "positivaz otra "negativa), lo cual desperdicia una secuencia de bits que podría usarse para representar otro número y ampliar el RR.
- 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**.

4.5. Sistema de Representación y Operación Complemento a 2

Para comprender el **Sistema de Representación Complemento a 2** es necesario primero conocer la **Pperación de Complemento a 2**.

4.5.1. Operación de Complemento a 2

La **operación** de complementar a 2 consiste en obtener el **opuesto** de un número, es decir, el que tiene el mismo valor absoluto pero signo opuesto.

Procedimiento para obtener el complemento a 2 de un número:

- Se invierte cada uno de los bits (es decir, cada cero se reemplaza por uno, y cada uno se reemplaza por cero.
- Se suma 1 al resultado de la inversión de bits.

Propiedad fundamental

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

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

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: $0011 + 1101 = 0$.

4.5.2. Conversión de base 10 a Complemento a 2

Procedimiento para representar un número a que está en base decimal, en el Sistema de Representación Complemento a 2:

- Si a es positivo o cero, lo representamos con en el sistema Sin Signo.
- Si a es negativo, tomamos su valor absoluto y lo representamos con en el sistema Sin Signo. Luego, a la cadena de bits obtenida le aplicamos la operación de complemento a 2.

Ejemplos

- Representemos el número 17 en complemento a 2 con 8 bits. Como es positivo, lo escribimos en base 2 con 7 bits, obteniendo 0010001.
- 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 esto le aplicamos la operación de C2. El resultado final es 11101111.

4.5.3. Conversión de Complemento a 2 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 significativo 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 luego se le agrega el signo $-$ para indicar 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 $-17_{(10}$.

4.5.4. Rango de Representación de C2(k)

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

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 ya que representa un valor más.

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 directamente ya que no se requiere hacer 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.

4.5.5. Complementar a 2 vs. Representar en C2

Un error frecuente es confundir la **operación de complementar a 2** y la **representación en complemento a 2**. ¡No son lo mismo! Al representar un número en complemento a 2, la operación de complementar a 2 **únicamente se aplica cuando queremos obtener el opuesto** de un número, y esto es en el caso que el número que queramos representar sea negativo.

4.5.6. 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. 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 para pensar

- 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?
- ¿Cuál es el **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?

4.5.7. Overflow o desbordamiento en C2

En todo sistema de representación de datos numéricos, la suma de **dos números positivos, o de dos números negativos** puede dar un resultado que sea imposible de representar porque puede caer fuera 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. Un ejemplo de una suma de números binarios en C2, que tiene desbordamiento (u overflow en inglés) se muestra en la figura 4.1. En este ejemplo, se suma $1 + 7$ que da 8, sin embargo si observamos el resultado es -8 en C2 con 4 bits.

Figura 4.1: Suma aritmética de dos operandos (representados en C2)

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 para informar al proceso que se produjo esa situación (el desbordamiento).

Overflow en la suma aritmética en C2

Para el caso particular de la suma aritmética de operandos en C2, una forma sencilla de detectar *overflow* es comparando los dos últimos *acarreos* de la operación. ¿Qué es el acarreo (o *carry* en inglés)? En aritmética, el acarreo es el nombre utilizado para describir un recurso mnemotécnico en una operación aritmética, principalmente en la operación suma[1]. En la Figura 4.2 vemos un ejemplo de una suma, donde el dígito 1 es el acarreo.

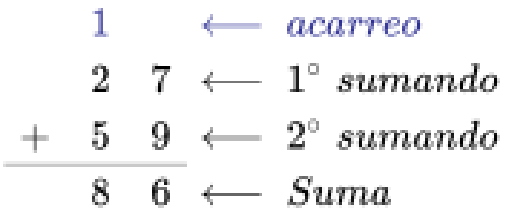


Figura 4.2: Acarreos o carris de una suma aritmética.

Ahora bien, volviendo al método para la detección del overflow en una operación de suma en C2, lo que debemos hacer es comparar los dos últimos acarreos, es decir el último *carry-in* con el último *carry-out*. El acarreo que ingresa a una columna se denomina *carry-in*, mientras que al acarreo que sale de una suma (para ir a la columna siguiente) se denomina *carry-out*. El procedimiento es el siguiente:

- Si, luego de efectuar una suma en C2, los valores de los bits del último *carry-in* y el último *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 del último *carry-in* y del último *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

La figura 4.3 muestra un ejemplo. La primera fila muestra los sucesivos acarreos, las siguientes dos filas los operandos a sumar, y se marcan con amarillo los dos últimos carris a comparar. Si seguimos el procedimiento indicado antes con este ejemplo, vemos que los dos últimos bits de acarreo son iguales (pintados de amarillo) lo que nos indica que no hubo overlfow.

Suma con overflow

[1] https://es.wikipedia.org/wiki/Acarreo

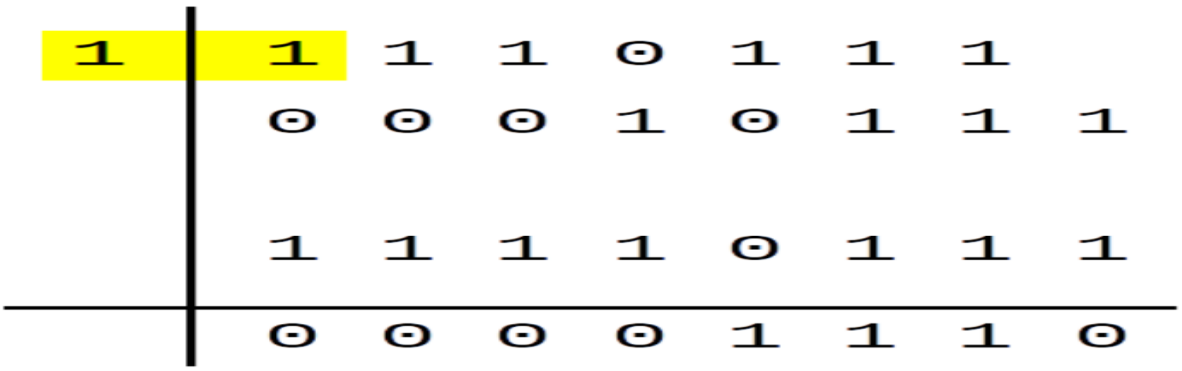


Figura 4.3: Comparación de los últimos acarreos para detectar overflow.

Veamos otro ejemplo, la operación $123 + 9$ en C2 a 8 bits. El resultado (que es 132) cae fuera del rango de representación. Si hacemos la suma de esos valores en binario y revisamos los dos últimos bits de acarreo deberían ser diferentes. Puede verificarlo? Primero hay que convertir esos valores en decimal a binario usando el sistema de representación de datos C2, con 8 bits. Luego hacer la suma, anotando todos los acarreos. Finalmente comparar los dos últimos acarreos. Sin son diferentes, es poque hubo overflow.

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?

4.5.8. Extensión de signo en C2

En una suma en C2, si uno de los operandos estuviera expresado con menos bits que el otro, será necesario **extenderlo** hasta el ancho del operando de mayor cantidad de bits, y operar con ambos operandos con el mismo ancho. Si el operando a extender es positivo, la extensión se realiza simplemente **completando con ceros a la izquierda** hasta obtener la cantidad de dígitos necesaria. Si el operando a extender 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 $C_2^5 \rightarrow$ llevar ambos a C_2^8
  - Se completa B (positivo) como $00000101_{(2}$
- $A + B = 1010_{(2} + 0110100_{(2}$
  - A está en C_2^4 y B en $C_2^7 \rightarrow$ llevar ambos a C_2^7
  - Se completa A (negativo) como $1111010_{(2}$

4.6. Notación en exceso

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 mapean a los enteros en $[a, b]$, uno por uno. Es decir, si usamos 3 bits, el mapeo sería el siguiente:

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

Notemos que tanto a como b pueden ser positivos o 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 (que veremos mas adelante).

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

- Para calcular la secuencia que corresponde al número 20, hacemos $20 - 10 = 10$, lo convertimos a SS(k=4) y el resultado será el binario **1010**.
- Para calcular el valor decimal que está representando la secuencia **1011**, convertimos 1011 a decimal, que es 11, y le sumamos 10 (que es el límite inferior del intervalo [a,b]); el resultado es 21.

Representemos en el sistema de representación 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 al resultado lo convertimos a SS(K=3) que será la el binario **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?

4.7. Números fraccionarios y decimales

Los **números fraccionarios** son aquellos **racionales** que no son enteros, y se escriben como una razón, fracción o cociente de dos enteros. Por ejemplo, $3/4$ y $-12/5$ son números fraccionarios. El signo de división que usamos para escribir las fracciones tiene precisamente ese significado aritmético: si hacemos la operación de división correspondiente entre numerador y divisor de la fracción, obtenemos **la forma decimal del mismo número**, con **una parte entera y una parte decimal**. Así, por ejemplo, $3/4$ también puede escribirse como 0,75, y $-12/5$ como $-2{,}4$. Estas dos formas son equivalentes. En los números decimales, la parte decimal puede ser **finita** o **periódica** (no finita).

Por otro lado, existen números reales que no son racionales, en el sentido de que no existe una razón, fracción o cociente que les sea igual, pero pueden escribirse como decimales (con una parte entera y una parte decimal). Estos son los **irracionales**. Los irracionales pueden expresarse como el resultado de alguna operación (como cuando escribimos $\sqrt{2}$) o en su forma decimal (con una parte entera y una parte decimal). Estos números tienen la característica de que su desarrollo decimal **es infinito** no periódico, por lo cual cuando escribimos un irracional en la forma de decimal (con parte entera y parte decimal), necesitamos **truncar** la parte decimal, ya que no podremos escribir la sucesión completa de decimales (pues es infinita). De manera que, al escribir irracionales en su forma decimal, lo que escribimos son aproximaciones racionales a esos irracionales. Por ejemplo, 3,14, 3,1416 y 3,14159 son aproximaciones racionales al verdadero valor irracional de π, cuya parte decimal tiene infinitos dígitos.

4.7.1. Conversión de binario a decimal

Usando la Expresión General Extendida Al escribir un número con cifras decimales usamos una marca especial para separar la parte entera de la decimal: la **coma o punto decimal**. Esta coma o punto señala el lugar donde los exponentes de la base de la expresión general **se hacen negativos**.

Ejemplo Para convertir el número $11{,}101_{(2}$ a base 10, la Expresión General Extendida nos dice que:

$$11{,}101_{(2} =$$

$$1 \times 2^1 + 1 \times 2^0 +$$

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

$$2 + 1 + 0{,}5 + 0 + 0{,}125 =$$

$$3{,}625$$

Usando el método de multiplicar, convertir y dividir Otra manera de obtener el valor decimal de un número con decimales n en base 2 consiste en utilizar el hecho de que cada vez que desplazamos el punto fraccionario un lugar hacia la derecha, estamos multiplicando n por 2, y viceversa, si desplazamos el punto hacia la izquierda, lo dividimos por 2. El método consiste en:

- Identificar cuántas posiciones fraccionarias tiene n (llamémosla f).
- Multiplicar n por 2^f obteniendo un **entero** en base 2.
- Convertir el entero resultante a base 10 (como hacíamos antes).
- Dividir el resultado por 2^f, obteniendo n en base 10.

Ejemplo El número $n = 11{,}101_{(2)}$ tiene tres cifras decimales ($f = 3$). Lo convertimos en entero dejando $11101_{(2)}$; averiguamos que este número en base 10 es 29; y finalmente dividimos 29 por 2^3. Concluimos que $n = 11{,}101_{(2)} = 29/8 = 3{,}625$.

4.7.2. Conversión de decimal a binario

Para convertir a binario un número con decimales que está en base 10, se convierte la parte entera y la parte decimal de manera separada. Para calcular la parte fraccionaria binaria de n seguimos un procedimiento **iterativo** (es decir, que consta de pasos que se repiten). El método consiste en:

1. Se separan la parte entera y la parte fraccionaria.
2. Se convierte la parte entera a binario.
3. La parte fraccionaria se multiplica por 2 y se toma la parte entera del resultado. Este dígito binario se agrega al resultado.
4. Se repite el paso anterior con la nueva parte fraccionaria obtenida, hasta que ésta sea 0, o hasta lograr la precisión deseada.

El procedimiento de separar, guardar, multiplicar, se repite hasta que la **parte fraccionaria** obtenida en una multiplicación sea 0 (ya no tiene sentido seguir el procedimiento porque el resultado será siempre 0) o hasta que tengamos suficientes dígitos decimales.

La sucesión de dígitos aparecidos como partes enteras durante este procedimiento servirán para **construir la parte fraccionaria** del resultado. Notemos que estos dígitos que aparecen solamente pueden ser ceros y unos, porque son la parte entera de $2 \times x$ con $x < 1$.

Ejemplo

Convirtamos el número $n = 3{,}625$ a base 2.

1. Separamos parte entera (3) y parte fraccionaria (0.625).
2. La parte entera se convierte a base 2 como entero sin signo dando $11_{(2)}$.
3. La parte fraccionaria se multiplica por 2 y separamos este resultado a su vez en parte entera y parte fraccionaria. Guardamos la parte entera del resultado. $0{,}625 \times 2 = 1{,}25$ (guardamos el 1)

4. Volvemos a multiplicar por 2 la parte fraccionaria recién obtenida, y guardamos la parte entera, $0{,}25 \times 2 = 0{,}5$ (guardamos el 0)

5. Tomamos la parte fraccionaria y volvemos a multiplicar por 2, $0{,}5 \times 2 = 1{,}0$ (guardamos el 1). Aquí el procedimiento finaliza porque la parte fraccionaria encontrada es 0.

6. Las partes enteras parciales (que se fueron guardando) fueron **1, 0 y 1**, dando la parte fraccionaria final $,101_{(2}$. A esta parte fraccionaria se le suma la parte entera: $11_{(2} + 0{,}101_{(2} = 11{,}101_{(2}$.

4.8. Representación de números fraccionarios

En esta sección veremos dos formas de representar en la computadora números fraccionarios: Punto Fijo y Punto Flotante. En nuestra vida cotidiana, cuando escribimos números que tienen una parte decimal, usamos un punto (o coma) para indicar el lugar donde comienzan los decimales. Pero, ¿cómo podemos representar en la computadora un número que contine un punto (o coma) en el medio? El problema de cómo guardar el punto o coma decimal es parecido al problema de cómo guardar el signo *menos* de los números negativos: en la memoria solo podemos guardar bits, de forma que habrá que establecer alguna convención para indicar dónde está el punto o coma fraccionaria.

4.8.1. Representación de punto fijo

Los sistemas de punto fijo establecen una cantidad fija de bits o **ancho total** (que llamaremos n) y una cantidad fija de bits para la parte fraccionaria (que llamaremos k). Por ejemplo, la notación $PF(8, 3)$ denota un sistema de punto fijo con 8 bits en total, de los cuales 3 son para la parte fraccionaria. Al ser fijos los anchos de parte entera y fraccionaria, la computadora **puede tratar aritméticamente a todos los números como si fueran enteros**, sin preocuparse por partes enteras ni fraccionarias. Solamente habrá que utilizar la convención al momento de imprimir o comunicar un resultado. La impresora, o la pantalla, deberán mostrar un resultado con coma fraccionaria en el lugar correcto.

Ejemplo

- Supongamos que queremos computar $3{,}625 + 1{,}25$ en un sistema $PF(8, 3)$.
- Las conversiones de estos sumandos a fraccionarios binarios son, respectivamente, 11,101 y 1,01.
- En la memoria se almacenarán como **00011101** y **00001010**. Nótese que al ser todas las partes fraccionarias del mismo ancho, quedan automáticamente encolumnados los invisibles puntos fraccionarios.
- La suma se efectuará bit a bit como si se tratara de enteros y será **00100111**.
- Si pedimos a la computadora que imprima este valor, aplicará la convención $PF(8, 3)$ e imprimirá **00100.111**, o su interpretación en decimal, 4,875, que es efectivamente $3{,}625 + 1{,}25$.

Conversión de decimal a Punto Fijo (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

Cuando vimos enteros, vimos que con cierta cantidad de bits podíamos representar valores dentro de un Rango de Representación. Valores fuera de ese rango no eran representables (porque los bits no alcanzaban). En los números fraccionarios veremos que la cantidad de bits designados para la parte fraccionaria puede no ser suficiente para representar la totalidad de los decimales, pero podemos representar una aproximación al número, **truncando** algunos decimales. El número almacenado en el sistema PF(n,k) será una aproximación al número original, y no estará representándolo con todos sus dígitos fraccionarios. La consecuencia de este truncamiento es la aparición de un **error de truncamiento** o pérdida de precisión. Si quisiéramos conocer de cuánto es ese error, deberíamos calcular la diferencia entre el número que queremos representar, y el número finalmente representado por la computadora. Veamos un ejemplo.

- Necesitamos representar el número 3.1459, y usaremos notación Punto Fijo(8,3).
- Obtenemos parte entera: 00011.
- Obtenemos parte fraccionaria: 001.
- Representación obtenida: 00011001.
- Reconvertimos el binario obtenido (00011001) a decimal: Obtenemos parte entera 3 y parte fraccionaria 0.125.
- Por lo tanto, el número representado en PF(8,3) como 00011001 es en realidad **3.125** y no 3.1459.
- Calculamos el error por truncamiento: $3{,}1459 - 3{,}1250 = 0{,}0209$.
- Observar que el error es menor que $2^{-3} = 0{,}125$ (donde 3 es la cantidad de dígitos usados para la parte fraccionaria).

Conversión de Punto Fijo (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 de repaso

- ¿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 $PF(8,k)$?

Ventajas y desventajas de Punto Fijo

La principal ventaja 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 sistemas 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.

La principal desventaja puede ser que no es una representación adecuada para cierta clase de problemas donde los datos que se manejan son de magnitudes y precisiones muy diferentes entre sí. 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 necesaria para representar la velocidad de la luz.

- 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.

4.8.2. Notación Científica

En Matemática, la respuesta al problema del cálculo con datos con valores muy 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, lo que permite operar entre ellos con facilidad. Al operar con números en esta notación podemos aprovechar las reglas del álgebra para calcular m y e separadamente, y evitar cuentas con muchos dígitos.

Ejemplo

La velocidad de la luz expresada en metros por segundo, 300,000,000 m/s aproximadamente, expresada en Notación Científica es: 3×10^8 m/s. La longitud en metros de un nanómetro, se representará en notación científica como 1×10^{-9}.

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 $3{,}33 \times 10^{-18}$.

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** (único valor que cumple la condición de normalización) y ajustando el exponente de base 2, de manera de no modificar el resultado.

Ejemplos

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

4.8.3. Representación en Punto Flotante

La herramienta matemática de la Notación Científica (vista en la sección anterior) 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. 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). 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:
  - 1 bit de signo.
  - 8 bits para el exponente.
  - 23 bits para la mantisa.
- IEEE 754 precisión doble, se define sobre un campo de 64 bits:
  - 1 bit de signo.
  - 11 bits para el exponente.
  - 52 bits para la mantisa.

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). 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

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 11000000101100000000000000000000.

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 escribir.

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* los pasos hechos anteriormente para convertir un decimal a punto flotante.

- 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 por 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. Un número que cae fuera del rango de representación porque es muy grande, es fácil de ver. Ahora bien, ¿qué sucede con los números muy pequeños?

Existe, por otro lado, un problema al tratar con números 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 programación reconoce una cadena de caracteres como *0.1*, introducida por la persona que programa o usa el programa, 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 los resultados que van arrastrando el error.

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.

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).

Capítulo 5

Representación de Texto e Imágenes

En el capítulo anterior vimos cómo podemos representar datos numéricos (enteros, fraccionarios). En este capítulo veremos cómo podemos representar otras clases de datos (no numéricos): textos e imágenes.

5.1. Representación de texto

5.1.1. Codificación de caracteres

Cuando escribimos texto en nuestra computadora, estamos almacenando en la memoria ciertas secuencias de bits que corresponden a los **caracteres**, o símbolos que tipeamos en nuestro teclado. Estos caracteres tienen una **representación gráfica** en nuestro teclado, en la pantalla, y 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 estándar para poder compartir información entre diferentes 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. Por este motivo, se estableció 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. Este mapeo se define en la **Tabla ASCII**. Como esta codificación usa 7 bits, hay $2^7 = 128$ posibles caracteres codificados en esta tabla.

Tabla de códigos ASCII

- Las 26 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.

Como solo dispone de 128 posibles caraceteres, esta forma de codificar caracteres 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.

Otro estándar utilizado, **ISO/IEC 8859-1**, codifica los caracteres de la mayoría de los idiomas de Europa occidental.

5.2. Representación de imagen

5.2.1. Digitalización

Introducir en la computadora una imagen analógica (tal como un dibujo o una pintura hecha a mano), o un fragmento de sonido tomado del ambiente, requiere un proceso de **digitalización**. Digitalizar es convertir en digital la información que es analógica, esto se hace convirtiendo 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 asigna un valor que codifica 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 imagen 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

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.

Una forma de representar el color en las imágenes digitales es definir, para cada pixel, tres coordenadas (valores) que describen las intensidades de luz **roja, verde y azul** que conforman el color de ese pixel.

- El valor mínimo de una coordenada representa la ausencia de ese color.
- El valor máximo de una coordenada representa la intensidad máxima de ese color.

Cuando las coordenadas se representan usando un byte, cada coordenada puede tomar un valor 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.

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. La **profundidad de color** se define como la cantidad de bits utilizados para la codificación de los colores de la imagen.

5.2.2. 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 menor cantidad 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 (como en la representación RGB), 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.

Además de la sucesión de bits que codifican los colores de los pixels, y de conocer la profundidad de color, para poder representar la imagen, necesitamos conocer el ancho y el alto de la misma.

5.2.3. Ejemplo de formato de imagen

Teniendo en cuenta lo visto anteriormente, podemos definir un formato de imagen de la siguiente manera:

- Primera Sección (o cabecera): contiene datos acerca de la imagen (*metadatos*).
- Segunda Sección: contiene los bits que conforman la imagen.

La cabecera tiene 3 campos, con el siguiente formato:

- El primer byte de la cabecera se reserva para especificar el **ancho** de la imagen (cantidad de columnas o pixels por fila).

- El segundo byte de la cabecera se reserva para especificar la **altura** (cantidad de filas).
- El tercer byte de la cabecera se reserva para especificar la profundidad de color (cantidad de bits por pixel). Esta cantidad 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 en esa imagen.

Por 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 leer y escribir binario puede ser confuso, escribiremos esta representación en hexadecimal cuando así lo precisemos.

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 podemos 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).

5.2.4. 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**. El algoritmo de compresión puede ser de **sin pérdida**, o **con pérdida**.

5.2.5. 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.

A veces usamos compresión al consultar páginas de Internet. Muchos sitios 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 la usuaria lo note.

5.2.6. Compresión con pérdida

Cuando la compresión se hace **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 de la usuaria. La pérdida de información **es intencional**, y es la usuaria 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 la calidad del audio o del video, es decir, la cantidad de bits por segundo usados para representar el audio o la imagen. Mientras menos sean esos bits, mas rápido se podrán transferir por la red. Para disminuir la calidad del audio o sonido, se aplican métodos de **compresión con pérdida**.

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 01 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 podría ser:

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?

5.2.7. 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.

Interesante

Ley de Zipf

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.

5.2.8. 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.

Capítulo 6

Organización de Computadoras

En este capítulo veremos una descripción de los componentes de una computadora: memoria principal, procesador, buses de interconexión, dispositivos de entrada y salida. Veremos sus principales funciones y cómo se interrelacionan.

6.1. Componentes de una computadora

1. Memoria principal: Aquí se almacenan las instrucciones y los datos. 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 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.

2. Procesador: También llamado *Unidad Central de Procesamiento (CPU por sus siglas en inglés)*, es el lugar donde se *ejecutan* las instrucciones. Trabaja leyendo instrucciones y datos de la memoria; y ejecuta esas instrucciones que operan sobre esos datos.

   Un procesador consta de tres partes (ver figura 6.1):

   - La unidad aritmético/lógica (ALU-Aritmethic Logic Unit), que contiene los circuitos que realizan las operaciones aritméticas (como una suma) y lógicas (como un AND) con los datos.
   - La unidad de control (UC), que contiene los circuitos que coordinan las actividades de la máquina. Esta unidad 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 de registros, que contiene celdas de almacenamiento de datos (similares a las celdas de la memoria principal), denominadas registros, que se emplean para almacenar temporalmente la información dentro del procesador.

3. Dispositivos de Entrada y Salida: Son todos aquellos dispositivos que conectamos a la computadora para que se pueda comunicar con el exterior. Son los dispositivos que permiten ingresar datos e intrucciones, y visualizar resultados.

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

4. Buses: Los componentes mencionados anteriormente se interconectan mediante buses. Existen diferentes clases de buses:

   - **Buses internos de la CPU** para comunicar la UC y la ALU. Cuando la UC disponga que se debe ejecutar una instrucción, tal como una suma, enviará los datos y la instrucción a la ALU a través de un bus interno.
   - **Buses de sistema** que relacionan la CPU y la memoria,
   - **Buses de Entrada/Salida** para comunicar todo el sistema con los dispositivos de entrada o de salida.

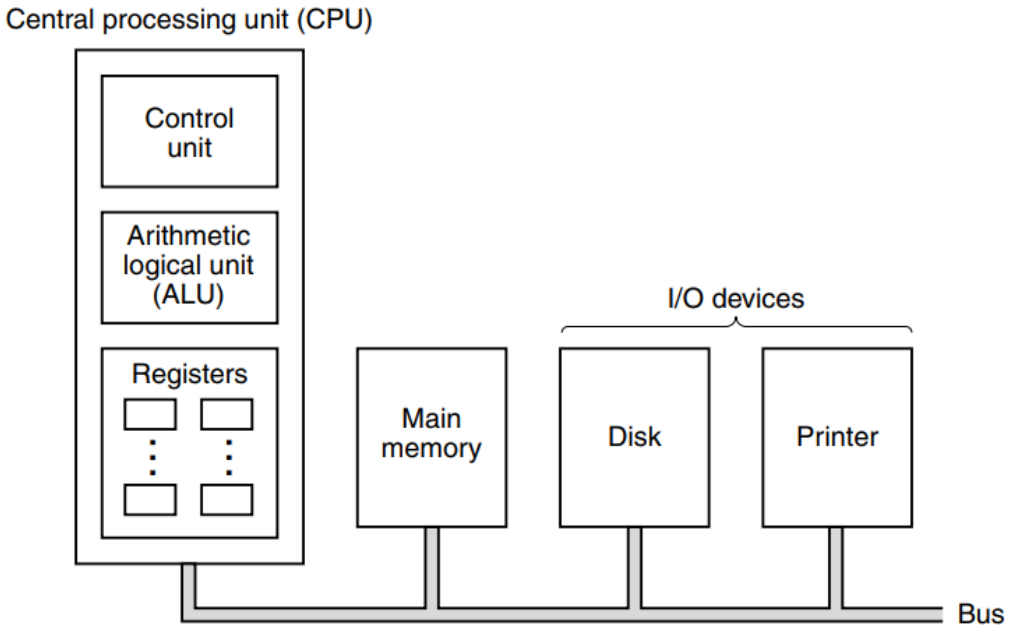


Figura 6.1: Organización de una computadora.

6.2. Arquitectura de von Neumann

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.

6.3. ISA: Conjunto de Instrucciones

Se denomina ISA (Instruction set architecture) al conjunto de instrucciones que ejecuta la CPU. Las instrucciones máquina son códigos binarios, de los cuales la CU decodifica el código de operación y los parámetros. Dos computadoras pueden compartir el mismo conjunto de instrucciones y arquitectura, pero tener distinta organización. Hay arquitecturas donde todas las instrucciones tienen el mismo tamaño (en bits), otras de tamaño variable.

6.4. Ciclo de Instrucción

Se denomina ciclo de instrucción a la secuencia de acciones que realiza el procesador para lograr ejecutar una instrucción del programa almacenado en memoria. La CPU contiene como mínimo 2 registros:

- PC (Program Counter): Contador de programa, contiene la dirección de la próxima instrucción a ejecutar.
- IR (Instruction Register): Donde se almacena la instrucción a ejecutar.

Básicamente, la CPU recupera de la memoria el dato almacenado en la posición (de memoria) indicada por el registro PC, y lo almacena en el registro IR. Luego, la Unidad de Control (UC) decodifica la instrucción almacenada en el registro IR y ordena la ejecuión de la instrucción en el procesador. Se actualiza el PC para buscar la siguiente instrucción en el siguiente ciclo.

Un ciclo de instrucción típico tiene 5 pasos característicos:

- Fetch: este paso consiste en leer la próxima instrucción a ejecutarse desde la memoria.
- Decode: en este paso se analiza el código binario de la instrucción para determinar qué se debe realizar (cuál operación, con qué operandos y donde guardar el resultado).
- Read: en este paso se accede a memoria para traer los operandos
- Execute: es la ejecución de la operación por parte de la ALU sobre los operandos
- Write: en el último paso se escribe el resultado en el destino indicado en la instrucción.

6.5. Estados del procesador (o CPU)

Como vimos anteriormente, dentro del procesador se encuentran los 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 un procesador, sus registros tendrán un cierto conjunto de valores. Ese conjunto de valores se llama **estado** de la CPU. La CPU, mientras procesa, 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.

Capítulo 7

Modelo Computacional Binario Elemental

Para comprender cómo funciona 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.

7.1. 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.

7.1.1. 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

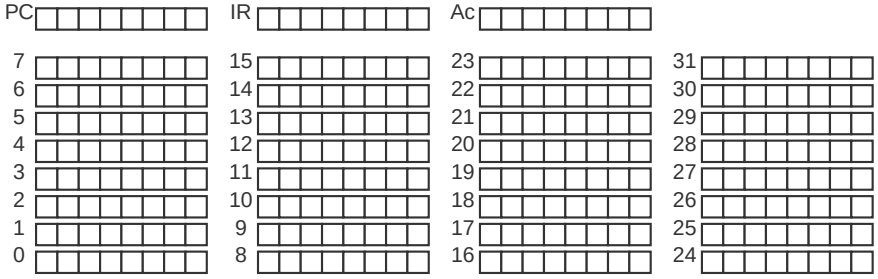


Figura 7.1: Esquema del MCBE

registros en un momento dado. Es decir, el estado de la máquina en cada momento, en MCBE, 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.

7.1.2. 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.

7.1.3. 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 están ubicados cerca del procesador y tienen tiempo de acceso mucho más rápido que la memoria. 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.

7.1.4. CPU del MCBE

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**.

7.1.5. Formato de instrucciones del MCBE

Una instrucción es una secuencia de bits (como todo!), por lo tanto debemos conocer cómo se define el formato de la instrucción para poder interpretarla. 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 (también de 8 bits). Cada instrucción se divide en dos partes:

- Código de instrucción: Son los 3 primeros bits de la instrucción.
- Argumentos u operandos: Son los 5 bits restantes, y representan el dato conel cual tiene que operar esa instrucción.

A su vez, los operandos pueden ser de dos clases: o bien **direcciones**, o bien **desplazamientos**. Si el operando es una dirección, su valor indica la dirección de un dato. Si es un desplazamiento, su valor indica **una cantidad de posiciones** que hay que saltar (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.

7.2. 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 MCBE:

- 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

7.2.1. 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.

7.2.2. 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.

7.2.3. 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.

7.2.4. 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.

7.3. Ciclo de instrucción

Ahora podemos definir el **ciclo de instrucción** de MCBE. MCBE siempre 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.

7.4. 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.

Ejemplo de 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.

Búsqueda de la instrucción		Decodificación de la instrucción		Ejecución de la instrucción			
PC	IR	Cod. Op.	Operando	Acumulador	Memoria	Salida	PC
00000000	01000111	010	00111	00000010	-	-	00000001
00000001	01111111	011	11111	-	-	00000010	00000010
00000010	10001000	100	01000	00000001	-	-	00000011
00000011	01100111	011	00111	-	(00000111)←00000001	-	00000100
00000100	11100010	111	00010	-	-	-	00000101
00000101	11011011	110	11011	-	-	-	00000000
00000000	01000111	010	00111	00000001	-	-	00000001
00000001	01111111	011	11111	-	-	00000001	00000010
00000010	10001000	100	01000	00000000	-	-	00000011
00000011	01100111	011	00111	-	(00000111)←00000000	-	00000100
00000100	11100010	111	00010	-	-	-	00000110
00000110	00100000	001	00000	-	-	-	-

Traza de ejecución de un programa de MCBE

- 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.

7.4.1. Traza de ejecución

¿Qué es realmente lo que hace el programa presentado anteriormente, y cuál es el resultado de ejecutarlo? Para poder saberlo, lo más conveniente es hacer una **traza del programa**. Una traza es una 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.

7.5. Para practicar para el coloquio o el final

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.

Capítulo 8

El Software

En este capítulo veremos 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.

8.1. Lenguajes de bajo nivel

8.1.1. Lenguaje máquina

En el capítulo anterior hemos visto un cómo escribir un programa para la MCBE. El lenguaje utilizado, la secuencia de 8 bits que representaban instrucciones o datos es el llamado **lenguaje de máquina** del MCBE.

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.

8.1.2. 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. En MCBE disponemos de los siguientes 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, 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. Por ejemplo, la instrucción en línea 2 del programa anterior, ST OUT, almacena el contenido del acumulador en la posición 31, lo que equivale a escribir ese contenido en la salida del MCBE.

Ensambladores

Como hemos dicho anteriormente, una CPU como el MCBE sólo sabe ejecutar instrucciones de código máquina (expresadas con unos y ceros). Vimos también que el lenguaje ensamblador es una versión legible de las instrucciones de máquina. Ahora bien, si escribimos el programa en lenguaje ensamblador, ¿cómo hacemos para que la máquina lo ejecute? Podemos pensar en un programa que automáticamente *traduzca* de un lenguaje a otro. A este programa se le conoce como **ensamblador** o **assembler**. Un ensamblador es un programa **traductor**. Los traductores traducen programas de un lenguaje a otro.

Como es de imaginar, 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 a otro procesador que no sea aquel para el cual fue ensamblado. Las instrucciones de máquina tendrán sentidos diferentes para uno y otro.

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. Estos lenguajes tan ligados a un procesador determinado, requieren conocimiento profundo de dicho procesador. Escribir un programa para resolver un problema complejo en un lenguaje de bajo nivel suele ser muy costoso en tiempo y esfuerzo. Estos programas además son poco **portables**, es decir, no pueden ejecutarse en otro procesador diferente sin re-escribir código.

Ensamblador x86

Cada CPU tiene su propio lenguaje ensamblador, y sus propios programas traductores (ensambladores). Por ejemplo, la familia de procesadores de Intel para computadoras personales comparte el mismo ISA o arquitectura (conjunto de instrucciones). Cualquiera de estos procesadores puede ser programado usando un ensamblador de la familia **x86**. Los procesadores de la familia x86 se encuentran en casi todas las computadoras personales y notebooks.

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.

Ejemplo programa ensamblador 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
```

Ensamblador ARM

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.

¿Podremos ejecutar el código máquina producido por un ensamblador para x86 en una computadora basada en otro procesador, como por ejemplo ARM? La respuesta es no. El programa en lenguaje ensamblador debería ser **portado** o traducido al ensamblador propio de ARM, por una persona programadora, y luego ensamblado con un ensamblador para ARM.

Ejemplo programa ensamblador familia 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"
```

Ensamblador PowerPC

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

Ejemplo programa ensamblador familia PowerPC

```
.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
```

8.2. Lenguajes de alto nivel

Para facilitar la programación (en los casos que no es necesario programar en bajo nivel) existen lenguajes de **alto nivel**. Estos lenguajes ocultan a la programadora 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 mucho más fácil.

8.2.1. Compiladores e intérpretes

Los traductores de lenguajes de alto nivel pueden funcionar de dos maneras:

- Compiladores: Producen una versión en código máquina del programa fuente. Este programa resultante puede ejecutarse (n cantidad de veces) sin necesidad de compilar cada vez.
- Intérpretes: Alternan las fases de traducción y ejecución, por lo cual la ejecución completa del mismo programa tardará algo más de tiempo. Traducen cada linea y la ejecutan cada vez que se ejecuta el programa. El programa interpretado debe ser traducido cada vez que se ejecute.

Al igual que el compilador, el intérprete traduce un programa fuente escrito en algún lenguaje de alto nivel, pero con la diferencia que cada instrucción es ejecutada inmediatamente, sin generar un programa en lenguaje de máquina.

Velocidad de ejecución

- 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.

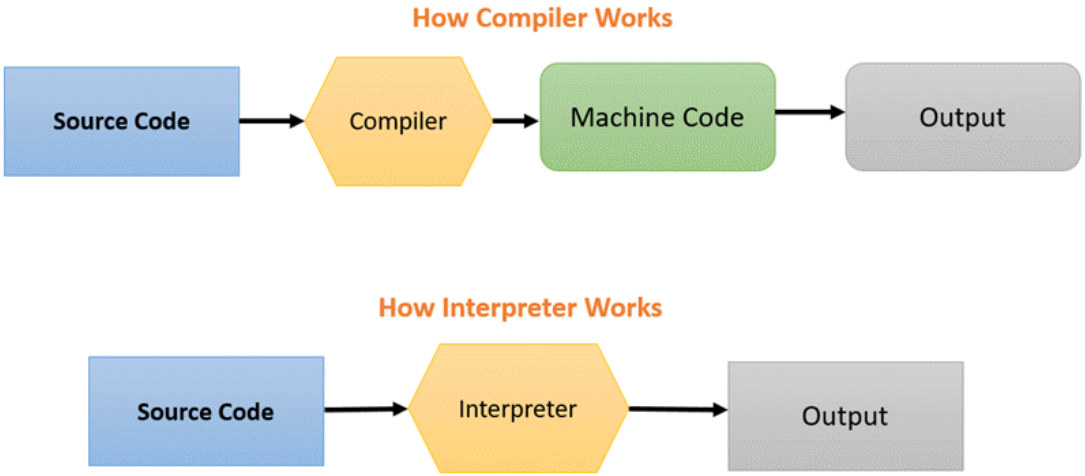


Figura 8.1: Cómo trabaja el compilador y el intérprete.

Portabilidad

- 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 el código de máquina de alguna arquitectura específica, así que no será compatible con otras.

Ciclo de compilación

Cuando utilizamos un **compilador** para obtener un programa **ejecutable**, el programa que 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. Este archivo objeto puede no estar completo, ya que el programador puede hacer uso de rutinas o funciones que vienen provistas con el sistema (de modo de no especificar cómo se realizan esas funciones cada vez). 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 quien programa 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.

8.2.2. Fases del ciclo de compilación

Quien programa necesita producir un archivo ejecutable y para ello utilizará varios programas de sistema como editores, traductores, vinculadores. Por otro lado, hará llamadas a funciones de biblioteca. Las bibliotecas consisten 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. El resultado final del ciclo de compilación es un **ejecutable**. La figura 8.2 muestra un esquema del ciclo.

Resumiendo:

- La primera fase del ciclo de compilación es 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.

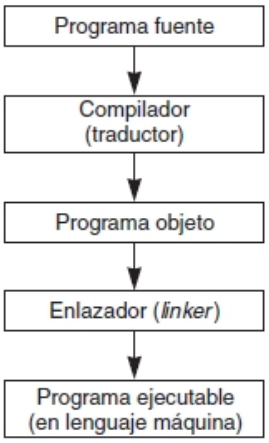


Figura 8.2: Ciclo de compilación.)

Entornos de desarrollo o IDE

Muchas personas utilizan algún **ambiente integrado de desarrollo (IDE)**, que es un programa que actúa como intermediario entre la persona usuaria 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.

Capítulo 9

Sistemas Operativos

En este capítulo veremos de qué manera evolucionó el software de base asociado a los sistemas de cómputo de diferentes momentos históricos hasta llegar a los actuales sistemas operativos. Un **sistema operativo** es un software cuya función principal es la de ser intermediario entre las personas usuarias y el hardware del sistema de cómputo.

9.1. Evolución del software de base

9.1.1. Open Shop

Las primeras computadoras estaban dedicadas a una única tarea, perteneciente a una única persona usuaria. Podían ser utilizadas por diferentes usuarixs, pero cada unx debía esperar su turno para reprogramarlas manualmente, lo cual era laborioso y se llevaba gran parte del tiempo por el cual esos usuarixs pagaban.

9.1.2. Sistemas Batch

Una vez que se popularizaron las máquinas de programa almacenado, se pudo minimizar el tiempo ocioso que presentaban los sistemas de cómputo anteriores, adoptando **esquemas de carga automática** de trabajos. Un trabajo típico consistía en la compilación y ejecución de un programa, o la carga de un programa compilado más un lote de datos de entrada, y la impresión de un cierto resultado de salida del programa. Estos trabajos estaban definidos por conjuntos o lotes de tarjetas perforadas, de ahí su nombre de trabajos **por lotes** o, en inglés, *batch*.

9.1.3. Sistemas Multiprogramados

Más adelante, conforme las tecnologías permitían ir aumentando la velocidad de procesamiento, se notó que los procesadores quedaban desaprovechados gran parte del tiempo debido a la inevitable **actividad de entrada/salida**. Así se idearon sistemas que optimizaban la utilización de la CPU, al poder cargar más de un programa en la memoria y poder conmutar el uso del procesador entre ellos. Éstos fueron los primeros **sistemas multiprogramados**.

9.1.4. Sistemas de Tiempo Compartido

Una vez que llegó la posibilidad de tener varios programas coexistiendo simultáneamente en la memoria, se buscó que la conmutación del uso del procesador entre ellos fuera tan rápida, que pareciera que cada programa funcionaba sin interrupciones. Aunque el sistema era de **tiempo**

compartido, el usuario utilizaba la computadora como si estuviera dedicada exclusivamente a correr su programa. Así los sistemas multiprogramados se volvieron **interactivos**.

9.1.5. Computación personal

Todas éstas fueron innovaciones de software, y fueron estableciendo principios y técnicas que serían adoptadas en lo sucesivo. Con la llegada de la computación personal, los sistemas de cómputo eran de capacidades modestas. Los **sistemas operativos** que permitían la ejecución de aplicaciones de lxs usuarixs en estos sistemas de cómputo comenzaron pudiendo correr una sola aplicación por vez y de unx solx usuarix; es decir, se trataba de sistemas **monotarea** y **monousuarix**.

Sin embargo, con la industria de las computadoras personales y la del software para computadoras personales traccionándose una a la otra, aparecieron sistemas operativos **multiusuario** y **multitarea**, sumamente complejos, que se convirtieron en un nuevo terreno para ensayar y mejorar las tecnologías de software y hardware.

9.1.6. Preguntas de repaso para el coloquio o el final

- ¿Cuáles son los cinco momentos evolutivos del software de base que reconocemos?
- ¿A qué se llama un **trabajo batch** o lote de trabajo? ¿Qué es un **archivo batch** en el mundo de la computación personal?
- ¿Cuál fue la necesidad que impulsó la creación de sistemas *batch*?
- ¿Cuál fue la necesidad que impulsó la creación de sistemas multiprogramados?
- ¿Cuál fue la necesidad que impulsó la creación de sistemas de tiempo compartido?

9.2. Componentes del SO

Los modernos sistemas operativos tienen varios componentes bien diferenciados. Los sistemas operativos **de propósito general** normalmente se presentan en una **distribución** que contiene e integra al menos tres componentes.

- **Kernel**: Es el componente que constituye el sistema operativo propiamente dicho. El **kernel** o núcleo es esencialmente un conjunto de rutinas que permanecen siempre residentes en memoria mientras la computadora está operando. Estas rutinas intervienen en todas las acciones que tengan que ver con la operación del hardware.
- **Software de sistema**: Junto al kernel es habitual encontrar un conjunto de **programas utilitarios o software de sistema**, que no es parte del sistema operativo estrictamente hablando, pero que en general es indispensable para la administración y mantenimiento del sistema.
- **Interfaz de usuario**: También se encuentra junto a este software del sistema alguna forma de **interfaz de usuario**, que puede ser gráfica o de caracteres. Esta interfaz de usuario se llama en general **shell**, especialmente cuando la interfaz es un procesador de comandos, basado en caracteres, y los comandos se tipean.

Hay algunas excepciones a esta estructura de componentes, por ejemplo, en los sistemas operativos **empotrados** o **embebidos** (*embedded systems*), que están ligados a un dispositivo especial y muy específico, como es el caso de algunos robots, instrumental médico, routers, electrodomésticos avanzados, etc. Estos sistemas operativos constan de un kernel que tiene la misión

de hacer funcionar cierto hardware especial, pero no necesariamente incluyen una interfaz de usuario (porque el usuario no necesita en realidad comunicarse directamente con ellos) o no incluyen software de sistema porque sus usuarios no son quienes se encargan de su mantenimiento.

Por otro lado, un típico sistema operativo multipropósito debe dar soporte entonces a la actividad de una gran variedad de aplicaciones. No solamente a la interfaz de usuario o procesador de comandos, más el software de sistema incluido, sino también a toda la gama de aplicaciones que desee ejecutar el usuario, como programas de comunicaciones (navegadores, programas de transferencia de archivos, de mensajería); aplicaciones de desarrollo de programas (compiladores, intérpretes de diferentes lenguajes), etc.

9.3. Kernel del SO

9.3.1. Funciones del Kernel

- El kernel se encarga de la administración y control del hardware o conjunto de recursos físicos. Los **recursos físicos** del sistema son todos los elementos de hardware que pueden ser de utilidad para el software, como la CPU, la memoria, los discos, los dispositivos de entrada/salida, etc.
- El kernel se encarga de la administración y control de los recursos lógicos. Los **recursos lógicos**, son recursos que no son físicos, como los archivos.
- El kernel se encarga de poner en ejecución a los programas que se encuentran almacenados en el sistema. Cuando un programa está en ejecución, lo llamamos un **proceso**. El kernel controla la creación, ejecución y finalización de los procesos.

9.3.2. Modo dual de operación

Si los procesos de usuarix pudieran utilizar directamente los recursos en cualquier momento y sin coordinación, los resultados podrían ser desastrosos. Por ejemplo, si dos o más programas quisieran usar la impresora al mismo tiempo, en el papel impreso se vería una mezcla de las salidas de los programas que no serviría a ninguno de ellos. Como el sistema operativo debe coordinar el acceso de los diferentes procesos a esos recursos, resulta necesario que cuente con alguna forma de imponer conductas y límites a esxs usuarixs y programas, para evitar que algunx tome control del sistema en perjuicio de lxs demás. Para garantizarle este poder por sobre lxs usuarixs, el sistema operativo requiere apoyo del hardware: su código se ejecuta en un modo especial de operación del hardware, el **modo privilegiado** del procesador.

Los modernos procesadores funcionan en lo que llamamos **modo dual** de ejecución, donde el ISA se divide en dos grupos de instrucciones:

- Instrucciones privilegiadas: Ciertas instrucciones que controlan el modo de operación de la CPU, el acceso a memoria, o a las unidades de Entrada/Salida, pertenecen al grupo de instrucciones del **modo privilegiado**.
- Instrucciones no privilegiadas: Un programa que se está ejecutando funciona en modo **no privilegiado** y tiene acceso a la mayoría de las instrucciones del ISA, pero no a las instrucciones del modo privilegiado.

9.3.3. Llamadas al sistema

El kernel ofrece su capacidad de control de todos los recursos a los procesos o programas en ejecución, quienes le solicitan determinadas operaciones sobre esos recursos. Por ejemplo,

un proceso que necesita utilizar un dispositivo de entrada/salida, o un recurso lógico como un archivo, hace una **petición de servicio, llamada al sistema, o system call**, solicitando un servicio al sistema operativo. El servicio puede tratarse de una operación de lectura, escritura, creación, borrado, etc. El sistema operativo centraliza y coordina estas peticiones de forma que los procesos no interfieran entre sí en el uso de los recursos.

El procesador ejecutará instrucciones del programa en ejecución en modo no privilegiado hasta que éste necesite un servicio del sistema operativo, tal como el acceso a un recurso físico o lógico. Para requerir este servicio, el proceso ejecuta una instrucción de **llamada al sistema** o **system call**, que es la única instrucción del conjunto no privilegiado que permite a la CPU conmutar al modo privilegiado.

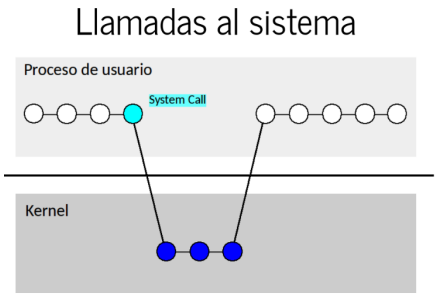


Figura 9.1: Funcionamiento de un system call.

La llamada al sistema conmuta el modo de la CPU a modo privilegiado y además fuerza el salto a una cierta dirección fija de memoria donde existe código del kernel. En esa dirección de memoria existe una rutina de atención de llamadas al sistema, que determina, por el contenido de los registros de la CPU, qué es lo que está solicitando el proceso. Con estos datos, esa rutina de atención de llamadas al sistema dirigirá el pedido al **subsistema del kernel** correspondiente, ejecutando siempre en modo privilegiado, y por lo tanto, con completo acceso a los recursos. El subsistema que corresponda hará las verificaciones necesarias para cumplir el servicio:

- ¿El proceso solicitante tiene los permisos necesarios?
- ¿El recurso está disponible o está siendo usado por otro proceso?
- ¿Los argumentos proporcionados por el proceso son razonables para el servicio que se pide?, etc.

Si se cumplen todos los requisitos, se ejecutará el servicio pedido y luego se volverá a modo no privilegiado, a continuar con la ejecución del proceso.

9.4. Una cronología de los SO

Entre la década de 1960 y principios del siglo XXI surgieron gran cantidad de innovaciones tecnológicas en el área de sistemas operativos. Muchas de ellas han tenido éxito más allá de los fines experimentales y han sido adoptadas por sistemas operativos con gran cantidad de usuarios. Diferentes sistemas operativos han influido en el diseño de otros posteriores, creándose así líneas genealógicas de sistemas operativos. Es interesante seguir el rastro de lo que ocurrió con algunos sistemas importantes a lo largo del tiempo, y ver cómo han ido reconvirtiéndose unos sistemas en otros.

- Por ejemplo, el sistema de archivos diseñado para el sistema operativo CP/M de la empresa Digital, en los años 70, fue adaptado para el MS-DOS de Microsoft, cuya evolución final fue **Windows**.

  Los diseñadores de Windows NT fueron los mismos que construyeron el sistema operativo VMS de los equipos VAX, también de Digital, y aportaron su experiencia. De hecho,

muchas características de la gestión de procesos y de entrada/salida de ambos sistemas son idénticas.

- Otra importante línea genealógica es la que relaciona el antiguo Multics, por un lado, con **Unix** y con Linux; y más recientemente, con el sistema para plataformas móviles Android.

  Unix fue el primer sistema operativo escrito casi totalmente en un lenguaje de alto nivel, el **C**, lo cual permitió portarlo a diferentes arquitecturas. Esto le dio un gran impulso y la comunidad científica lo adoptó como el modelo de referencia de los sistemas operativos de tiempo compartido.

  En 1991 **Linus Torvalds** lanzó un proyecto de código abierto dedicado a la construcción de un sistema operativo compatible con Unix pero sin hacer uso de ningún código anteriormente escrito, lo que le permitió liberarlo bajo una licencia libre. La consecuencia es que Linux, su sistema operativo, rápidamente atrajo la atención de centenares de desarrolladores de todo el mundo, que sumaron sus esfuerzos para crear un sistema que fuera completo y disponible libremente.

  Linux puede ser estudiado a fondo porque su código fuente no es secreto, como en el caso de los sistemas operativos propietarios. Esto lo hace ideal, entre otras cosas, para la enseñanza de las Ciencias de la Computación. Esta cualidad de sistema abierto permitió que otras compañías lo emplearan en muchos otros proyectos.

- Otra empresa de productos de computación de notable trayectoria, **Apple**, produjo un sistema operativo para su línea de computadoras personales Macintosh. Su sistema MacOS estaba influenciado por desarrollos de interfaces de usuario gráficas realizadas por otra compañía, Xerox, y también derivó en la creación de un sistema operativo para dispositivos móviles.

- Otros sistemas operativos han cumplido un ciclo con alguna clase de final, al no superar la etapa experimental, haberse transformado definitivamente en otros sistemas, desaparecer del mercado o quedar confinados a cierto nicho de aplicaciones. Algunos, por sus objetivos de diseño, son menos visibles, porque están destinados a un uso que no es masivo, como es el caso del **sistema de tiempo real** QNX.

9.5. Servicios del SO

Después de conocer estas cuestiones generales sobre los sistemas operativos, veremos con un poco más de detalle los diferentes **servicios** provistos por los principales subsistemas de un SO:

- Gestión de procesos
- Gestión de memoria
- Gestión de archivos
- Operaciones de Entrada/Salida
- Protección

Si bien la discusión que sigue es suficientemente general para comprender básicamente el funcionamiento de cualquier sistema operativo moderno, nos referiremos sobre todo a la manera como se implementan estos subsistemas y servicios en la familia de sistemas **Unix**, que, como hemos dicho, es el modelo de referencia académico para la mayoría de la investigación y desarrollo de sistemas operativos. Veremos una introducción a los tres primeros servicios únicamente.

9.5.1. Gestión de procesos

Creación de procesos

¿Cómo se inicia la ejecución de un proceso? Inicialmente, el SO crea una cantidad de procesos de sistema. Uno de ellos es un **shell** o interfaz de usuario. Este proceso sirve para que el usuario pueda comunicarse con el SO y solicitarle la ejecución de otros procesos. El *shell* puede ser **gráfico**, con una interfaz de ventanas; o **de texto**, con un **intérprete de comandos**. En cualquiera de los dos casos, de una forma u otra, usando el shell el usuario selecciona algún **programa**, que está residiendo en algún medio de almacenamiento como los discos, y pide al SO que lo ejecute. En respuesta a la petición de la usuaria, el SO carga ese programa en memoria y pone a la CPU a **ejecutar el código** de ese programa.

Una vez que el programa está en ejecución, decimos que tenemos un nuevo **proceso** activo en el sistema. Este nuevo proceso es un **hijo** del shell, y a su vez puede crear nuevos procesos hijos si es necesario. Todo proceso es **hijo** de algún otro proceso que lo crea.

Estados de los procesos

Cada proceso pasará por varios estados durante su vida. Básicamente, una vez creado el proceso pasa a estar en estado **listo**, cuando el sistema operativo le asigna el procesador a ese proceso, pasa a estado **ejecutando**, cuando el proceso realiza una operación de entrada/salida pasa a estado **en espera**.

Ciclo de estados en un sistema multiprogramado

Como vimos en la subsección 9.1.3, en un sistema **multiprogramado**, varios procesos pueden estar presentes en la memoria del sistema de cómputo. Durante su vida en el sistema, cada proceso atravesará un ciclo de estados.

- Cuando recién se crea un proceso, su estado es **listo**, porque está preparado para recibir la CPU cuando el planificador o scheduler lo disponga.
- En algún momento recibirá la CPU y pasará a estado **ejecutando**.
- El proceso en algún momento requerirá servicios del SO (por ejemplo, para operaciones de entrada/salida, como recibir datos por el teclado o por la red, imprimir resultados, etc.). Durante estas operaciones de entrada/salida, el proceso no utilizará la CPU para realizar cómputos, sino que deberá esperar el final de este servicio del SO. Como las operaciones de entrada/salida son mucho mas lentas (por varias magnitudes) que la velocidad en que la CPU procesa instrucciones, el sistema pone en estado de **espera** al proceso hasta que finalice la operación de entrada/salida.
- Mientras tanto, como la CPU ha quedado libre, el SO aprovecha la oportunidad de darle la CPU a algún otro proceso que esté en estado **listo**.
- Cuando finalice una operación de entrada/salida que ha sido requerida por un proceso, este proceso volverá al estado de **listo** y esperará que algún otro proceso libere la CPU para volver a **ejecutando**.
- Al volver al estado **ejecutando**, el proceso retomará la ejecución desde la instrucción inmediatamente posterior a la que solicitó el servicio del SO.
- En algún momento, el proceso ejecutará la última de sus instrucciones y finalizará. El SO libera los recursos que utilizó para su gestión.

Figura 9.2: Primer proyecto que implementó un sistema de tiempo compartido (1957) IBM 704 modificado.

Ciclo de estados en un sistema de tiempo compartido

Como vimos en la subsección 9.1.4, los sistemas **de tiempo compartido** están diseñados para ser **interactivos**, y tienen la misión de hacer creer a cada usuarix que el sistema de cómputo está dedicado exclusivamente a sus procesos. Sin embargo, normalmente existen muchísimos procesos activos simultáneamente en un SO de propósito general. Para lograr esto el planificador de estos SO debe ser capaz de hacer los cambios de estado con mucha velocidad. El resultado es que lxs usuarixs prácticamente no perciben estos cambios de estado. En un sistema **de tiempo compartido**, el ciclo de estados de los procesos es similar al del sistema multiprogramado, pero con una importante diferencia.

- El sistema de tiempo compartido tiene la capacidad de **desalojar** a un proceso de la CPU, sin necesidad de esperar a que el proceso solicite un servicio del SO.
- Para esto, el SO define un **quantum** o tiempo máximo de ejecución (típicamente de algunos milisegundos), al cabo del cual el proceso obligatoriamente deberá liberar la CPU.
- El SO, al entregar la CPU a un proceso que pasa de listo a ejecutando, pone en marcha un reloj para medir un quantum de tiempo que pasará ejecutando el proceso.
- Al agotarse el quantum, el SO **interrumpirá** al proceso y lo pondrá en el estado de **listo**. Al quedar libre la CPU, el siguiente proceso planificado entrará en estado de ejecución.
- Si un proceso decide solicitar un servicio del SO antes de que se agote su quantum, el ciclo continuará de la misma manera que en el sistema multiprogramado, pasando a estado **en espera** hasta que finalice el servicio.

Comparando multiprogramación y *time sharing*

Notemos que los diagramas de estados del **sistema multiprogramado** y del sistema **de tiempo compartido** se diferencian sólo en una transición: la que lleva del estado de **ejecutando** al de **listo** en este último sistema.

- En un sistema multiprogramado, un proceso sólo abandona la CPU cuando ejecuta una petición de servicio al SO.
- En un sistema de tiempo compartido, un proceso abandona la CPU cuando ejecuta una petición de servicio al SO **o bien** cuando se agota su quantum.

Preguntas para estudiar para el coloquio o el final

- ¿Por qué un proceso que ejecuta una solicitud de entrada/salida no pasa directamente al estado de **listo**?
- ¿En qué radica la diferencia entre el scheduling de un sistema multiprogramado y el de un sistema de tiempo compartido?
- Si en un sistema multiprogramado se ejecuta un programa escrito de forma que **nunca** ejecuta una operación de entrada/salida, ¿liberará alguna vez la CPU durante su vida?
- ¿Y en un sistema de tiempo compartido?

Concurrencia y paralelismo

Si el sistema de tiempo compartido dispone de **una sola unidad de ejecución o CPU**, habrá solamente **un proceso ejecutándose** en cada momento dado, pero muchos procesos podrán desarrollar su vida al mismo tiempo, alternándose en el uso de esa CPU.

- Cuando los procesos coexisten en el sistema simultáneamente pero se alternan en el uso de **una única CPU** decimos que esos procesos son **concurrentes**. Todos están activos en el sistema durante un período de tiempo dado; sin embargo, no hay dos procesos en estado de ejecución en el mismo momento, por lo cual no podemos decir que se ejecutan *simultáneamente*.
- Cuando el sistema de cómputo tiene **más de una CPU**, entonces podemos tener dos o más procesos en estado de ejecución **simultáneamente**, y entonces decimos que esos procesos son **paralelos**.

Comandos Linux

Monitorización de procesos

Los sistemas operativos suelen ofrecer herramientas para monitorizar o controlar los procesos del sistema.

Comando top

En Linux, el comando **top** ofrece una vista de los procesos, información acerca de los recursos que están ocupando, y algunas estadísticas globales del sistema. Es conveniente consultar el manual del comando (man top) para investigar a fondo los significados de cada uno de los datos presentados en pantalla.

Estadísticas globales

En el cuadro superior, **top** muestra:

- El **tiempo** de funcionamiento desde el inicio del sistema.
- La cantidad de usuarixs activos.
- La **carga promedio** (longitud de la cola de procesos listos) medida en tres intervalos de tiempo diferentes.
- La cantidad de procesos o tareas en actividad y en diferentes estados.
- Las estadísticas de uso de la CPU, contabilizando los tiempos:
  - **De usuarix**
  - **De sistema**

  - **De nice** o de cortesía.
  - **Ocioso**
  - **De espera**
  - Tiempos imputables a **interrupciones**
- Estadísticas de memoria RAM total, usada y libre, y ocupada por buffers de entrada/salida
- Estadísticas de espacio de intercambio o **swap** total, usado y libre.

Estadísticas de procesos

Para cada proceso, el programa top muestra:

- El **PID** o identificación de proceso.
- El **usuario dueño** del proceso.
- La **prioridad** a la cual está ejecutando y el **valor de nice** o de cortesía
- El tamaño del **espacio virtual** del proceso y los tamaños del **conjunto residente** y **regiones compartidas**.
- El **porcentaje de CPU** recibido durante el ciclo de top.
- El **porcentaje de memoria del sistema** ocupada.
- El **tiempo** de ejecución que lleva el proceso desde su inicio.
- El **comando** u orden con la que fue creado el proceso.

El comando top tiene muchas opciones y comandos interactivos. Uno de ellos muestra los datos de sistema desglosados por CPU o unidad de ejecución.

Comandos de procesos

En los sistemas operativos de la familia de Unix encontramos un rico conjunto de comandos de usuario destinados al control de procesos. Algunos interesantes son:

- ps y pstree listan los procesos activos en el sistema.
- nice cambia la prioridad de un proceso.
- kill envía una señal a un proceso para terminarlo.

Interesante

Administración de procesos

Prioridades de ejecución de procesos

9.5.2. Gestión de archivos

La información que se guarda en medios de almacenamiento permanente, como los **discos**, se organiza en **archivos**, que son secuencias de bytes. Estos bytes pueden estar codificando cualquier clase de información: texto, código fuente de programas, código ejecutable, multimedia, etc. Cualquier pieza de información que sea tratable mediante las computadoras puede ser almacenada y comunicada en forma de archivos.

El componente del SO responsable de los servicios relacionados con archivos es el llamado **sistema de archivos** o **filesystem**. En general, el filesystem no se ocupa de cuál es el contenido de los archivos, o de qué sentido tienen los datos que contienen. Son las aplicaciones quienes tienen conocimiento de cómo interpretar y procesar los datos contenidos en los archivos. En cambio, el filesystem mantiene información **acerca** de los archivos: en qué bloques del disco están almacenados, qué tamaño tienen, cuándo fueron creados, modificados o accedidos por

última vez, qué usuarios tienen permisos para ejecutar qué acciones con cada uno de ellos, etc. Como todos estos datos son **acerca de los archivos**, y no tienen nada que ver con los datos **contenidos en** los archivos, son llamados **metadatos**. El sistema de archivos o filesystem mantiene tablas y listas de metadatos que describen los archivos contenidos en un medio de almacenamiento.

Una característica compartida por la mayoría de los sistemas de archivos es la organización jerárquica de los archivos en estructura de **directorios**. Los directorios son contenedores de archivos (y de otros directorios). Los directorios han sido llamados, en la metáfora de las interfaces visuales de usuario, **carpetas**.

En rigor de verdad, el nombre de sistema de archivos o filesystem designa varias cosas, relacionadas pero diferentes:

- **Una pieza de software**: El filesystem es el subsistema o conjunto de rutinas del kernel responsable de la organización de los archivos del sistema. Es un componente de software o módulo del kernel, y como tal, es **código** ejecutable.

- **Un conjunto de metadatos**: El filesystem también es un conjunto de metadatos acerca de los archivos grabados en un medio de almacenamiento. El filesystem en este sentido, es la **información** que describe unos archivos y reside en el mismo medio de almacenamiento que ellos.

- **Un conjunto de características**: Además, cuando se diseña un sistema de archivos, se lo dota de ciertas capacidades distintivas. Al referirnos al filesystem, podemos estar hablando del **conjunto de características ofrecidas** por alguna implementación en particular de un sistema de archivos.

    - Algunos sistemas de archivos específicos tienen ciertas restricciones en la forma de los nombres de los archivos, y otros no.
    - Algunos permiten la atribución de permisos o identidades de usuario a los archivos, y otros no.
    - Algunos ofrecen servicios como encriptación, compresión, o versionado de archivos.

Árbol de directorios

En los filesystems de tipo Unix, la organización de los directorios es jerárquica y recuerda a un árbol con **raíz** y ramas. Algunos directorios cumplen una función especial en el sistema porque contienen archivos especiales, y por eso tienen nombres establecidos.

- Por ejemplo, el directorio raíz, donde se origina toda la jerarquía de directorios, tiene el nombre especial `/`.
- El directorio lib (abreviatura de **library** o biblioteca) contiene bibliotecas de software.
- Los directorios bin, sbin. /usr/bin, etc., contienen archivos ejecutables (a veces llamados **binarios**).

Nombres de archivo y referencias

Los nombres completos, o **referencias absolutas**, de los archivos y directorios se dan indicando cuál es el camino que hay que recorrer, para encontrarlos, **desde la raíz** del sistema de archivos.

Ejemplo

La referencia absoluta para el archivo texto.txt ubicado en el directorio juan, que está dentro del directorio home, que está dentro del directorio raíz, es /home/juan/texto.txt.

Una **referencia relativa**, por otro lado, es una forma de mencionar a un archivo que depende de dónde está situado el proceso o usuario que quiere utilizarlo. Todo proceso, al ejecutarse, tiene una noción de lugar del filesystem donde se encuentra.

Comandos Linux

- Al iniciar un shell, el directorio actual es el directorio **home**, que es el espacio privado del usuario. Éste es el **directorio actual** del proceso shell.
- Para cambiar de directorio se usa el comando *$cd directorio destino.*
- El comando *pwd* dice cuál es el directorio actual de un shell.

La referencia relativa de un archivo indica cuál es el camino que hay que recorrer para encontrarlo **desde el directorio actual** del proceso. Para el mismo archivo del ejemplo anterior, si el directorio activo del shell es /home/juan, la referencia relativa será simplemente texto.txt. La referencia es relativa porque, si el proceso cambia de directorio activo, ya no servirá como referencia para ese mismo archivo.

Listar el contenido de un directorio

El comando *ls -l* muestra los nombres de los archivos contenidos en un directorio. El listado se compone de varios elementos por cada fila, separados por espacios.

```
$ ls -l util
total 60
-rwxr-xr-x 1 oso oso   40 Apr 18 16:54 github
-rw-r--r-- 1 oso oso 1337 May  4 12:11 howto.txt
-rwxrwxr-x 1 oso oso  458 Feb  9 16:28 macro
drwxr-xr-x 2 oso oso 4096 May 26 18:14 prueba
-rwxr-xr-x 1 oso oso   30 Feb  9 16:28 server
```

Tipo de archivo

El primer elemento de cada fila contiene un carácter que indica el tipo del archivo, y los siguientes caracteres indican los permisos asignados al archivo. El tipo indicado por el **guión** es **archivo regular**, y el indicado por la letra **d** es **directorio**. En el ejemplo, todos los archivos son regulares salvo **prueba**, que es un directorio. Otros tipos de archivo tienen otros caracteres indicadores.

Permisos

Los permisos de cada archivo están indicados por los nueve caracteres siguientes hasta el espacio. Para interpretarlos, se separan en tres grupos de tres caracteres. Los primeros tres caracteres indican los permisos que tiene el usuario **dueño** del archivo; los siguientes tres, los permisos para el **grupo** al cual pertenece el archivo; y los últimos tres, los permisos que tienen **otros usuarios**.

Cada grupo de permisos indica si se permite la **lectura**, **escritura**, o **ejecución** del archivo. Una letra **r** en el primer lugar del grupo indica que el archivo puede ser escrito. Una **w** en el segundo lugar, que puede ser escrito o modificado. Una **x** en el tercer lugar, que puede ser ejecutado. Cuando no existen estos permisos, aparece un carácter **guión**.

- Así, el archivo github del ejemplo tiene permisos **rwxr-xr-x**, que se separan en permisos **rwx** para el dueño (el dueño puede leerlo, escribirlo o modificarlo, y ejecutarlo), **r-x** para el grupo (cualquier usuario del grupo puede leerlo y ejecutarlo), y lo mismo para el resto de los usuarios.
- Por el contrario, el archivo howto.txt tiene permisos **rw-r–r–**, que se separan en permisos **rw-** para el dueño (el dueño puede leerlo, escribirlo o modificarlo, pero no ejecutarlo), **r–** para el grupo (cualquier usuario del grupo puede leerlo, pero no modificarlo ni ejecutarlo), y lo mismo para el resto de los usuarios.

Cuenta de links

La cuenta de links es la cantidad de **nombres** que tiene un archivo.

Dueño y grupo

Las columnas tercera y cuarta indican el usuario y grupo de usuarios al cual pertenece el archivo.

Tamaño

Aparece el tamaño en bytes de los archivos regulares.

Fecha y hora

Aparecen la fecha y hora de última modificación de cada archivo.

Nombre

El último dato de cada línea es el nombre del archivo.

Bloques de disco

El SO ve los discos como un vector de bloques o espacios de tamaño fijo. Cada bloque se identifica por su número de posición en el vector, o **dirección de bloque**. Esta dirección es utilizada para todas las operaciones de lectura o escritura en el disco.

- Cuando el SO necesita acceder a un bloque para escribir o leer sus contenidos, envía un mensaje al controlador del disco especificando su dirección.
- Si la operación es de lectura, además indica una dirección de memoria donde desea recibir los datos que el controlador del disco leerá.
- Si la operación es de escritura, indica una dirección de memoria donde están los datos que desea escribir.

9.5.3. Gestión de memoria

En un sistema multiprogramado, la memoria debe ser dividida de alguna forma entre los procesos que existen simultáneamente en el sistema. La tarea de controlar qué proceso recibe qué región de memoria, o **gestión de memoria**, es un problema con varias soluciones.

Mapa de memoria

Un programa se compone, como mínimo, de:

- Las instrucciones para el procesador (**código** o **texto** del programa).
- Los **datos** con los que operarán esas instrucciones.

Para que este programa pueda convertirse en un proceso, tanto instrucciones como datos deben estar almacenados en posiciones de **memoria principal**. Solamente de allí pueden ser leídos por el procesador. Además, los procesos requieren otros espacios de memoria para otros usos.

- Por ejemplo, al llamar a una función o rutina, es necesario guardar temporariamente la **dirección de retorno**, que es la dirección de la instrucción a la cual se debe volver una vez terminada la rutina. Estas direcciones se guardan en una zona denominada la **pila** o **stack** del proceso.
- El proceso puede necesitar crear dinámicamente **estructuras de datos** que no existían al momento de carga del programa en memoria. Estos componentes también necesitan ser almacenados en memoria, típicamente en una zona denominada el **heap**.

Todos estos componentes forman lo que a veces se llama **mapa de memoria** de cada proceso, y requieren memoria física.

Espacios de direcciones

Espacio de direcciones físicas

La memoria física del sistema se ve como un arreglo, vector o secuencia ordenada de celdas o posiciones de almacenamiento. Cada posición tiene una **dirección** que es el número con el que se la puede acceder para leer o escribir su contenido. En un sistema de cómputo, el conjunto de direcciones de la memoria física es un **espacio de direcciones físicas**.

Espacio de direcciones lógicas

Al ejecutarse un proceso, las instrucciones que va ejecutando la CPU **referenciarán** a los objetos del mapa de memoria mediante su dirección. Cada vez que la CPU necesite cargar una instrucción para decodificarla, hará una referencia a la dirección donde reside esa instrucción. Cada vez que una instrucción necesite acceder a un dato en memoria, la CPU hará una referencia a su dirección. El conjunto de todas las direcciones de estos objetos forma el **espacio de direcciones lógicas** del proceso.

Ejemplo

En nuestro modelo MCBE, los espacios físico y lógico coinciden. Como sabemos, una instrucción como **01000111** indica que se debe cargar en el acumulador el contenido de la dirección 7.

- Al ejecutarse esta instrucción, el procesador envía el número 7 al sistema de memoria para que éste le entregue el contenido de esa posición. El procesador hace una **referencia** a la dirección 7. Por lo tanto, la dirección 7 pertenece al espacio lógico del programa.
- Además, el sistema de memoria utiliza directamente el número 7 recibido de la CPU como la dirección de memoria que debe devolver. La posición física consultada por el sistema de memoria es exactamente la número 7.

Traducción de direcciones

En un sistema multiprogramado, los programas son cargados en diferentes posiciones del espacio de memoria física. Esto hace que los espacios **lógico y físico** de direcciones de un proceso, en general, no coincidan.

Para que las referencias a direcciones **lógicas** conserven el sentido deseado por el programador, el sistema utiliza alguna forma de **traducción de direcciones**. Las referencias a direcciones generadas por el procesador pertenecerán al espacio lógico del proceso; pero el mecanismo de traducción de direcciones **mapeará** esas direcciones lógicas a las direcciones físicas asignadas.

Unidad de gestión de memoria o MMU

Esta traducción tiene lugar, automáticamente, en el momento en que el procesador emite una dirección hacia el sistema de memoria, y está a cargo de un **componente especial del hardware**. Este componente se llama la **unidad de gestión de memoria** (**MMU, Memory Management Unit**).

- El sistema de memoria recibe únicamente las direcciones **físicas**, traducidas, y no sabe que el procesador ha solicitado acceder a una dirección **lógica** diferente.
- Por su parte, el procesador no sabe que la dirección física accedida es diferente de la dirección lógica cuyo acceso ha solicitado.

Si el sistema no ofreciera un mecanismo automático de traducción de direcciones, el programador necesitaría saber de antemano en qué dirección va a ser cargado su programa, y debería preparar las referencias a las direcciones de modo de que ambos espacios coincidieran.

Ejemplo

El siguiente programa sencillo en lenguaje ensamblador de MCBE hace referencias a algunas direcciones.

```
00000        LD   DATO
00001        ADD  CANT
00010        ST   SUMA
00011        HLT
00100  DATO: 10
00101  CANT: 1
00110  SUMA: 0
```

En este ejemplo, DATO, CANT y SUMA son las posiciones de memoria 4, 5 y 6. Si este programa se carga en la posición **cero** de la memoria física, los espacios físico y lógico coincidirán. Sin embargo, en un sistema multiprogramado, es posible que el proceso reciba otras posiciones de memoria física. Por ejemplo, el programa podría haber sido cargado a partir de la dirección 20 de la memoria. Entonces, la posición de memoria física donde residirá el valor SUMA no es la posición 6, sino la 26.

El mecanismo de traducción de direcciones del sistema **deberá sumar el valor base de la memoria** (en este caso, 20) **a todas las referencias a memoria generadas por el procesador** para mantener el funcionamiento deseado. **Sin** traducción de direcciones, la instrucción ST SUMA almacenará el resultado en la posición cuya dirección física es 6... ¡que pertenece a otro proceso!

9.5.4. Asignación de memoria contigua

Uno de los esquemas de asignación de memoria más simples consiste en asignar una región de memoria **contigua** (un conjunto de posiciones de memoria sin interrupciones) a cada proceso.

Si un SO utiliza este esquema de asignación de memoria, establece **particiones** de la memoria, de un tamaño adecuado a los requerimientos de cada proceso. Cuando un proceso termina, su región de memoria se libera y puede ser asignada a un nuevo proceso.

Fragmentación externa

El problema de este esquema es que, a medida que el sistema opere, las regiones que queden libres pueden ser tan pequeñas que un proceso nuevo no pueda obtener una región de tamaño suficiente, **a pesar de que exista memoria libre en cantidad suficiente** en el sistema. Este fenómeno se llama **fragmentación externa**.

El problema de este esquema es que, a medida que el sistema opere, las regiones que queden libres pueden ser tan pequeñas que un proceso nuevo no pueda obtener una región contigua de tamaño suficiente, **a pesar de que exista memoria libre en cantidad suficiente** en el sistema. Este fenómeno se llama **fragmentación externa**.

Un remedio para la fragmentación externa es la **compactación** de la memoria, es decir, reubicar los procesos que estén ocupando memoria, de manera de que sus regiones sean contiguas entre sí. De esta forma los "huecos" en la memoria se unen y se crean regiones libres contiguas grandes.

El problema con esta solución es que la reubicación de los procesos es muy costosa en tiempo. Mientras el sistema esté compactando la memoria, los procesos que estén siendo reubicados no podrán realizar otra tarea, y el sistema perderá productividad.

Esta clase de cargas extra en tareas administrativas, que quitan capacidad al sistema para atender el trabajo genuino, se llama **sobrecarga** u **overhead**.

9.5.5. Segmentación

Un esquema de asignación que reduce la fragmentación externa es el de **segmentación**. Con este esquema, el mapa de memoria del proceso se divide en trozos de diferentes tamaños, llamados **segmentos**, conteniendo cada uno un conjunto de instrucciones o de datos.

Durante la compilación de un programa fuente, el compilador distribuye los trozos de código y las estructuras de datos en distintos segmentos. Cada segmento tiene un tamaño o **límite**, calculado y especificado por el compilador, y grabado en la cabecera del archivo ejecutable resultante de la compilación.

Al cargar un programa en memoria, el SO destina cada segmento a una determinada **dirección base** física. La dirección base de cada segmento es utilizada por el mecanismo de traducción de direcciones. El dato de tamaño o límite de cada segmento es utilizado para la **protección**, asegurando que las referencias a memoria generadas por la CPU no rebasen los límites de cada segmento. De esta forma, un proceso no puede corromper el espacio de otros.

Este esquema de asignación de memoria reduce, aunque no elimina, la fragmentación externa, ya que los segmentos son más pequeños y reubicables dinámicamente.

9.5.6. Paginación

El esquema de asignación de memoria conocido como **paginación** considera la memoria dividida en regiones del mismo tamaño (**marcos** de memoria), y el espacio lógico de los procesos dividido en regiones (**páginas**) de igual tamaño que los marcos. Las páginas de los procesos se asignan individualmente a los marcos, una página por vez.

Al contrario que en un sistema de particiones, en un sistema con paginación de memoria los procesos reciben más de una región de memoria o marco. Los marcos asignados a un proceso pueden no ser contiguos.

Fragmentación interna

Bajo este esquema no hay fragmentación externa, porque, si existe espacio libre, siempre será suficiente para alojar al menos una página. Sin embargo, en general, el tamaño del espacio lógico del proceso no es exactamente divisible por el tamaño de la página; por lo tanto, puede haber algún espacio desaprovechado en las páginas asignadas. Esta condición se llama **fragmentación interna**.

Tabla de páginas

Para poder mantener la correspondencia entre marcos de memoria y páginas de los procesos, el SO mantiene una **tabla de páginas** por cada proceso.

- La tabla de páginas de cada proceso dice, para cada página del proceso, qué marco le ha sido asignado, además de otra información de control.
- La tabla de páginas puede contener referencias a marcos compartidos con otros procesos. Esto hace posible la creación de regiones de memoria compartida entre procesos.
- En particular, los marcos de memoria ocupados permanentemente por el kernel pueden aparecer en el mapa de memoria de todos los procesos.

Paginación por demanda

Debido a que los programas no utilizan sino una pequeña parte de su espacio lógico en cada momento (fenómeno llamado **localidad de referencias**), la asignación por paginación tiene una propiedad muy interesante: no es necesario que todas las páginas de un proceso estén en memoria física para que pueda ser ejecutado.

Esto permite la creación de sistemas con **paginación por demanda**, donde las páginas se cargan en memoria a medida que se necesitan. Un proceso puede empezar a ejecutarse apenas esté cargada la primera de sus páginas en memoria, sin necesidad de esperar a que todo su espacio lógico tenga memoria física asignada.

Cada proceso **demanda** al SO la carga de una página de su espacio lógico al espacio físico en el momento en que referencia algún objeto perteneciente a esa página. De esta manera la actividad de entrada/salida desde el disco a la memoria se reduce al mínimo necesario.

Utilizando 1) **paginación por demanda**, 2) agregando algunas características al mecanismo de **traducción de direcciones**, y 3) contando con un espacio de almacenamiento extra en disco para **intercambio de páginas** o **swapping**, se puede implementar un sistema de **memoria virtual**. La mayoría de los SO multipropósito para hardware con MMU utiliza esta técnica.

9.5.7. Memoria virtual

Con memoria virtual, el espacio de direcciones lógicas y el espacio físico se independizan completamente. El espacio lógico puede tener un tamaño completamente diferente del espacio físico. La cantidad de páginas de un proceso ya no se ve limitada por la cantidad de marcos de la memoria física.

- Podemos ejecutar **más procesos** de los que cabrían en memoria física si debiéramos asignar todo el espacio lógico de una vez.
- Los procesos pueden tener un tamaño de espacio lógico **más grande** de lo que permite el tamaño de la memoria física.

En un sistema de memoria virtual, la tabla de páginas mantiene, además de los números de página y de marco asociados, datos de estado sobre la condición de cada página.

- **Bit de validez**

  Indica si la página del proceso tiene memoria física asignada.

- **Bit de modificación**

  Indica si la página ha sido modificada desde que se le asignó memoria física.

El bit de validez de cada página indica si la página tiene o no asignado un marco, y es crucial para el funcionamiento del sistema de memoria virtual. Cuando la CPU genera una referencia a una página no válida, la condición que se produce se llama un **fallo de página (page fault)** y se resuelve asignando un marco, luego de lo cual el proceso puede continuar.

Además de estos bits de validez y modificación, la tabla de páginas contiene datos sobre los permisos asociados con cada página.

El mecanismo de memoria virtual funciona de la siguiente manera:

- Cada dirección virtual tiene un cierto conjunto de bits que determinan el número de página. Los bits restantes determinan el desplazamiento dentro de la página.
- Cuando un proceso emite una referencia a una dirección virtual, la MMU extrae el número de página de la dirección y consulta la entrada correspondiente en la tabla de páginas.
- Si la información de control de la tabla de páginas dice que este acceso no es permitido, la MMU provoca una condición de error que interrumpe el proceso.
- Si el acceso es permitido, la MMU computa la dirección física reemplazando los bits de página por los bits de marco.
- Si el bit de validez está activo, la página ya está en memoria física.
- Si el bit de validez no está activo, ocurre un fallo de página, y se debe asignar un marco. Se elige un marco de una lista de marcos libres, se lo marca como utilizado y se completa la entrada en la tabla de páginas. Los contenidos de la página se traerán del disco.
- La MMU entrega la dirección física requerida al sistema de memoria.
- Si la operación era de escritura, se marca la página como **modificada**.

Reemplazo de páginas

Cuando no existan más marcos libres en memoria para asignar, el SO elegirá una página **víctima** del mismo u otro proceso y la desalojará de la memoria. Aquí es donde se utiliza el bit de **modificación** de la tabla de páginas.

- Si la página víctima no está modificada, simplemente se marca como **no válida** y se reutiliza el marco que ocupaba.
- Si la página víctima está modificada, además de marcarla como no válida, sus contenidos deben guardarse en el **espacio de intercambio o swap**.

Posteriormente, en algún otro momento, el proceso dueño de esta página querrá accederla. La MMU verificará que la página no es válida y disparará una condición de **fallo de página**. La página será traída del espacio de intercambio, en el estado en que se encontraba al ser desalojada, y el proceso podrá proseguir su ejecución.

Ejemplo

Supongamos un sistema donde existen dos procesos activos, con algunas páginas en memoria principal, y una zona de intercambio en disco.

- El proceso P1 tiene asignadas cuatro páginas (de las cuales sólo la página 2 está presente en memoria principal), y P2, dos páginas (ambas presentes). Hay tres marcos libres (M4, M6 y M7) y la zona de intercambio está vacía.
- P1 recibe la CPU y en algún momento ejecuta una instrucción que hace una referencia a una posición dentro de su página 3 (que no está en memoria).
- Ocurre un fallo de página que trae del almacenamiento la página 3 de P1 a un marco libre. La página 3 se marca como válida en la tabla de páginas de P1.
- Enseguida ingresa P3 al sistema y comienza haciendo una referencia a su página 2.
- Como antes, ocurre un fallo de página, se trae la página 2 de P3 del disco, y se copia en un marco libre. Se marca la página 2 como válida y P3 continúa su ejecución haciendo una referencia a una dirección que queda dentro de su página 3.
- Se resuelve como siempre el fallo de página para la página 3 y P3 hace una nueva referencia a memoria, ahora a la página 4.
- Pero ahora la memoria principal ya no tiene marcos libres. Es el momento de elegir una página víctima para desalojarla de la memoria. Si la página menos recientemente usada es la página 2 de P1, es una buena candidata. En caso de que se encuentre modificada desde que fue cargada en memoria, se la copia en la zona de intercambio para no perder esas modificaciones, y se declara libre el marco M2 que ocupaba.
- Se marca como no válida la página que acaba de salir de la memoria principal. La próxima referencia a esta página que haga P1 provocará un nuevo fallo de página.
- Se copia la página que solicitó P3 en el nuevo marco libre, se la marca como válida en la tabla de páginas de P3, y el sistema continúa su operación normalmente.

Notemos que en este ejemplo existen tres procesos cuyos tamaños de espacio lógico miden **4, 5 y 6 páginas**, dando un total de **15 páginas**. Sin embargo, el sistema de cómputo sólo tiene **ocho marcos**.

Sin paginación por demanda y memoria virtual, solamente podría entrar en el sistema uno de los tres procesos. Durante las operaciones de entrada/salida de ese proceso, la CPU quedaría desaprovechada. Además, si alguno de los procesos tuviera un espacio lógico de más de ocho páginas, no podría ser ejecutado.

Con la técnica de memoria virtual, los tres procesos pueden estar activos simultáneamente en el sistema, aumentando la utilización de CPU. Y, si alguno de esos procesos tuviera un espacio lógico de **más de 8 páginas**, el sistema seguiría funcionando del mismo modo.

Capítulo 10

Redes de computadoras

Una **red de computadoras** es un conjunto de computadoras conectadas por medio de uno o varios enlaces, para compartir información u otros recursos.

10.1. Clasificación de las redes

Las redes se pueden clasificar por su tamaño:

- Una red cuyos límites (o **diámetro**) son pequeños, se llama una **red de área local o LAN (Local Area Network)**. Típicamente, una LAN está contenida en una oficina, piso o edificio.
- Una red que abarca el área de una ciudad (y por lo tanto, cuyos enlaces utilizan espacios públicos) suele llamarse **red metropolitana o MAN (Metropolitan Area Network)**.
- Una red mayor, que cubre distancias entre ciudades, países o continentes, se llama una **red de área extensa o WAN (Wide Area Network)**. Las redes de los **proveedores de servicios de Internet (ISPs)** suelen clasificarse como WANs.

10.2. Componentes de Hardware

En cualquier red se distinguen por lo menos tres elementos de hardware:

1. Hosts o nodos terminales.
2. Enlaces.
3. Nodos intermedios.
4. Interfaces.

10.2.1. Hosts

Las computadoras conectadas se llaman **hosts** o nodos terminales.

La figura 10.2 muestra una red simplificada donde podemos distinguir los elementos de hardware recién enumerados. Los *hosts* o nodos terminales serían la PC, la laptot y la impresora, todos conectados con un router (en este caso inalámbrico para proveer wifi). En la figura se muestra además cómo el router da acceso a la red de redes (internet)

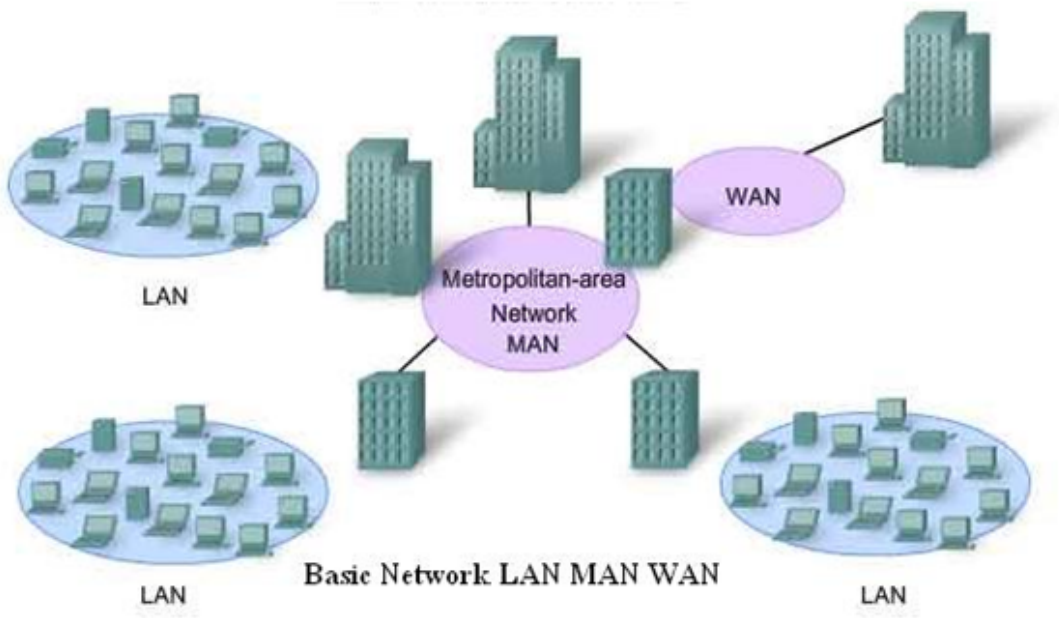


Figura 10.1: Una posible clasificación de las redes.

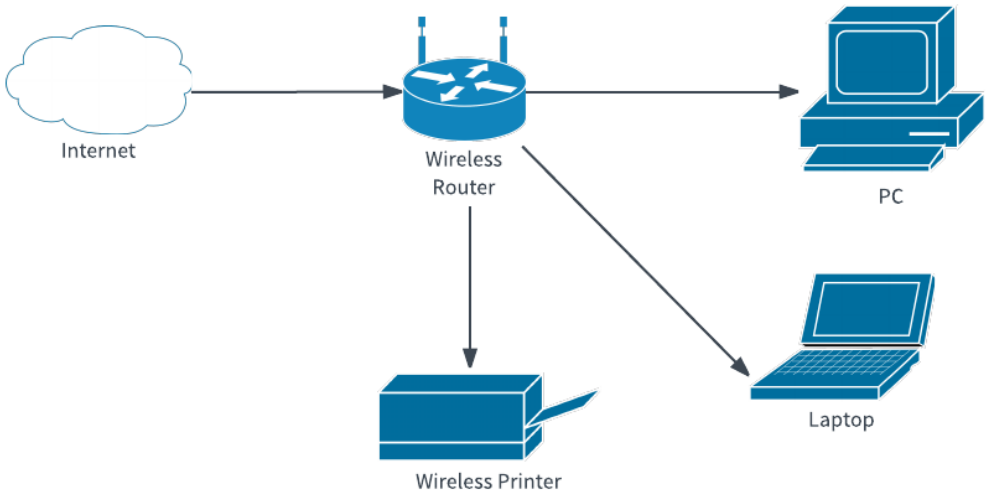


Figura 10.2: Principales componentes de hardware

Figura 10.3: Medios guiados.

10.2.2. Enlaces

Los **enlaces** que conectan las computadoras. Los enlaces suelen llamarse **punto a punto** cuando conectan únicamente dos nodos, o compartidos cuando al mismo enlace se conectan más de dos nodos.

Las señales (que representan los bits) son transmitidas sobre un enlace. El material de dicho enlaces se llama **medio** del enlace. Las tecnologías de construcción de los enlaces son muchas.

- Cuando las señales se codifican mediante impulsos eléctricos, como en las redes de cables de **par trenzado** o **coaxial**, el medio es un conductor, como el cobre.
- Para distancias mayores (como las transoceánicas) o para ambientes donde existe mucho **ruido** o interferencia electromagnética (como en fábricas), se utiliza **fibra óptica**.
- Cuando no es posible o práctico tender un cable, no queda más solución que utilizar emisiones de **radio**. Ejemplos de tecnologías de radio son los enlaces satelitales, los de microondas, y las LAN inalámbricas bajo norma **802.11** conocidas popularmente como **WiFi**. Estas tecnologías utilizan como medio el **espacio**.

Las principales compañías de conectividad del mundo tienden enlaces de fibra óptica transoceánicos. Como la instalación de estos cables es una maniobra muy compleja y tiene un costo altísimo, se aseguran de instalar capacidad de transmisión en abundancia. Por ejemplo, uno de estos enlaces tiene una capacidad de 3.2 Tbps, lo que permitiría transmitir el contenido completo de un disco rígido de 1 TB en menos de tres segundos. Esta capacidad es compartida entre varios proveedores de Internet que compran el servicio de transporte.

Interesante

Submarine Cable Map

En el pasado también se usaron los enlaces satelitales para resolver el problema de cubrir grandes distancias. Los satélites son repetidores de radio colocados en órbita. Reciben emisiones de una estación terrena y la comunican a otra distante, superando el problema de la curvatura terrestre, que no permitiría la propagación en línea recta de la emisión de radio.

Su principal inconveniente es la alta **latencia o retardo** en la llegada de la señal desde un punto a otro, debido a las grandes distancias que se deben enlazar. Los satélites **geoestacionarios** o de órbita alta (**GEO**) se instalan a una altura de alrededor de 35700 km. Al ubicarlos a esta altura se alcanza un equilibrio entre la fuerza de gravedad terrestre y la fuerza centrífuga del satélite, lo que garantiza que permanecerán inmóviles respecto de algún punto de la superficie terrestre, y así cubrirán siempre la misma región del planeta. Pero la órbita alta implica una

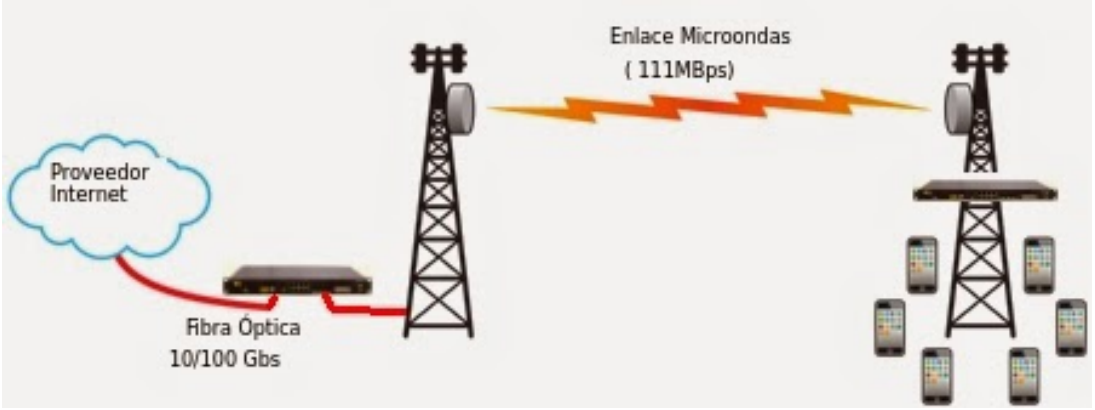


Figura 10.4: Ejemplo de medio no guiado: Enlace microonda.

gran distancia a recorrer para las señales, lo que introduce demoras de alrededor de un cuarto de segundo entre estaciones terrestres. Estas demoras son tolerables para algunas aplicaciones de tráfico de datos, pero perjudiciales para las comunicaciones interactivas.

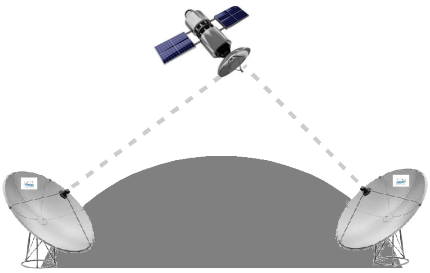


Figura 10.5: Ejemplo de medio no guiado: Enlace satelital.

Paulatinamente van siendo abandonados en favor de la fibra óptica para comunicación de datos a grandes distancias. Hoy se estima que sólo un 5 % del tráfico internacional es satelital, y el resto es conducido por fibras ópticas. Sin embargo, siguen siendo una buena solución para atravesar áreas continentales, o para distribuir tráfico hacia muchos puntos simultáneos de bajada, como en los medios de comunicación televisivos (aplicación llamada **broadcasting**).

10.2.3. Nodos intermedios

Los **nodos intermedios** sirven para compartir un enlace (switches) o encaminar el tráfico de información entre los nodos terminales (routers).

Switches

En las redes de área local, o LAN, encontramos enlaces compartidos. El cableado de una oficina, un aula o un edificio es un único medio de comunicación compartido por todos los nodos de la red. El cableado se concentra en un punto de conmutación llamado **switch** o, justamente, conmutador, que distribuye el tráfico entre los nodos conectados. Un switch tiene muchas **interfaces** donde se conectan cables punto a punto hacia los nodos de la LAN.

Routers

Suele definirse a Internet como **red de redes**. Las grandes redes, y en particular Internet, se componen interconectando redes a través de enlaces, a veces de gran longitud. Entre cada dos de estas redes siempre existe un **router**.

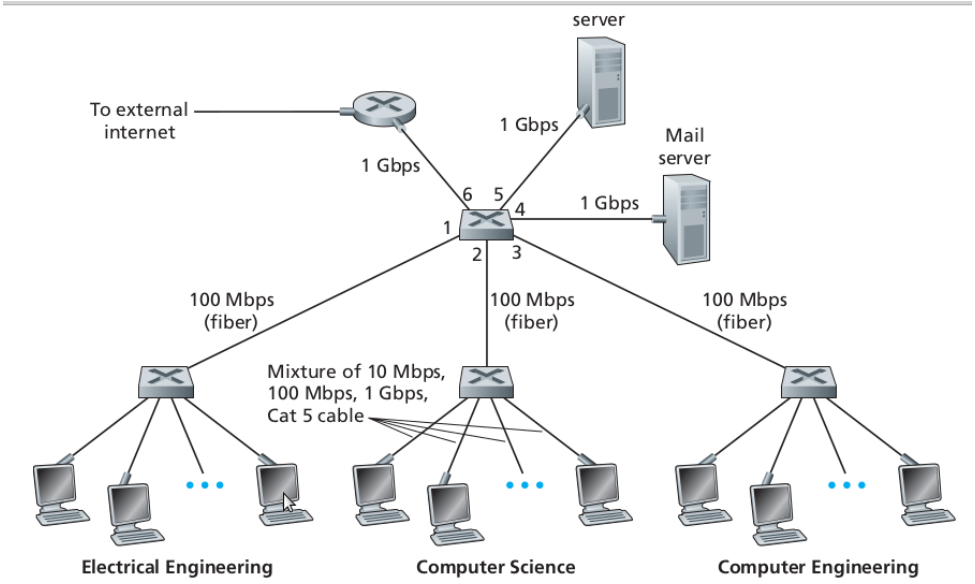


Figura 10.6: Red institucional coctada con 4 switches.

El router presta el servicio que no alcanza a prestar el nivel de enlace, que es el de enviar el tráfico fuera de la red de origen. El nivel al que pertenecen los routers se llama **nivel de red**.

Figura 10.7: Router de alto rendimiento.

Los routers son los elementos que toman las decisiones de **enrutamiento** o ruteo, al determinar por cuál de sus interfaces, que a veces son muchas, debe ser enviada la información que reciben. Esta tarea de enrutar la información se cumple mediante software de enrutamiento.

El hardware y el sistema operativo de los routers pueden estar altamente especializados en la tarea de ruteo, pero también es perfectamente posible construir un router a partir de una computadora corriente de escritorio y un sistema operativo multipropósito. Es decir que los routers son computadoras, con un sistema operativo y un hardware similares a los que encontramos en muchas otras computadoras, pero dedicadas a la tarea del enrutamiento.

Figura 10.8: Router hogareño.

Dependiendo del ambiente donde deben trabajar y de la cantidad de tráfico que deben procesar, los routers pueden adoptar muchas formas físicas y tamaños.

- Los routers pueden ser pequeños y baratos, para uso doméstico.
- Algunos pueden dar servicio a muchos nodos terminales al incluir múltiples dispositivos de nivel de enlace, como un switch 802.3 de varias interfaces, y un **punto de acceso** para una **red**

inalámbrica secundaria basada en tecnología de radio.

- Algunos, de muy altas prestaciones, usados por los proveedores de servicios de Internet, son **modulares** y pueden ser configurados a medida de las necesidades. Cada módulo contiene **interfaces** especializadas en alguna tecnología de enlace, lo que les permite conectar redes de tecnologías completamente diferentes.

10.2.4. Interfaces

La interfaz es el punto de conexión entre un enlace y un nodo de la red. Es la pieza de hardware que convierte bits a señales capaces de viajar por la red, y viceversa. Cuando un nodo debe comunicar algo a otro, prepara su mensaje en una zona de la memoria, y entrega esos contenidos binarios a la interfaz a través de un bus de comunicación.

La interfaz contiene el hardware necesario para traducir ese **tren de bits** a señales eléctricas (cuando los enlaces son cableados), de radio (cuando el enlace es inalámbrico), o luminosas (cuando el enlace es de fibra óptica).

Las modernas interfaces de red pueden funcionar a **velocidades de transmisión** de muchos bits por segundo. Una LAN cableada actual funciona comúnmente en velocidades de 1 a 10 Gb/s o Gbps (gigabits por segundo). Una LAN inalámbrica suele funcionar a una velocidad de transmisión mucho menor (y, además, variable, dependiendo de condiciones físicas ambientales que tienden a limitar la velocidad de transmisión).

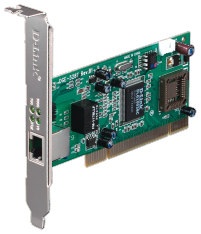


Figura 10.9: Interface.

El tren de bits viaja en forma de señales físicas por el enlace hasta llegar a la interfaz del nodo destino dentro de la red de área local. La interfaz receptora decodifica las señales, recuperando los bits originales y comunicándolos al software que espera los datos. Ambas partes de la aplicación distribuida se han comunicado un mensaje.

10.3. Componentes de Software

Para utilizar la red, todos los nodos que están conectados a ella, ya sean terminales o intermedios, corren software de red como **protocolos** y **aplicaciones de red**. Los protocolos son, por un lado, convenciones que establecen con todo detalle cómo se realiza una comunicación, y por otro lado, los componentes de software que implementan esa forma de comunicación.

10.3.1. Aplicaciones de Red

Las aplicaciones de red son aplicaciones **distribuidas**, es decir, se componen de al menos dos partes, preparadas para comunicarse una con otra, y esas partes funcionan en nodos terminales de la red diferentes. Cada aplicación de red utiliza un protocolo, porque la interacción entre

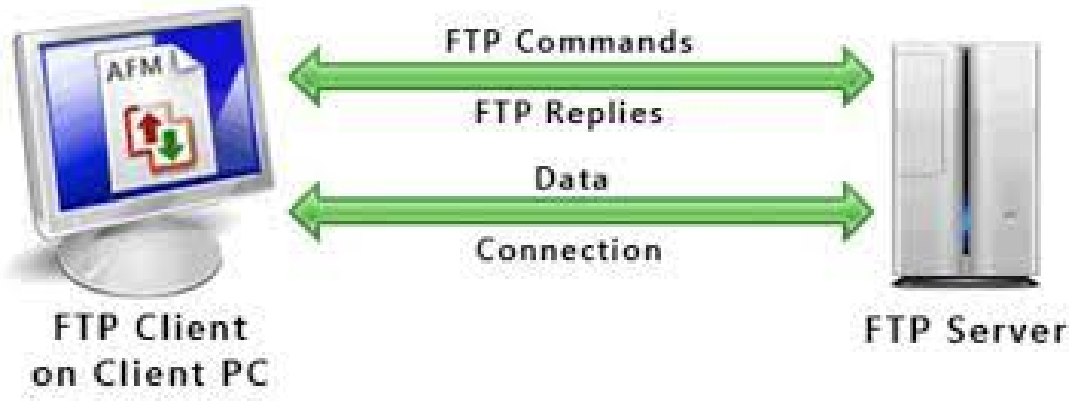


Figura 10.10: Ejemplo de una aplicación de red (usando protocolo de aplicación ftp).

las partes debe estar perfectamente determinada para que los nodos se entiendan sin errores ni ambigüedades. Un protocolo puede ser utilizado por varias aplicaciones.

10.3.2. Protocolos

Los **protocolos** son conjuntos de reglas que definen la interacción entre dos entidades de la red. Para comunicarse, las entidades de cualquier nivel deben compartir un protocolo. Los protocolos especifican:

- Cuál es el **formato** de los mensajes que pueden intercambiar las entidades;
- Qué tipo de **acciones** o respuestas debe dar cada entidad al recibir cada mensaje.

Puede ser útil comparar los protocolos de red con protocolos sencillos de la vida cotidiana. Muchas interacciones entre las personas están gobernadas por protocolos, a veces poco evidentes. Por ejemplo, comprar un artículo cualquiera en un comercio sigue unas pautas bastante definidas.

Aunque los contenidos específicos de los mensajes pueden variar, es habitual que existan fases en la interacción entre las personas, como el **inicio de sesión, la autentificación, la autorización, las peticiones o requerimientos (*requests*), las respuestas (*responses*), y el cierre de sesión**. Todas éstas son fases habituales en la comunicación entre los humanos, pero también en los protocolos de las redes.

10.3.3. Modelo de Internet de 5 capas

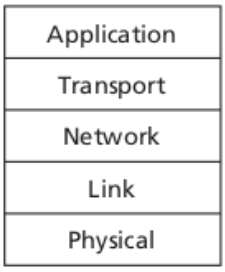


Figura 10.11: Modelo de 5 capas de Internet.

Hoy, la mayoría de las redes están interconectadas por una única red global llamada **Internet**. Para conectarse a Internet, todos los nodos, terminales e intermedios, ejecutan un protocolo básico llamado **IP (Internet Protocol)**, y se puede decir que, desde el punto de vista del software, constituyen una sola red. Sin embargo, desde el punto de vista de la administración, dentro de Internet existen numerosas redes. Cada red es propiedad de una persona u organización que tiene el control sobre los nodos y enlaces de esa red, decide qué nodos pertenecen o no pertenecen a ella, y qué protocolos y aplicaciones utilizan.

Las redes pueden estudiarse y comprenderse mediante modelos jerárquicos compuestos por capas, donde cada pieza de hardware o de software pertenece a una capa o nivel. Cada capa corresponde a un conjunto de problemas relacionados, y a las soluciones posibles. Para funcionar, cada capa se apoya en las soluciones provistas por la capa inmediatamente inferior.

- Aplicación: En la capa de Aplicación se encuentran los protocolos sobre los cuales se basan directamente las aplicaciones distribuidas.
- Transporte: La capa de Transporte soluciona el problema de la entrega de datos entre **procesos** de nodos diferentes.
- Red: La capa de Red soluciona el problema de la entrega de datos entre **nodos** de diferentes redes.
- Enlace: La capa de Enlace soluciona el problema de la entrega de datos entre **nodos de la misma red**.
- Física: La capa Física define la forma como se codifican y transmiten las señales que representan la información.

10.4. Direccionamiento en Internet

10.4.1. Direcciones de red

Para poder dirigir los mensajes entre nodos, es necesario identificarlos de alguna forma, asignándoles **direcciones** o identificadores de red. En Internet, las direcciones son asignadas a las interfaces, y no a los nodos. De esta manera, si un nodo tiene más de una interfaz, recibirá más de una dirección. El caso típico de un nodo con más de una interfaz son los routers, que tienen una interfaz perteneciente a cada una de las redes que conectan.

El protocolo IPv4 define las direcciones de red como números de 32 bits que se asignan a cada interfaz de los nodos. Estas direcciones de 32 bits suelen escribirse como cuatro valores decimales, entre 0 y 255, separados por puntos.

Ejemplo

- La dirección IPv4 **11000000101010000000000100000001** se puede escribir en notación decimal con punto como **192.168.1.1**.

10.4.2. Paquetes IP

Internet es una red del tipo de **conmutación por paquetes**, lo que significa que los flujos de datos que van de un nodo emisor a un receptor son fraccionados en **paquetes** o trozos de datos, de un cierto tamaño máximo, y que los nodos intermedios tratan a cada paquete individualmente para encaminarlos a su destino.

En una red conmutada por paquetes, los nodos intermedios, o routers, no necesitan conocer todo el camino que debe atravesar cada uno de los paquetes. En cambio, un router sólo necesita saber a cuál de los **nodos intermedios adyacentes** encaminarlo, basándose en información transportada por el mismo paquete.

Cada nodo intermedio o router en el camino entre el emisor y el receptor tomará una nueva decisión de ruteo ante cada uno de los paquetes que llegan a él.

Al generar un paquete, para que pueda ser encaminado, el emisor completa los datos con un **encabezado** conteniendo la dirección IP del nodo emisor, o **dirección origen**, y la dirección IP del nodo destino, o **dirección destino**.

10.4.3. Ruteo o encaminamiento

Cuando un paquete llega a un router, lo hace por algún enlace. La tarea del router es **reenviar** este paquete por otro de sus enlaces, de modo que se aproxime a su destino.

El router debe aplicar alguna regla lógica para decidir hacia qué otro enlace **reenviar** el paquete. Esta decisión de cuál será ese otro enlace es una acción de **ruteo** o **encaminamiento**.

Tabla de reenvío o de ruteo

La decisión de ruteo es tomada por los routers usando la información de **destino** que llevan consigo los paquetes, más información de ruteo contenida en una **tabla de reenvío** o tabla de ruteo, almacenada en la memoria del router. La tabla de ruteo contiene reglas para la decisión de encaminamiento de los paquetes. Cada regla se llama una **ruta** e indica cuál será la interfaz de salida de los paquetes cuya dirección destino coincida con la dirección destino de la ruta. En líneas generales, el algoritmo de ruteo es como sigue (una versión muy simplificada):

- Al recibir un paquete, el router examinará la dirección destino **del paquete** y la comparará con la dirección destino **de cada ruta**.
- Al encontrar una coincidencia de dirección destino entre el paquete y la ruta, utilizará la información en la columna de **interfaz de salida** de esa ruta, reenviando el paquete por esa interfaz.

10.5. Servicio de Nombres de Dominio (DNS)

Como hemos visto, el funcionamiento de Internet se basa en la existencia de **direcciones**, que permiten enviar y encaminar el tráfico entre los diferentes puntos de la red. Sin embargo, las direcciones IP de 32 bits, o su equivalente de cuatro decimales con puntos, son incómodas de manejar para los humanos. El **servicio de nombres de dominio**, o Domain Name Service (**DNS**) es un servicio agregado a la Internet para comodidad de los usuarios.

El servicio DNS permite a los usuarios referirse a los nodos de Internet mediante nombres simbólicos en lugar de direcciones, y entra en acción cada vez que se menciona el **nombre** de un nodo para hacer contacto con él. Un nodo que necesita enviar un mensaje cualquiera a otro nodo necesita conocer su dirección IP. Si únicamente conoce su nombre, pedirá una **traducción de nombre** a algún **servidor DNS**.

Técnicamente, Internet podría funcionar perfectamente (y, de hecho, lo hizo durante algún tiempo) **sin** la existencia del servicio DNS, pero la costumbre lo ha convertido en una parte indispensable de la red.

10.5.1. Jerarquía de nombres de dominio

Estos nombres simbólicos tienen una cierta estructura jerárquica, es decir, organizada por niveles. Un nombre consta de varias partes, separadas por puntos. En cada nombre, las partes más a la derecha designan conjuntos mayores de nodos, y las partes más a la izquierda, conjuntos más pequeños, contenidos en aquellos conjuntos mayores.

Ejemplo

- El nombre **pedco.uncoma.edu.ar** designa al nodo **pedco**, que pertenece al conjunto **uncoma**, contenido en el conjunto **edu** contenido en el conjunto **ar**.
- El nombre **pedco.uncoma.edu.ar** equivale, gracias al servicio DNS, a la **dirección IP 170.210.81.41**.

Estos conjuntos de nombres se llaman **dominios**. Los dominios que aparecen más a la derecha son los más generales. Inicialmente fueron siete de propósito general (**org, mil, gov, edu, com, net, int**), más los pertenecientes a los países (**ar** para Argentina, **cl** para Chile, **uy** para Uruguay...), pero luego se agregaron otros. Por ser los más generales, están al tope de la jerarquía, y se llaman dominios de nivel superior o **TLD (Top-Level Domains)**.

Los dominios de nivel superior contienen otros espacios de nombres, llamados a su vez dominios, y éstos, otros llamados subdominios. En el ejemplo anterior, el TLD es **ar**, el dominio es **edu.ar** (educación, Argentina), el subdominio es **uncoma.edu.ar** (Universidad del Comahue, educación, Argentina) y finalmente **pedco.uncoma.edu.ar** es el nombre de un nodo perteneciente a la Universidad del Comahue, educación, Argentina.

10.5.2. Resolución de nombres

Los servidores DNS se clasifican por el tipo de función que cumplen. Cada uno interviene de una manera especial en el mecanismo de traducción de nombres a direcciones. Este mecanismo de traducción se llama **resolución** de nombres.

- Cada nodo de Internet lleva una configuración que le dice cuál es la dirección de su **servidor DNS local**. El servidor local es quien responde efectivamente una consulta DNS. Normalmente, el servidor local se encuentra "cerca" del cliente en términos de redes. Posiblemente, en la misma red local, o en la del proveedor de acceso a Internet.

- Al ser consultado por un nombre, un servidor local usará la estrategia de analizar ese nombre **de derecha a izquierda**. Utilizará los componentes del nombre en ese orden, es decir, yendo de lo general a lo particular. En primer lugar utilizará el TLD o nombre de nivel superior para averiguar información sobre ese conjunto de nombres.

  Los servidores locales no conocen todas las posibles traducciones de nombre a dirección IP, por lo cual necesitan el servicio de los **servidores raíz**. Estos son alrededor de una veintena de servidores distribuidos en diferentes lugares del planeta, y todos tienen la misma información, replicada: las direcciones de los servidores de los TLD.

  Así, un servidor local obtendrá, de un servidor raíz, el dato de dónde ubicar al servidor del TLD **ar**.

- Conociendo la dirección IP del servidor del TLD, el servidor local lo interrogará entonces acerca del siguiente componente del nombre.

  Los servidores de los TLD tampoco conocen todas las posibles traducciones, sino que conocen las direcciones de los servidores DNS de los dominios por debajo de ellos. Así, el servidor del TLD **ar** puede decirle al servidor local dónde ubicar al servidor del dominio **edu.ar**.

  Otra información que este servidor del dominio TLD **ar** podría darle al servidor local, si la pidiera, serían las direcciones de los servidores de los dominios **com.ar**, **org.ar**, etc.

- Ahora el servidor local puede consultar al servidor del dominio **edu.ar**, que es quien puede informarle la dirección del servidor del subdominio **uncoma.edu.ar**.

- Finalmente, el servidor local consulta al servidor del subdominio **uncoma.edu.ar** por la traducción del nombre de nodo **www.uncoma.edu.ar**. Este servidor tiene en sus tablas la información que dice cuál es la dirección IP asignada a ese nombre. Ahora el servidor local puede devolver esa información al cliente que originalmente hizo la consulta.

Todo este complejo mecanismo debería tener lugar cada vez que un cliente de la red consulta por un nombre. Sin embargo, como este mecanismo es costoso en tiempo y en ancho de banda de las redes, se adopta un esquema de **cache** o reserva de información. Como es muy probable que esa información vuelva a ser solicitada, los pares (nombre, dirección) que han sido resueltos quedan guardados en una memoria temporaria o **cache** del servidor local. De esta manera las próximas consultas podrán responderse sin necesidad de volver a generar tráfico hacia el resto de la Internet.

10.6. Administración de redes

Existen herramientas de software que permiten diagnosticar las condiciones en que se realiza el ruteo, o encaminamiento, de los paquetes IP. El administrador de redes las utiliza para investigar el origen de los problemas en la red.

10.6.1. Comando ping

El comando **ping** emite paquetes hacia un nodo destino. Si los paquetes logran atravesar la Internet, el nodo destino emitirá una respuesta. El comando ping muestra los paquetes de respuesta que llegan o se pierden, y el tiempo que demora cada respuesta en llegar.

Cuando los usuarios tienen problemas con alguna aplicación de red, el comando ping es útil como herramienta de diagnóstico porque permite saber si la red es capaz de hacer llegar paquetes de un nodo a otro. Si el diagnóstico de ping es positivo, el administrador de red no se preocupa en comprobar cuestiones asociadas con los niveles inferiores a la capa de red: la comunicación a nivel físico, de enlace y de red entre ambos nodos es operativa. Si existe alguna condición de error, se deberá a problemas relacionados con las aplicaciones, que habrá que investigar.

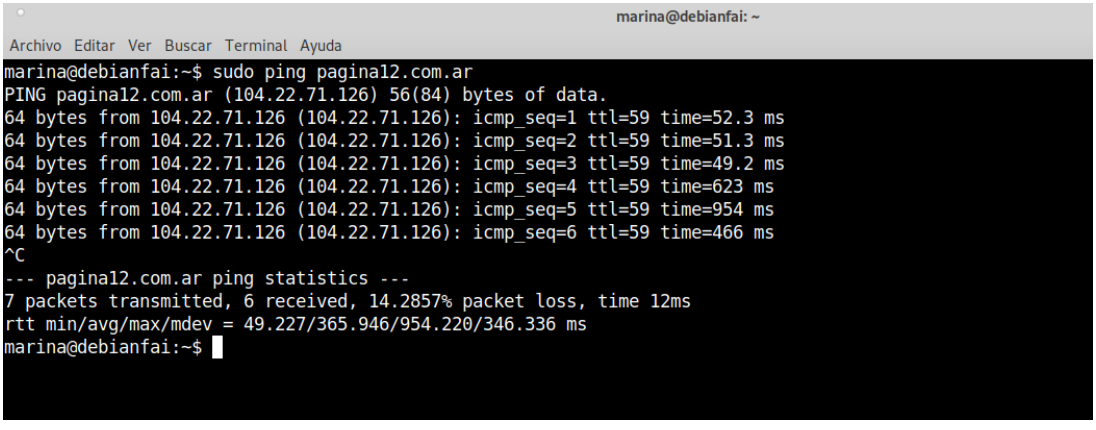


Figura 10.12: Ejemplo de uso de ping.

10.6.2. Comando traceroute

El comando **traceroute** permite investigar cuál es la cadena particular de routers que debe atravesar un paquete para llegar a un destino dado. Además, da información sobre la demora

en atravesar cada enlace, lo que puede dar una idea de si existe una condición de **congestión** en el camino de los paquetes, y en qué lugar de Internet.

Una condición de congestión es aquella que aparece cuando los nodos intentan utilizar un enlace más allá de su capacidad. Si un enlace es capaz de transmitir una cantidad de bits por segundo, y las demandas de los nodos de la red superan esa capacidad, el router comienza a acumular o encolar paquetes hasta que no tiene más espacio en su memoria para almacenarlos. A partir de este momento, si llegan nuevos paquetes, simplemente los descarta. El programa traceroute hace evidentes las pérdidas de paquetes, y dice en cuál de los enlaces ocurren.

El programa traceroute también permite detectar anomalías de ruteo como los lazos o ciclos de ruteo, que se producen cuando los paquetes toman caminos circulares de los cuales no pueden salir.

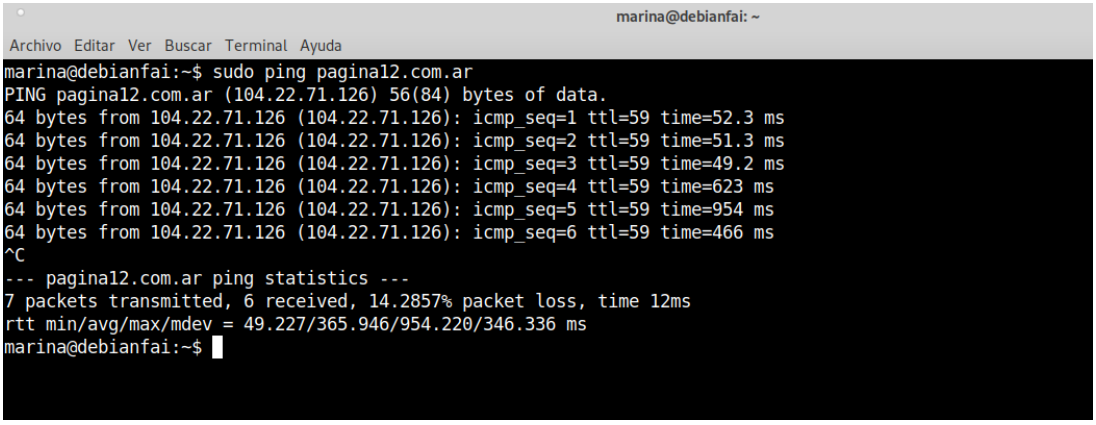


Figura 10.13: Ejemplo de uso del comando traceroute.