Universidad Nacional del Comahue

Facultad de Informática

Introducción a la computación

Apunte de la materia

Versión 3.1 2026

Basado en la versión original de Eduardo Grosclaude, el *Oso*

Índice general

1. Historia de la Computación **1**

1.1. Antecedentes 1

1.2. Primera Generación (1938-1950) 2

1.3. Segunda Generación (1951-1964) 5

1.4. Tercera Generación (1965-1971) 5

1.5. Cuarta Generación (1971-actualidad) 7

2. Sistemas de Numeración **9**

2.1. Introducción 9

2.2. Sistema posicional y base de un sistema de numeración 10

2.2.1. Expresión General 11

2.3. Conversión de base 11

2.3.1. Conversión de otras bases a base 10 11

2.3.2. Conversión de base 10 a otras bases 12

2.3.3. Conversión entre bases arbitrarias 13

2.3.4. Conversión entre sistemas binario y octal o hexadecimal 14

3. Unidades de Información **17**

3.1. ¿Qué es la Información? 17

3.2. El Bit 17

3.2.1. El viaje de un bit 18

3.3. El Byte 18

3.4. Sistemas de medición 19

3.4.1. Sistema Internacional 19

3.4.2. Sistema de Prefijos Binarios 19

3.4.3. ¿Por qué dos sistemas? 19

4. Representación de datos numéricos **21**

4.1. Rango de Representación 21

4.1.1. Valores representables con k bits: Cuántos y cuáles 21

4.2. Sistema de Representación Sin Signo: SS(k) 22

4.2.1. Rango de representación de SS(k) 22

4.3. Sistema de Representación con Signo 23
4.4. Sistema de Representación Signo-magnitud: SM(k) 23
4.4.1. Rango de Representación de SM(k) 23
4.4.2. Limitaciones de Signo-Magnitud 24
4.5. Sistema de Representación y Operación Complemento a 2 24
4.5.1. Rango de Representación de C2(k) 24
4.5.2. Aritmética en C2 24
4.5.3. Overflow o desbordamiento en C2 25
4.5.4. Extensión de signo en C2 26
4.6. Notación en exceso 27
4.6.1. Conversión entre exceso y decimal 27
4.7. Números fraccionarios y decimales 28
4.7.1. Conversión de binario a decimal 28
4.7.2. Conversión de decimal a binario 29
4.8. Representación de números fraccionarios 30
4.8.1. Representación de punto fijo 30
4.8.2. Notación Científica 32
4.8.3. Representación en Punto Flotante 34

Índice de figuras

1.1. Charles Babbage (1792–1871) 1
1.3. Tarjeta Perforada 2
1.2. Ada Lovelace (1815–1852) 2
1.5. Colossus (1943,1944) 3
1.4. Tubo de Vacío 3
1.6. Colossus (1943,1944) 4
1.7. ENIAC (1945) 4
1.8. ENIAC (1945) 5
1.9. Transistor (1947) 5
1.10. PDP-1 (1959) 6
1.11. Intel 4004 (1971) 7
1.12. IBM PC 5150 (1981) 7

2.1. Sistema de numeración egipcio. 9
2.2. Diferencia entre numeral y número 10
2.3. Notación del Sistema Posicional (ejemplo con base 10) 10
2.4. Ejemplo de conversión de cualquier base a base 10 12
2.5. Ejemplo de conversión de base 10 a otra base. 12
2.6. Conversión de binario a octal agrupando de a 3 dígitos binarios 14

3.1. Un byte son 8 bits 18

4.1. Suma aritmética de dos operandos (representados en C2) 25
4.2. Acarreos o carris de una suma aritmética. 25
4.3. Comparación de los últimos acarreos para detectar overflow. 26

Capítulo 1

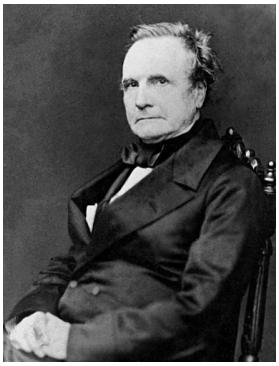
Historia de la Computación

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.

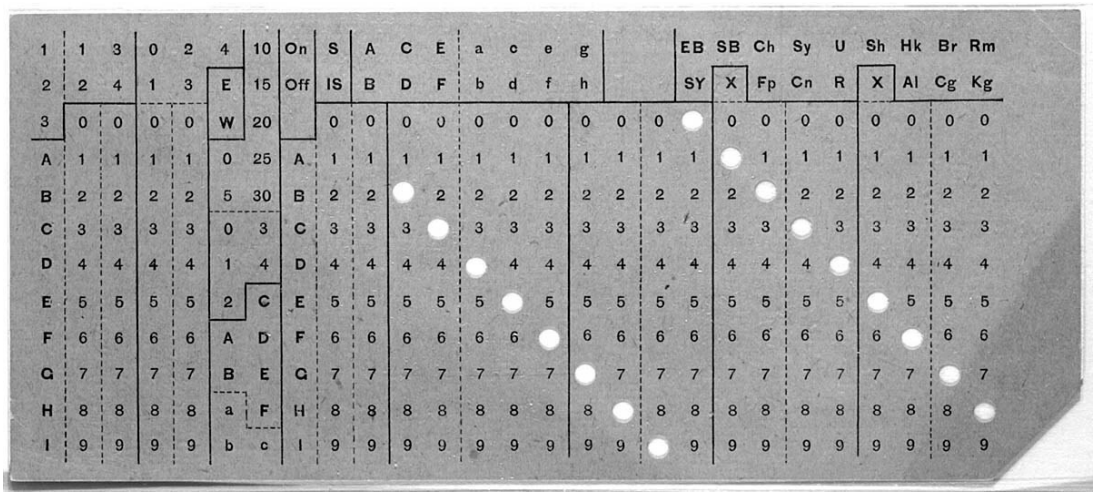
Por lo que respecta a la capacidad de seguir un algoritmo, podemos ver una cierta progresión en la flexibilidad de estas máquinas. La máquina de Pascal se construyó para realizar únicamente sumas. En consecuencia, la secuencia apropiada de pasos estaba integrada dentro de la propia estructura de la máquina. 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.

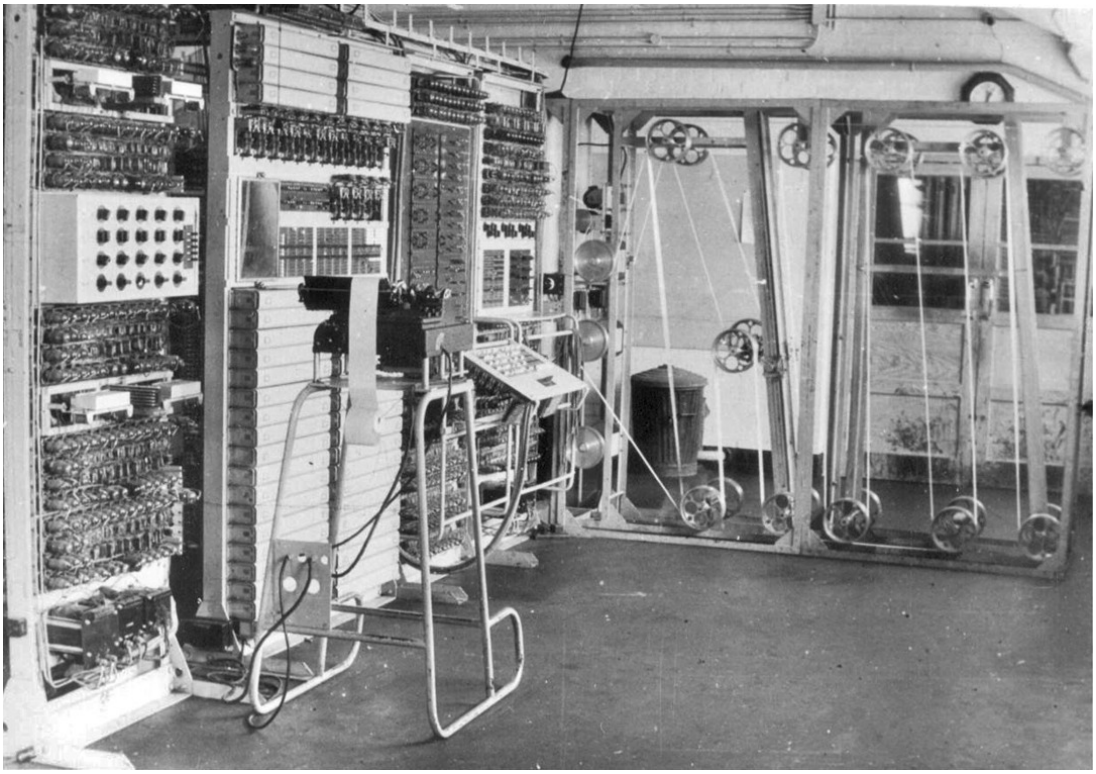
1.2. Primera Generación (1938-1950)

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.

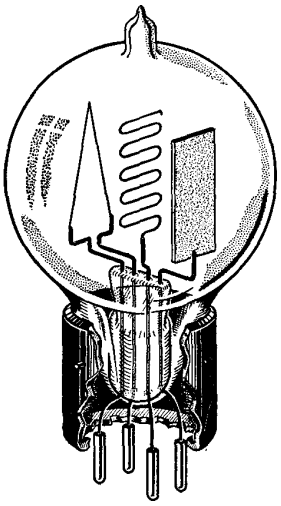
De esta manera se puede controlar el flujo de corriente por un circuito y construir dispositivos que implementen funciones lógicas. Así, dos válvulas de este tipo, conectadas en serie, simulan una función lógica de conjunción o **AND**; dos válvulas conectadas en paralelo, simulan una disyunción u **OR**, 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

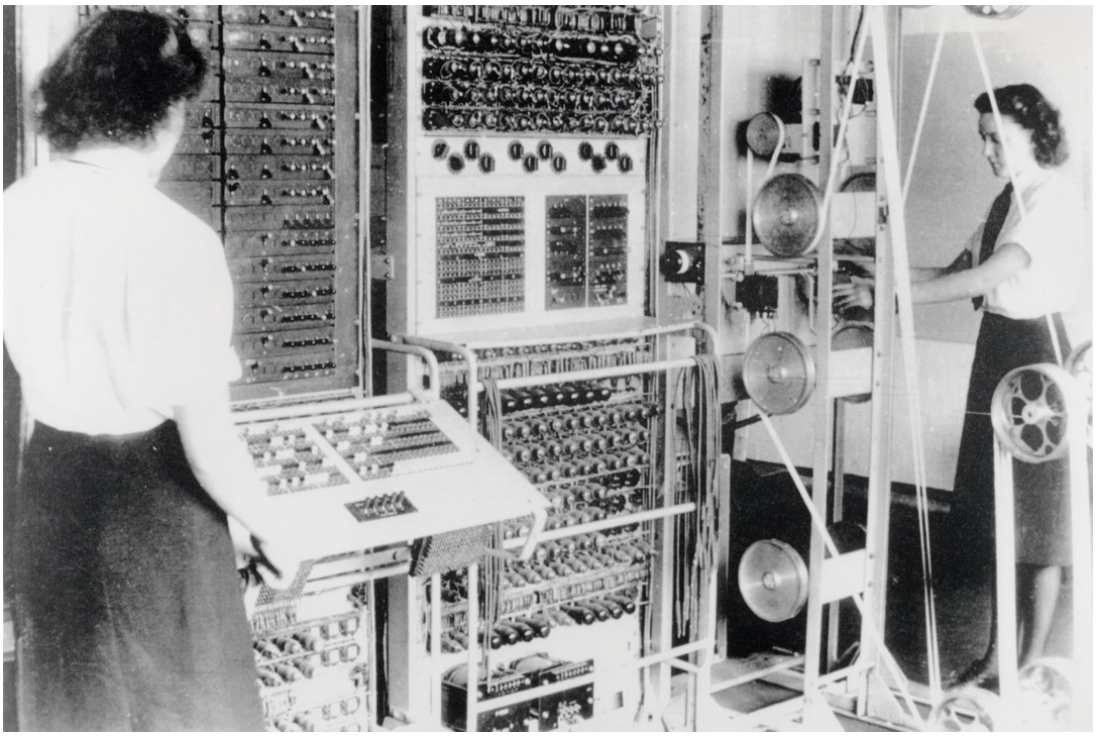
Colossus (1943,1944) fue construida en Inglaterra para decodificar los mensajes alemanes durante los últimos años de la Segunda Guerra Mundial, y se programaba activando y desactivando interruptores físicos. Cada computadora Colossus utilizaba entre 1.600 y 2.400 tubos de vacío. La existencia de la máquina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

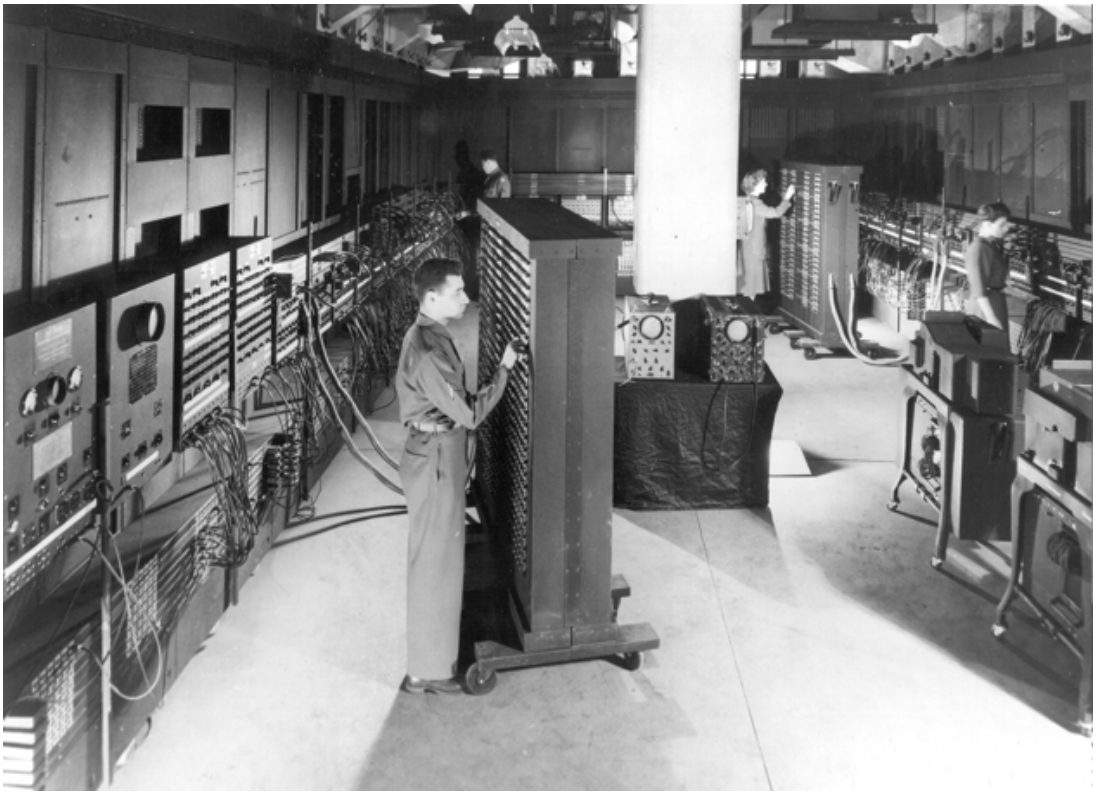
La tecnología de tubo de vacío requería una gran cantidad de electricidad. La computadora ENIAC (1946) tenía más de 17,000 tubos y sufría fallas de tubo (que tardaría aproximadamente 15 minutos en ubicarse) en promedio cada dos días. Debido a que la falla de cualquiera de los miles de tubos en una computadora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

Clementina

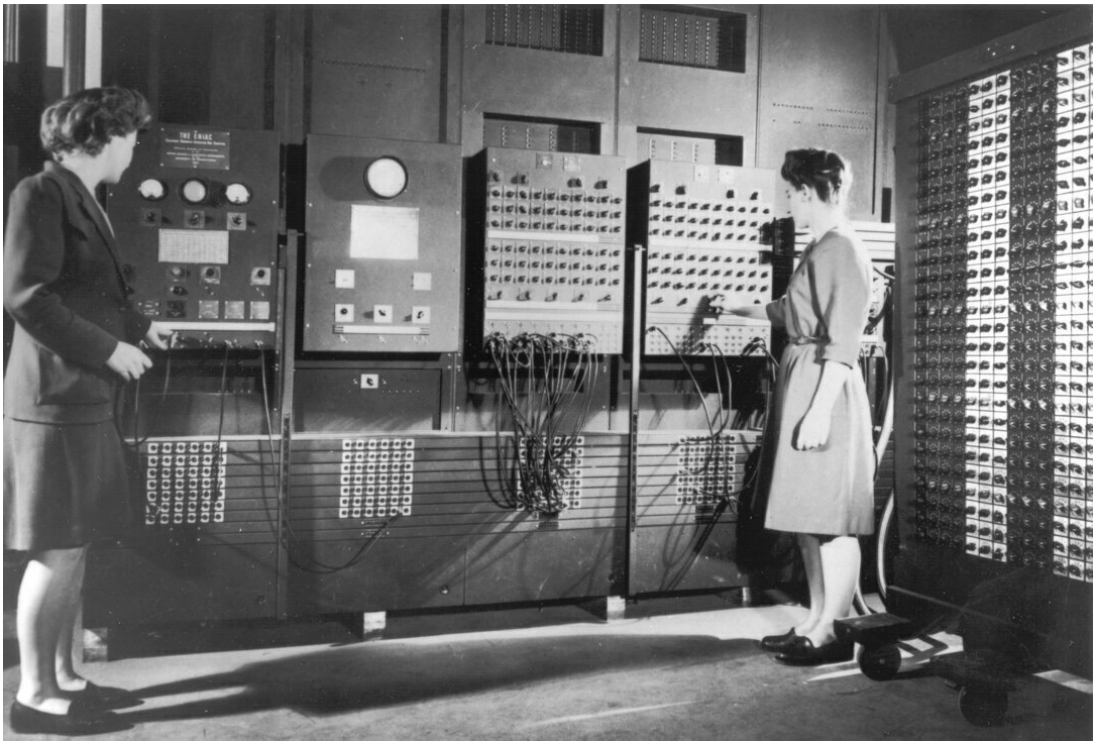
¿Qué pasaba en nuestro país durante estas épocas? La actividad de la computación aquí no había comenzado. Recién a principios de los años 60 la universidad argentina decidió hacer una importante inversión, que fue la compra de una computadora de primera generación, bautizada aquí **Clementina**.



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:

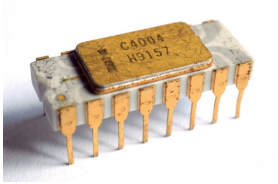
- Menor consumo de energía eléctrica.
- Apreciable reducción del espacio que ocupaba el aparato.
- Aumento de fiabilidad y flexibilidad.

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.

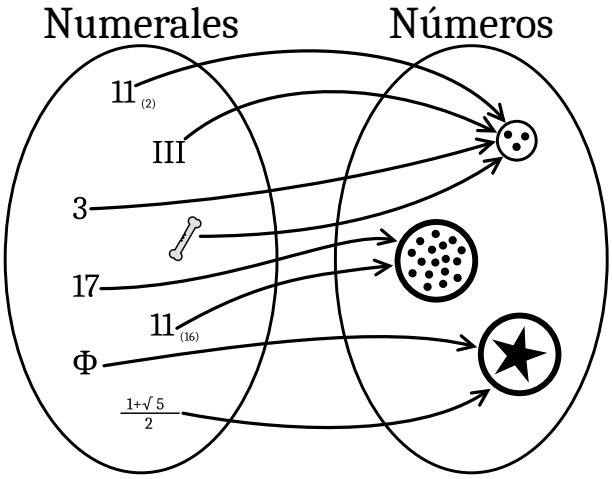


Figura 2.2: Diferencia entre numeral y número

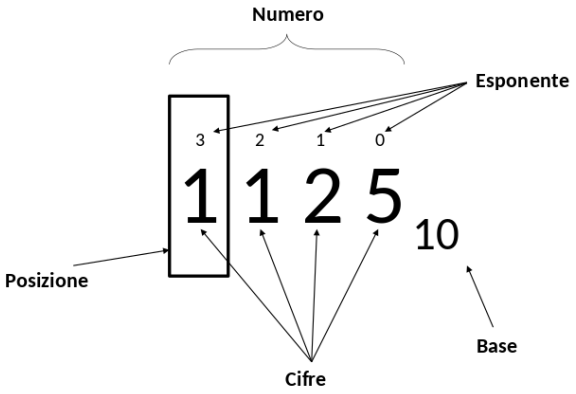


Figura 2.3: Notación del Sistema Posicional (ejemplo con base 10)

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 casillero. 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 3.1). 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.

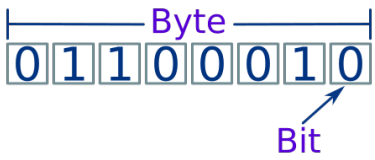


Figura 3.1: Un byte son 8 bits

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: pueden ser números, caracteres como letras u otros símbolos, o pueden formar parte de otra estructura de información más compleja.

3.4. Sistemas de medición

Para indicar diferentes cantidades de bytes (por ejemplo para indicar el tamaño de un archivo) utilizamos la unidad más apropiada para esa cantidad. 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. Utilizaremos unidades de dos sistemas diferentes: el **Sistema Internacional** y el **Sistema de Prefijos Binarios**. Las unidades de ambos sistemas son parecidas, pero no iguales.

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 × 1024 bytes
- 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 celda de la memoria tiene una dirección que la identifica.
- 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.

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

- 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 otro lado, los fabricantes de medios de almacenamiento, como memorias, discos rígidos o pendrives, si bien deberían utilizar el sistema de Prefijos Binarios para expresar las capacidades de almacenamiento de esos medios, en general utilizan el sistema Internacional. Así, un *"pendrive de dieciséis gigabytes"*, en realidad tiene una capacidad de 16×2^{30} bytes, y debería publicitarse 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

4.5.1. 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.

4.5.2. 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.

4.5.3. 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.

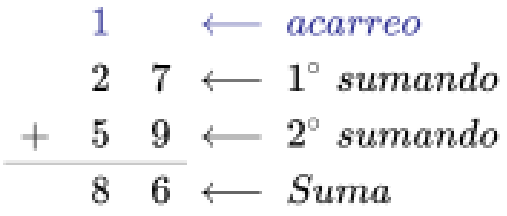


Figura 4.2: Acarreos o carris de una suma aritmética.

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

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

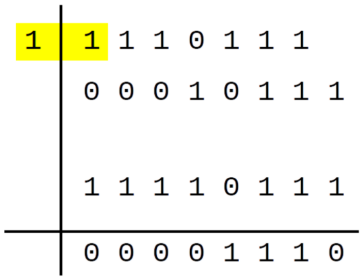


Figura 4.3: Comparación de los últimos acarreos para detectar overflow.

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

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?

4.5.4. 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:

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

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, al proveer un punto que puede *flotar* o moverse, de manera de tener números muy grandes y con pocos decimales, o muy pequeños con muchos decimales. Inspirándose en la notación científica, los formatos de punto flotante permiten escribir números de un gran rango de magnitudes y precisiones.

En esta materia trabajaremos con el formato denominado *Minifloat*, de 8 bits. Sin embargo, en la actualidad se utilizan los estándares de cómputo en punto flotante definidos por la organización de estándares **IEEE** (Instituto de Ingeniería Eléctrica y Electrónica), llamados **IEEE 754 en precisión simple (de 32 bits) y en precisión doble (de 64 bits)**.

- Minifloat de 8 bits:
  - 1 bit de signo.
  - 4 bits para el exponente.
  - 3 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 7.
   - Al exponente se le suma 7 para representar valores en el intervalo $[-7, 8]$ con 3 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 3 bits disponibles (o completando con ceros).

Error por truncamiento

Como veremos existen números que no pueden ser representados. 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** a los 3 bits de la mantisa. 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.

El **cero** se representa con cero en el exponente y en la mantisa, dando lugar a cero positivo y cero negativo.

El infinito se representa con el máximo exponente (1111) y la mantisa puesta en cero. El bit del signo puede ser positivo o negativo. Los valores **infinito** positivo y negativo aparecen cuando una operación en este formato resulta en **overflow**.

Similarmente, cuando el exponente vale cuatro 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).