Arquitectura Q

Funcionamiento del computador desde la perspectiva de una arquitectura conceptual

Material para la asignatura

Organización de Computadoras

Licenciatura en informática
Tecnicatura Universitaria en Programación Informática
Departamento de Ciencia y Tecnología

V 0.58
2026

Capítulo 1

Presentación

El presente documento es un texto de apoyo a las clases de la asignatura Organización de Computadoras de las carreras Licenciatura en Informática y Tecnicatura Universitaria en Programación Informática de la **Universidad Nacional de Quilmes**. Presenta los temas según el recorrido que se realiza en las clases y se utiliza una arquitectura conceptual estructurada en diferentes versiones con una complejidad creciente y con objetivos específicos planteados por diferentes necesidades de programación.

La propuesta original de la arquitectura data del año 2013 y fue consolidándose a través de la experiencia didáctica de los diferentes grupos docentes. Si bien este texto está lejos de ser definitivo es el resultado de un gran esfuerzo de construcción colectiva.

Participaron de esta construcción

Escribieron	**Colaboraron**
Chiara Forti Dono	Axel Lopez Garabal
Candela Marzaroli	David Gonzalez
Denise Pari	Damian Romairone
Duglas Español	Esteban Dimitroff
Federico Martínez	Flavia Fernandez
Flavia Saldaña	Federico Salguero
Franco Garcino	Hector Maidana
Ian Ghioni	Ignacio Ferro
Mara Dalponte	Ignacio Mendoza
Ornella Bottiggi	Kevin Stanley
Pablo Pissi	Lucía Canosa
Pablo Nieloud	Martín Enriquez
Susana Rosito	Maximiliano Díaz
Tatiana Molinari	Nahuel Iglesias
Warmi Guercio	Martín Gonzalez Buitrón
	Verónica Fernandez Fattori

Este documento está realizado bajo la licencia Creative Commons Atribución - compartir igual - no comercial 4.0 internacional.

Índice general

Capítulo 2

Sistemas de cómputos

2.1. Evolución de las computadoras

En este capítulo se desarrolla un relato histórico de las necesidades de cómputo de la humanidad y se describen los aportes mas significativos. Si bien el relato intenta ser cronológico, se ve que hubieron avances científicos simultáneos.

Prehistoria

1642: Blaise Pascal Nació en Clermont-Ferrand, Francia, el 19 de Junio de 1623. Hijo de un recaudador de impuestos y miembro de la alta burguesía, el joven Blaise Pascal no tuvo una instrucción formal y fue educado por su padre. Su juventud transcurrió entre los salones de la nobleza y los círculos científicos de la sociedad francesa de la época. Cuando apenas contaba con 19 años Blaise Pascal empezó a construir una máquina que le permitiera automatizar las sumas y restas, que fue concluida 3 años más tarde. En 1649 gracias a un decreto real obtuvo el monopolio para la fabricación y producción de su máquina de calcular conocida como la *Pascalina* (ver Figura 2.1) que realizaba operaciones (sumas y restas) en base decimal de hasta 8 dígitos.

Figura 2.1: La Pascalina

1671: Gottfried Leibniz Nació el 10 de Julio de 1646 en Leipzig, Alemania. Realizó estudios de Leyes en la universidad de su ciudad natal y en 1675 estableció los fundamentos para el cálculo integral y diferencial. En 1676 publicó su

Figura 2.2: Tarjetas perforadas y telar de Jackard

Nuevo Método para lo Máximo y Mínimo, una exposición de cálculo diferencial. Fue filósofo, matemático y logístico. En 1670, Leibniz mejora la máquina inventada por Blaise Pascal, al agregarle capacidades de multiplicación, división y raíz cúbica. En 1679 crea y presenta el modo aritmético binario, basado en ceros y unos , lo cual serviría unos años más tarde para estandarizar la representación y procesamiento de información en las computadoras modernas.

1750 : Tarjetas perforadas Aproximadamente en este año se comienza a usar las tarjetas perforadas para especificar patrones de tejido que luego son interpretadas manualmente por los tejedores. Esto permite formalizar y recordar los patrones a través de un lenguaje que consiste en 2 estados (presencia o ausencia de perforación) en lugares específicos.

1801: Jacquard Joseph Marie Charles (7 julio de 1752 - 7 agosto de 1834), conocido como Joseph Marie Jacquard, fue un tejedor y comerciante francés que participó en el desarrollo y dio su nombre al primer telar programable con tarjetas perforadas: el telar de Jacquard (ver figura 2.2). Hijo de un obrero textil, trabajó de niño en telares de seda, y posteriormente automatizó esta tarea con el uso de **tarjetas perforadas**. Conforme fue creciendo, fue ideando distintos modos de resolver uno de los principales problemas que tenían los telares de esa época: empalmar los hilos rotos. Su telar fue presentado en Lyon en 1805. Aunque su invento revolucionó la industria textil, inicialmente sufrió el rechazo de los tejedores, incluso quemaron públicamente uno de sus telares. El método de su telar dió lugar al paradigma de la primera máquina computacional, desarrollada por Charles Babbage.

1822: Charles Babbage y Ada Lovelace Charles Babbage fue un matemático británico y científico de la computación. Diseñó y parcialmente implementó una máquina para calcular tablas de números. También diseñó, pero

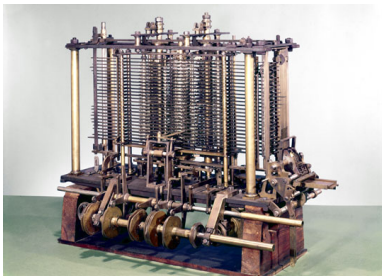


Figura 2.3: Motor de Babbage

nunca construyó, la máquina analítica para ejecutar programas de tabulación o computación. Es una de las primeras personas en concebir la idea de lo que hoy llamaríamos una computadora, por lo que se le considera como *El Padre de la Computación*.

En 1812 Babbage intentó encontrar un método por el cual se pudieran hacer cálculos automáticamente por una máquina, eliminando errores debidos a la fatiga o aburrimiento que sufrían las personas encargadas de compilar las tablas matemáticas de la época. Presentó un modelo que llamó máquina diferencial en la *Royal Astronomical Society* en 1822. Su propósito era tabular polinomios usando un método numérico llamado el método de las diferencias. La sociedad aprobó su idea, y apoyó su petición de una concesión de 1500 £ otorgadas para este fin por el gobierno británico en 1823. Babbage comenzó la construcción de su máquina, pero ésta nunca funcionó correctamente debido a que la fricción de los engranajes internos y las vibraciones impedían alcanzar la precisión que se necesitaba (sin embargo, en 1991 el Museo de Ciencias de Londres, construyó una máquina diferencial basándose en los dibujos de Babbage y utilizando sólo técnicas disponibles en aquella época y la máquina funcionó sin problemas). Mas tarde, entre 1833 y 1842, Babbage intentó construir una máquina que fuese programable para hacer cualquier tipo de cálculo, no sólo los referentes al cálculo de tablas logarítmicas o funciones polinómicas. Ésta fue la máquina analítica. El diseño se basaba en el telar de Joseph Marie Jacquard, el cual usaba tarjetas perforadas para determinar cómo una costura debía ser realizada. Babbage adaptó su diseño para conseguir calcular funciones analíticas. La máquina analítica tenía dispositivos de entrada basados en las tarjetas perforadas de Jacquard, un procesador aritmético, que calculaba números, una unidad de control que determinaba qué tarea debía ser realizada, un mecanismo de salida y una memoria donde los números podían ser almacenados hasta ser procesados.

Ada Byron, también conocida como Lady Lovelace, nació en 1815. Hija del poeta Lord Byron y de la activista política y matemática Anne Isabella Noel, Lovelace se destacó, gracias a su herencia, en el campo de la escritura y matemática. Durante su juventud, e impulsada por la curiosidad, entabló una amistad con Charles Babbage quien había diseñado la máquina analítica. Entre 1842 y 1843, tradujo un artículo del Ingeniero Militar italiano Luigi Menabrea sobre la máquina al cual le añadió un amplio conjunto de notas de la propia Lovelace, en un trabajo que denominó Notas y el cual condensa lo que se considera co-

mo el **primer programa** de computadora, esto es, un **algoritmo codificado para que una máquina lo procese**. Ada fue la primera mujer en idear un algoritmo que podría ser procesado por una máquina lo que la convierte en la primera programadora de computadoras del mundo.

1889: Máquina tabuladora de Hollerith Entre los años 1880 y 1890 se realizaron censos en los estados unidos, los resultados del primer censo se obtuvieron después de 7 años, por lo que se suponía que los resultados del censo de 1890 se obtendrían entre 10 a 12 años, es por eso que Herman Hollerith propuso la utilización de su sistema basado en tarjetas perforadas, y que fue un éxito ya que a los seis meses de haberse efectuado el censo de 1890 se obtuvieron los primeros resultados, los resultados finales del censo fueron luego de 2 años, el sistema que utilizaba Hollerith ordenaba y enumeraba las tarjetas perforadas que contenía los datos de las personas censadas, fue el primer uso automatizado de una máquina. Al ver estos resultados Hollerith funda una compañía de máquinas tabuladoras que posteriormente pasó a ser la International Business Machines (IBM).

Primera generación de computadoras: Tubos de vacío

1944 : MARK 1 (Harvard University) El IBM Automatic Sequence Controlled Calculator (ASCC), más conocido como Harvard Mark I o Mark I, fue el primer ordenador electromecánico, construido en IBM y enviado a Harvard en 1944. Tenía 760.000 ruedas y 800 kilómetros de cable y se basaba en la máquina analítica de Charles Babbage.

El computador empleaba señales electromagnéticas para mover las partes mecánicas. Esta máquina era lenta (tomaba de 3 a 5 segundos por cálculo) e inflexible (la secuencia de cálculos no se podía cambiar); pero ejecutaba operaciones matemáticas básicas y cálculos complejos de ecuaciones sobre el movimiento parabólico.

Funcionaba con relés, se programaba con interruptores y leía los datos de cintas de papel perforado. La Mark I se programaba recibiendo sus secuencias de instrucciones a través de una cinta de papel, en la cual iban perforadas las instrucciones y números que se transferían de un registro a otro por medio de señales eléctricas.

Cuando la máquina estaba en funcionamiento el ruido que producía era similar al que haría un habitación llena de personas mecanografiando de forma sincronizada. El tiempo mínimo de transferencia de un número de un registro a otro y en realizar cada una de sus operaciones básicas (resta, suma, multiplicación y división) era de 0,3 segundos. Aunque la división y la multiplicación eran más lentas.

La capacidad de modificación de la secuencia de instrucciones con base en los resultados producidos durante el proceso de cálculo era pequeño. La máquina podía escoger de varios algoritmos para la ejecución de cierto cálculo. Sin embargo, para cambiar de una secuencia de instrucciones a otra era costoso, ya que la máquina se tenía que detener y que los operarios cambiaran la cinta de control. Por tanto, se considera que la Mark I no tiene realmente saltos incondicionales. Aunque, posteriormente se le agregó lo que fue llamado Mecanismo Subsidiario de Secuencia (era capaz de definir hasta 10 subrutinas, cada una de las cuales podía tener un máximo de 22 instrucciones), que estaba compuesto

de tres tablones de conexiones que se acompañaban de tres lectoras de cinta de papel. Y se pudo afirmar que la Mark I, podía transferir el control entre cualquiera de las lectoras, dependiendo del contenido de los registros.

1946 : ENIAC (University of Pensilvania) ENIAC es un acrónimo de Electronic Numerical Integrator And Computer (Computador e Integrador Numérico Electrónico), utilizada por el Laboratorio de Investigación Balística del Ejército de los Estados Unidos. Se ha considerado a menudo la primera computadora de propósito general, aunque este título pertenece en realidad a la computadora alemana Z1. Además está relacionada con el Colossus, que se usó para descifrar código alemán durante la Segunda Guerra Mundial y destruido tras su uso para evitar dejar pruebas, siendo recientemente restaurada para un museo británico. Era totalmente digital, es decir, que ejecutaba sus procesos y operaciones mediante instrucciones en lenguaje máquina, a diferencia de otras máquinas computadoras contemporáneas de procesos analógicos. Presentada en público el 15 de febrero de 1946.

La ENIAC fue construida en la Universidad de Pensilvania por John Presper Eckert y John William Mauchly, pesaba 27 Toneladas, medía 2,4 m x 0,9 m x 30 m y ocupaba una superficie de 167 m^2. Operaba con un total de 17.468 válvulas electrónicas o tubos de vacío, 7.200 diodos de cristal, 1.500 conmutadores electromagnéticos y relés, 70.000 resistencias, 10.000 condensadores y 5 millones de soldaduras. La ENIAC permitía realizar cerca de 5000 sumas y 300 multiplicaciones por segundo. En su utilización requería la operación manual de unos 6.000 interruptores y la modificación de su programa o software demoraba semanas de instalación manual.

La ENIAC elevaba la temperatura del local a 50 grados. Para efectuar las diferentes operaciones era preciso cambiar, conectar y reconectar los cables como se hacía, en esa época, en las centrales telefónicas, de allí el concepto. Este trabajo podía demorar varios días dependiendo del cálculo a realizar.

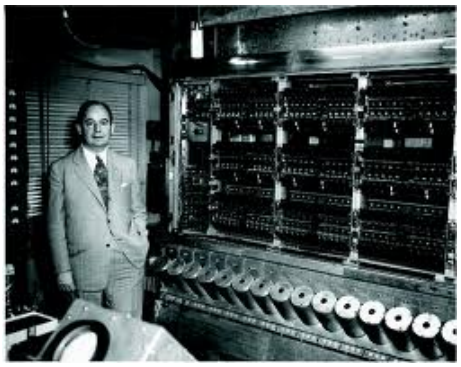


Figura 2.4: John Von Neumann

1952 : IAS (Princeton) El IAS machine fue el primer computador digital construido por el Instituto para el Estudio Avanzado (IAS, por sus siglas en inglés de Institute for Advanced Study), en Princeton, NJ, Estados Unidos.

El artículo que describe el diseño del IAS machine fue editado por John Von Neumann, un profesor de matemáticas tanto en la Universidad de Princeton como en el Instituto de Estudio Avanzado. El computador fue construido a partir de 1942 hasta 1951 bajo su dirección. El IAS se encontraba en operación limitada en el verano de 1951 y plenamente operativo el 10 de junio de 1952.

La máquina era un computador binario con palabras de 40 bits, capaz de almacenar 2 instrucciones de 20 bit en cada palabra. La memoria era de 1024 palabras (5.1 Kilobytes). Los números negativos se representaban mediante formato "complemento a dos". Tenía dos registros: el acumulador (AC) y el Multiplicador/Cociente (MQ).

Aunque algunos afirman que el IAS machine fue el primer diseño para mezclar los programas y datos en una sola memoria, que se había puesto en práctica cuatro años antes por el 1948 Manchester Small Scale Experimental Machine (MSSEM).

Von Neumann mostró cómo la combinación de instrucciones y datos en una memoria podría ser utilizada para implementar bucles, por ejemplo: mediante la modificación de las instrucciones de rama en un bucle completo. La demanda resultante de que las instrucciones y los datos se colocaran en la memoria más tarde llegó a ser conocida como el cuello de botella de Von Neumann.

Mientras que el diseño estaba basado en tubos de vacío llamado RCA Selectron para la memoria, problemas con el desarrollo de estos complejos tubos obligó el cambio al uso de los tubos de Williams. Sin embargo, utilizó cerca de 2300 tubos en los circuitos. El tiempo de adición (operación de suma) fue de 62 microsegundos y el tiempo de multiplicación fue de 713 microsegundos. Era una máquina asíncrona, es decir, que no había reloj central que regulara el calendario de las instrucciones. Una instrucción empieza a ejecutarse cuando la anterior ha terminado.

1951 UNIVAC I Las computadoras UNIVAC I fueron construidas por la división UNIVAC de Remington Rand (sucesora de la Eckert-Mauchly Computer Corporation, comprada por Rand en 1951). Era una computadora que pesaba 7.250 kg, estaba compuesta por 5000 tubos de vacío, y podía ejecutar unos 1000 cálculos por segundo. Era una computadora que procesaba los dígitos en serie. Podía hacer sumas de dos números de diez dígitos cada uno, unas 100.000 por segundo. Funcionaba con un reloj interno con una frecuencia de 2,25 MHz, tenía memorias de mercurio. Estas memorias no permitían el acceso inmediato a los datos, pero tenían más fiabilidad que las memorias de tubos de rayos catódicos, que son los que se usaban normalmente.

El primer UNIVAC fue entregado a la Oficina de Censos de los Estados Unidos (United States Census Bureau) el 31 de marzo de 1951 y fue puesto en servicio el 14 de junio de ese año. El quinto, construido para la Comisión de Energía Atómica (United States Atomic Energy Commission) fue usado por la cadena de televisión CBS para predecir la elección presidencial estadounidense de 1952. Con una muestra de apenas el 1 % de la población votante predijo correctamente que Eisenhower ganaría, algo que parecía imposible.

Además de ser la primera computadora comercial estadounidense, el UNIVAC I fue la primera computadora diseñada desde el principio para su uso en administración y negocios (es decir, para la ejecución rápida de grandes cantidades de operaciones aritméticas relativamente simples y transporte de datos,

a diferencia de los cálculos numéricos complejos requeridos por las computadoras científicas). UNIVAC competía directamente con las máquinas de tarjeta perforada hechas principalmente por IBM; curiosamente, sin embargo, inicialmente no dispuso de interfaz para la lectura o perforación de tarjetas, lo que obstaculizó su venta a algunas compañías con grandes cantidades de datos en tarjetas debido a los potenciales costos de conversión. Esto finalmente se corrigió, añadiéndole un equipo de procesamiento de tarjetas fuera de línea, los convertidores UNIVAC de tarjeta a cinta y de cinta a tarjeta, para la transferencia de datos entre las tarjetas y las cintas magnéticas que empleaba UNIVAC nativamente.

Figura 2.5: UNIVAC 1

Segunda generación de computadoras: Transistores

El primer gran cambio en la computadora electrónica ocurrió con el reemplazo de los tubos de vacío por transistores. El transistor es mas económico, pequeño y disipa menos calor. A diferencia del tubo de vacío, que requiere cables, platos metálicos, una cápsula de vidrio, etc, el transistor es un dispositivo de estado sólido que está fabricado con silicio. El uso del transistor define la segunda generación de computadoras, y cada nueva generación se caracteriza por mayor capacidad de procesamiento, mayor capacidad de memoria y menor tamaño que la anterior.

Figura 2.6: IBM 701

1952 IBM 701 Desde la introducción de la serie 700 en 1952 al lanzamiento del último modelo de la serie 7000 en 1964, esta línea de productos IBM mostró una evolución que es típica de las computadoras: los modelos sucesivos de una línea muestran un desempeño mejorado, mayor capacidad y menor costo.

Tercera generación: Circuitos integrados

Los primeros computadores de la segunda generación contenían aproximadamente 10000 transistores, pero estos números crecieron hasta cientos de miles, haciendo que la fabricación de las computadoras mas poderosas sea cada vez mas compleja e impracticable.

1964 IBM 360 El IBM S/360 fue el primer computador en usar microprogramación, y creó el concepto de familia de arquitecturas . La familia del 360 consistió en 6 ordenadores que podían hacer uso del mismo software y los mismos periféricos. El sistema también hizo popular la computación remota, con terminales conectados a un servidor, por medio de una línea telefónica. Así mismo, es célebre por contar con el primer procesador en implementar el algoritmo de Tomasulo en su unidad de punto flotante.

El IBM 360 es uno de los primeros computadores comerciales que usó circuitos integrados, y podía realizar tanto análisis numéricos como administración o procesamiento de archivos. Fue el primer computador en ser atacado con un virus en la historia de la informática; y ese primer virus que atacó a esta máquina IBM Serie 360 (y reconocido como tal), fue el *Creeper*, creado en 1972. Inicialmente, IBM anunció una familia de seis ordenadores y de cuarenta periféricos, pero finalmente entregó catorce modelos, incluyendo los modelos on-off para la NASA. El modelo más económico era el S/360/20 con tan solo 4K de memoria principal, ocho registros de 16 bits en vez de los dieciséis registros de 32 bits del 360s original, y un conjunto de instrucciones que era un subconjunto del usado por el resto de la gama.

El modelo 44 (1966) fue una variante cuyo objetivo era el mercado científico de gama media que tenía un sistema de punto flotante pero un conjunto de instrucciones limitado.

Aunque las diferencias entre modelos fueron sustanciales (por ejemplo: presencia o no de microcodigo) la compatibilidad entre ellos fue muy alta. Salvo en los casos específicamente documentados, los modelos fueron arquitectónicamente compatibles, y los programas portables.

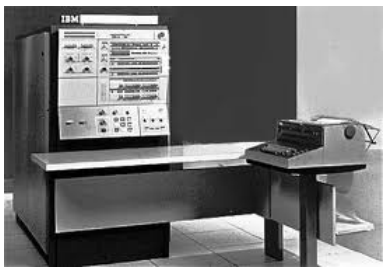


Figura 2.7: IBM 360

Figura 2.8: PDP8

1964 PDP-8 La PDP-8 (Programmed Data Processor - 8), fue la primera minicomputadora comercialmente exitosa, con más de 50 000 unidades vendidas, creada por Digital Equipment Corporation (DEC) en abril de 1965. Se la considera minicomputadora dado que podía ubicarse sobre un escritorio y resultaba económica pues podía haber una para cada técnico de laboratorio.

Los lenguajes soportados por PDP-8 fueron el Basic, Focal 71, y Fortran II/IV.

Cuarta generación: Microelectrónica

La microelectrónica significa literalmente, electrónica pequeña. Desde el comienzo de la electrónica digital y la industria de computadoras, ha habido una tendencia persistente de reducir los tamaños de los circuitos electrónicos.

1974 Intel 8080 El Intel 8080 fue un microprocesador temprano diseñado y fabricado por Intel. La CPU de 8 bits fue lanzado en abril de 1974. Corría a 2 MHz, y generalmente se le considera el primer diseño de CPU microprocesador verdaderamente usable.

Varios fabricantes importantes fueron segundas fuentes para el procesador, entre los cuales estaban AMD, Mitsubishi, NatSemi, NEC, Siemens, y Texas Instruments. También en el bloque oriental se hicieron varios clones sin licencias, en países como la Unión de Repúblicas Socialistas Soviéticas y la República Democrática de Alemania. El Intel 8080 fue el sucesor del Intel 8008, esto se debía a que era compatible a nivel fuente en el lenguaje ensamblador porque usaban el mismo conjunto de instrucciones desarrollado por Computer Terminal Corporation. Con un empaquetado más grande, DIP de 40 pines, se permitió al 8080 proporcionar un bus de dirección de 16 bits y un bus de datos de 8 bits, permitiendo el fácil acceso a 64 KB de memoria. Tenía siete registros de 8 bits, seis de los cuales se podían combinar en tres registros de 16 bits, un puntero de pila en memoria de 16 bits que reemplazaba la pila interna del 8008, y un contador de programa de 16 bits.

1976 Apple 1 El Apple I fue uno de los primeros computadores personales, y el primero en combinar un microprocesador con una conexión para un teclado y un monitor. Fue diseñado y hecho a mano por Steve Wozniak originalmente para uso personal. Un amigo de Steve Wozniak, Steve Jobs, tuvo la idea de vender el computador. Fue el primer producto de Apple, presentado en abril de 1976 en el Homebrew Computer Club en Palo Alto, California y se fabricaron

200 unidades. A diferencia de otras computadoras para aficionados de esos días, que se vendía en kits, el Apple I era un tablero de circuitos completamente ensamblado que contenía 62 chips. Sin embargo, para hacer una computadora funcional, los usuarios todavía tenían que agregar una carcasa, un transformador para fuente de alimentación, el interruptor de encendido, un teclado ASCII, y una pantalla de video. Más adelante se comercializó una tarjeta opcional que proporcionaba una interfaz para casetes de almacenamiento.

Las máquinas de la competencia como el Altair 8800 generalmente se programaban con interruptores de palanca montados en el panel frontal y usaban luces señalizadoras para la salida, (comúnmente LEDs rojos), y tenían que ser extendidas con hardware separado para permitir la conexión a un terminal de computadora o a una máquina de teletipo. Esto hizo al Apple I una máquina innovadora en su momento, a pesar de su carencia de gráficos o de capacidades de sonido.

Figura 2.9: Procesador 8080

Figura 2.10: Apple I

Capítulo 3

Sistemas de numeración

Un sistema de numeración es un conjunto de mecanismos y reglas basados en un determinado alfabeto (conjunto de símbolos), que permiten usar cadenas de esos simbolos para representar cantidades y realizar operaciones aritméticas. Los sistemas de numeración pueden clasificarse en:

- Sistemas no posicionales
- Sistemas posicionales

En esta asignatura se analizarán principalmente los sistemas de numeración posicionales, haciendo foco especialmente en los de base binaria, es decir: usando los símbolos 0 (cero) y 1 (uno). La principal característica de los sistemas posicionales está dada por la base del sistema, la cual establece la cantidad de símbolos diferentes de los que se dispone. Algunos ejemplos de este tipo de sistemas son el sistema decimal (10 símbolos), el sistema octal (8 símbolos), sistema binario (2 símbolos) y el sistema hexadecimal (16 símbolos).

3.1. Sistema Binario

Como sistema de numeración principal que hasta ahora conocemos y manejamos es el **sistema decimal**. Por ejemplo sabemos que 3548 representa el *tres mil quinientos cuarenta y ocho*. Pero ¿qué significa esto? Otra manera de indicar la misma cifra es diciendo: "*3 elementos que valen mil* sumado a *5 elementos que valen 100*, *4 elementos que valen 10* (o decenas), y *8 unidades*". Es decir que a cada símbolo se le asocia una determinada potencia de 10 según su posición, para darle distinta importancia o *peso*. Dicha potencia de 10 recibe el nombre de *peso del dígito* .

peso del dígito

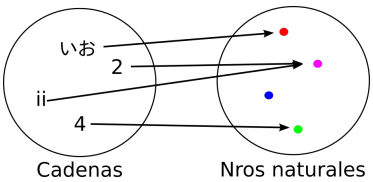


Figura 3.1: Función que relaciona cadenas con los números naturales

De esta manera, 3548 puede escribirse como sigue:

$$3548 = 3 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$$
$$= 3 \times 1000 + 5 \times 100 + 4 \times 10 + 8 \times 1$$

Pero ¿por qué se utilizan potencias de 10? Pues porque el sistema decimal tiene base 10 , y por lo tanto, 10 símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Con estos símbolos se construyen cadenas que permiten representar todos los valores del conjunto de los números naturales. Pero existen otros sistemas que tienen otros símbolos para representar los mismos números (ver figura 3.1).

Tal es el caso donde solo se dispone de un par de símbolos para definir un sistema de numeración, es decir un **sistema binario** donde la base a considerar es 2. Por lo tanto se deben utilizar potencias de 2 y los símbolos que utiliza son 0 y 1, denominados *bits* . Las secuencias de bits se denominan *cadenas binarias* .

Como todo sistema de numeración, para poder apropiarse de él es necesario estudiar el conjunto de sus mecanismos o funcionalidades. En los sistemas estudiados aquí se podrá:

- Interpretar cadenas binarias
- Representar valores naturales
- Calcular el rango del sistema
- Realizar operaciones aritméticas

Interpretación de cadenas

La interpretación es el mecanismo por el cual se determina el valor (o cantidad) que representa una cadena. Este mecanismo se denota como una función cuyo conjunto dominio son los valores y cuyo conjunto imagen son las cadenas (en este caso: binarias).

Suponer un sistema binario donde todas las cadenas tienen sólo 2 bits, y por lo tanto se tienen 4 cadenas diferentes, pues las combinaciones posibles son 4: `00`, `01`, `10` y `11`. Entonces al asignarle valores a partir del cero, y de manera ordenada, se establece que `00` representa al valor 0, `01` representa al valor 1, `10` representa al valor 2 y `11` representa al valor 3 (esto se describe en la tabla 3.1). A este sistema se lo denomina binario sin signo de 2 bits y tiene como conjunto dominio al subconjunto de los naturales $\{0, 1, 2, 3\}$. Para referirnos a este sistema utilizaremos la sintaxis: *BSS(2)*.

cadena	valor
`00`	0
`01`	1
`10`	2
`11`	3

Tabla 3.1: Sistema BSS de 2 bits - BSS(2)

cadena	Interpretación
`000`	$I_{bss(3)}(000) = 0$
`001`	$I_{bss(3)}(001) = 1 \times 2^0 = 1$
`010`	$I_{bss(3)}(010) = 1 \times 2^1 = 2$
`011`	$I_{bss(3)}(011) = 1 \times 2^1 + 1 \times 2^0 = 3$
`100`	$I_{bss(3)}(100) = 1 \times 2^2 = 4$
`101`	$I_{bss(3)}(101) = 1 \times 2^2 + 1 \times 2^0 = 5$
`110`	$I_{bss(3)}(110) = 1 \times 2^2 + 1 \times 2^1 = 6$
`111`	$I_{bss(3)}(111) = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$

Tabla 3.2: Interpretación en BSS de 3 bits - BSS(3)

Por otro lado, la asignación de valores a las cadenas que se detalla en la tabla 3.1 cumple con la aplicación de los pesos que se presentó anteriormente para el caso del sistema decimal. Por ejemplo, al decir que la cadena `10` (se lee *uno cero*, no *diez*) representa el valor 2, es posible ponderar (darle peso a) los dígitos de la cadena como sigue:

$$I_{bss(2)}(10) = 0 \times 2^0 + 1 \times 2^1 = 0 + 2 = 2$$

Este mecanismo puede extenderse a otros sistemas con mayor cantidad de dígitos en sus cadenas, o mas complejos con otro conjunto de símbolos. Por ejemplo el sistema binario sin signo de 3 bits ($BSS(3)$) se describe en la tabla 3.2.

Como último ejemplo de sistema considerar $BSS(6)$ y la cadena a interpretar `110101` (obtener el número que esta representa). La interpretación de dicha cadena aplica el proceso mencionado anteriormente: se debe asignar una potencia de base 2 a cada posición **empezando de derecha a izquierda y desde la potencia 0**:

$$I_{bss(6)}(110100) = 0\times2^0+0\times2^1+1\times2^2+0\times2^3+1\times2^4+1\times2^5 = 4+16+32 = 52$$

Tener en cuenta que el orden de escribir cada término es indistinto, es decir puede ser ascendente como el ejemplo anterior o descendente, comenzando por la potencia máxima. Esto es posible debido a que la suma es conmutativa. Lo que sí se debe respetar es la posición que tiene cada bit dentro de la cadena y por lo tanto el peso (potencia de 2) que le corresponde. La siguiente es manera de ordenar los términos de la interpretación de la cadena anterior:

$$I_{bss(6)}(110100) = 1\times2^5+1\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0 = 32+16+4 = 52$$

En los capítulos siguientes se utilizará indistintamente cualquiera de las notaciones.

Representación de valores

La representación de valores es el mecanismo para construir una cadena en un determinado sistema de numeración posicional (en este caso, el sistema binario) a partir de un valor que se quiere representar. Para hacerlo se utiliza el algoritmo de las *divisiones sucesivas*, donde x es el valor a representar:

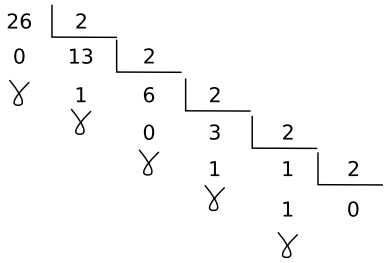


Figura 3.2: Representación del valor 26 en el sistema binario

1. Sea $w = \emptyset$ la cadena resultado (está vacía al comenzar).
2. Si $x > 0$ calcular la division entera:
   - Sea $c = x/2$ el cociente.
   - Sea r el resto, y entonces $r < 2$, es decir que es un valor en el conjunto $\{0, 1\}$.
3. Tomar r como un bit, y ubicarlo en la posición más significativa de la cadena w (la posición del extremo izquierdo).
4. Si $c > 0$ volver al paso 2 tomando c como el nuevo valor de x.

Es importante notar que el algoritmo anterior finaliza cuando se obtiene un cociente igual a 0.

Suponer por ejemplo que se necesita representar el valor 26 en el sistema binario sin restricción de la cantidad de bits, por lo que el resultado obtenido deberá ser una cadena binaria cuya longitud a priori se desconoce.

Entonces como primer ciclo se divide el 26 en 2, obteniendo como resto 0 y cociente 13. El resto 0 se utiliza como primer bit y se agrega a la cadena (`w=0`). Como el cociente es mayor a cero ($c = 13$) entonces se vuelve al paso 2, tomando $x = 13$.

En un segundo ciclo se calcula $x \div 2$ obteniendo resto 1 y cociente 6. El resto 1 es el segundo bit de la cadena, y entonces `w=10`. Como el cociente es mayor a cero ($c = 6$) entonces se vuelve al paso 2, tomando $x = 6$.

En el tercer ciclo, x toma el valor 6 y se debe calcular $6 \div 2$ obteniendo resto 0 y cociente 3. El resto 0 es el tercer bit de la cadena (`c=010`). Como el cociente es mayor a cero ($c = 3$) entonces se vuelve al paso 2, tomando $x = 3$.

En el cuarto ciclo se calcula $3 \div 2$ obteniendo resto 1 y cociente 1. Entonces el resto 1 es el cuarto bit de la cadena (`c=1010`). Como el cociente es mayor a cero ($c = 1$) entonces se vuelve al paso 2, tomando $x = 1$.

En el último ciclo se calcula $1 \div 2$ obteniendo resto 1 y cociente 0. El resto 1 es el quinto bit de la cadena (`c=11010`) y el algoritmo finaliza. Finalmente la cadena que representa al valor 26 es `c=11010`, que pertenece al sistema $BSS(5)$, lo que formalmente de denota: $R_{BSS(5)}(26) = 11010$. Este mecanismo se aprecia gráficamente en la Figura 3.2.

cadena	Interpretación	Representación	valor
`000`	$I_{bss(3)}(000) = 0$	$R(0) = 000$	0
`001`	$I_{bss(3)}(001) = 1 \times 2^0 = 1$	$R(1) = 001$	1
`010`	$I_{bss(3)}(010) = 1 \times 2^1 = 2$	$R(2) = 010$	2
`011`	$I_{bss(3)}(011) = 1 \times 2^1 + 1{,}2^0 = 3$	$R(3) = 011$	3
`100`	$I_{bss(3)}(100) = 1 \times 2^2 = 4$	$R(4) = 100$	4
`101`	$I_{bss(3)}(101) = 1 \times 2^2 + 1 \times 2^0 = 5$	$R(5) = 101$	5
`110`	$I_{bss(3)}(110) = 1 \times 2^2 + 1 \times 2^1 = 6$	$R(6) = 110$	6
`111`	$I_{bss(3)}(111) = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$	$R(7) = 111$	7

Tabla 3.3: Sistema BSS de 3 bits - BSS(3)

3.1.1. Dualidad entre Interpretación y Representación

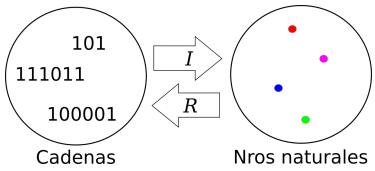


Figura 3.3: Relación dual entre las funciones $I()$ y $R()$

Los mecanismos de interpretación y la representación se presentaron como funciones y presentan una dualidad: Si la cadena `c` al ser interpretada se la asocia al valor x, entonces al representar dicho valor, se debe obtener nuevamente la cadena `c`.

Simbólicamente usaremos la notación $I(c)$ y $R(x)$ como las dos funciones del sistema binario. Entonces la dualidad mencionada se define en la siguiente ecuación y se describe gráficamente la Figura 3.3.

$$I_{sistema}(c) = x \leftrightarrow R_{sistema}(x) = c$$

Esta característica permite comprobar la validez del mecanismo aplicado (tanto sea la representación o la interpretación). Retomando el ejemplo de la sección anterior (representación del valor 26), se interpreta la cadena resultante para compararla con el valor original de x:

$$I_{BSS(5)}(11010) = 1 \times 2^1 + 1 \times 2^3 + 1 \times 2^4 = 16 + 8 + 2 = 26$$

Verificando la propiedad de dualidad:

$$I_{BSS(5)}(11010) = 26 \leftrightarrow R_{BSS(5)}(26) = 11010$$

Y con esto se concluye que la función se aplicó correctamente. Notar que este cálculo no menciona los términos que corresponden a los bits 0s, dado que el cero es el elemento neutro de la suma, y por lo tanto no impacta en el resultado final. Esta es una manera de abreviar el cálculo, pero para llegar a un resultado correcto, será necesario tener cuidado y respetar la posición de cada bit.

En la tabla 3.3 es posible ver la relación entre las cadenas y los valores que se representan en un sistema BSS de 3 bits completo, a través de ambos mecanismos de representación e interpretación.

3.1.2. Rango

Hasta este punto se describió el sistema de numeración en términos de sus símbolos, su función de interpretación y su función de representación, pero queda pendiente analizar la capacidad de representación en término del conjunto de números representables: es decir de su *rango* .

rango

Si bien un sistema numérico permite construir un conjunto infinito de cadenas, esto no se cumple en el contexto de un sistema de cómputos pues se tiene una cantidad limitada de bits, por lo que el conjunto de números representables también será limitado. Por este motivo se dice que el sistema es un *sistema restringido* a n cantidad de bits.

Por lo tanto, el rango de un sistema de numeración es **el conjunto de valores representables del sistema**. El cual se denotará a partir de sus extremos, es decir el valor mínimo y el máximo.

Considerar por ejemplo un sistema binario restringido a 3 bits y que sólo contemple los números naturales, denotado como $BSS(3)$.

Para analizar su rango se debe determinar el valor mínimo y el máximo representable. Para el primer caso se interpreta la primer cadena (en un orden léxicográfico), que se construye sólo con ceros: `000`. Para conocer qué valor representa, debemos aplicar la función de interpretación:

$$I_{BSS(3)}(000) = 0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 = 0$$

Para el segundo caso se interpreta la útima cadena, compuesta por todos unos: `111`, obteniendo:

$$I_{BSS(3)}(111) = 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = 7$$

Es decir que el rango del sistema $BSS(3)$ son todos los números naturales comprendidos entre el 0 y el 7 inclusive, denotándose: $[0, 7]$. De esta manera el conjunto de valores representables tiene 8 elementos. Además con 3 bits se pueden construir 8 cadenas de números representables. Esto se obtiene como resultado del cálculo: $2^3 = 8$. Notar que 2 es la base, y la potencia 3 es la cantidad de bits del sistema.

Generalizando lo anterior, se puede decir que en un sistema $BSS(n)$ se pueden representar 2^n valores y su rango será $[0, 2^n - 1]$.

3.1.3. Operaciones Aritméticas

Como se mencionó previamente, los sistemas de numeración deben proveer, además de los mecanismos de interpretación y representación, algoritmos para operar con esas cadenas: suma, resta, multiplicación y división.

Tal como se enseña el mecanismo de suma *"en papel"* para el sistema decimal, en el sistema binario también conviene pensar la suma como un algoritmo organizado por columnas. Es decir que se suman las columnas empezando por la menos significativa, es decir la de menor peso (extremo de la derecha) y *"acarreando"* a la siguiente columna si se alcanza o supera la base. Por ejemplo, si se requiere sumar dos dígitos cuya suma supera el límite de la base (que en decimal es 10), como por ejemplo $9 + 3$, en el resultado se coloca el valor 2 y *"se acarrea un 1"*. Esto se debe a que $9 + 3 = 12$ que puede reescribirse como $12 = 10 + 2$, es decir: una decena y dos unidades.

$$\begin{array}{rr} {\scriptstyle 1} & \\ + & 9 \\ & 3 \\ \hline {\scriptstyle 1} & 2 \end{array}$$

A la práctica de *"llevarse un 1"* se la denomina *acarreo* y representa el concepto de utilizar un nuevo orden de magnitud. En el ejemplo, al alcanzar un conjunto de 10 unidades se obtiene una decena, y esto no puede representarse con un sólo dígito.

acarreo

En el caso del sistema binario el algoritmo de suma sigue la misma lógica pero utilizando la base 2, como se describe a continuación. Sean las cadenas A y B los operandos de la suma:

1. Comenzar con la columna del bit menos significativo (extremo derecho de las cadenas). Al comenzar, el bit de acarreo es 0 ($c = 0$).

2. Se definen las siguientes variables:
   Sea a el bit en la columna actual correspondiente al operando A.
   Sea b el bit en la columna actual correspondiente al operando B.
   Sea c el bit de acarreo de la columna anterior (el que *"me llevé"* previamente.)
   Considerar la cantidad total de 1s (unos) como $t = a + b + c$.
   Entonces:

   a) Si $t = 0$ (no hay ningún 1): el resultado de la columna es 0 ($r = 0$) y no se genera acarreo ($c = 0$).

   b) Si $t = 1$ (hay un sólo 1): el resultado de la columna es 1 ($r = 1$) y no se genera acarreo ($c = 0$).

   c) Si $t = 2$ (hay dos 1s): el valor 2 en binario se representa con la cadena `10` que no cabe en una sóla columna, por lo que el resultado de la columna es 0 ($r = 0$) y se genera acarreo ($c = 1$).

   d) Si $t = 3$ (todos son 1s): el valor 3 en binario se representa con la cadena `11`, y como en el caso anterior no es representable en una columna, por lo que el resultado de la columna es 1 ($r = 1$) y se genera acarreo: ($c = 1$).

3. Tomar la siguiente columna hacia la izquierda y volver al paso 2.

Por ejemplo, calcular la siguiente suma: `001 + 011`.

Al comenzar el bit de acarreo tiene el valor cero ($c = 0$), ya que al ser la primera columna no hay acarreo anterior. En el primer ciclo se toma la columna del bit menos significativo: $1+1$ y a partir de esto se definen las variables : $a = 1$ y $b = 1$. Entonces $t = 1 + 1 + 0 = 2$ y por lo tanto el resultado de la columna es 0, con acarreo ($c = 1$) que lo denotamos con un (1) bit rojo en la columna siguiente.

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

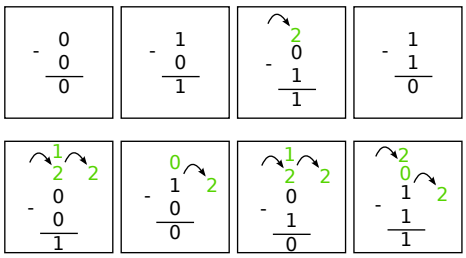


Figura 3.4: Casos posibles en una resta

En el segundo ciclo se toma la columna $0+1$: $a=0$, $b=1$ y $c=1$ (el bit del acarreo anterior en color rojo). Entonces $t=0+1+1=2$ y nuevamente el resultado de la columna es 0, generando un nuevo acarreo (ver bits en en rojo).

$$\begin{array}{rccc} & \color{red}{1} & \color{red}{1} & \\ & 0 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline & & 0 & 0 \end{array}$$

Finalmente, en el tercer ciclo (tercera y última columna) se tiene $a=0$, $b=0$ y $c=1$. Entonces $t=0+0+1=1$, donde el resultado de la columna es 1, y sin generar un nuevo acarreo.

$$\begin{array}{rccc} & \color{red}{1} & \color{red}{1} & \\ & 0 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 \end{array} \leftarrow \text{cadena resultante}$$

Para comprobar la correctitud del algoritmo anterior, se deberán interpretar las cadenas de ambos operadores, calcular el resultado esperado y compararlo con la interpretación de la cadena resultado.

Al interpretar las cadenas de los operandos se tiene que:

$$I_{bss(3)}(001) = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 = 1$$

$$I_{bss(3)}(011) = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 = 3$$

El resultado esperado es $1+3=4$, donde su representación en BSS(3) es

$$R_{bss(3)}(4) = 100$$

Esta cadena coincide con la cadena resultante del algoritmo de suma, por lo que se concluye que el algoritmo se aplicó correctamente.

Resta Binaria

Al igual que en la suma, al momento de restar, se aplicará un algoritmo que procesa columnas de derecha a izquierda, generando un acarreo hacia la derecha (o *pidiendo prestado a la siguiente columna*) en caso de ser necesario, es decir, cuando el resultado de la columna podría ser menor a cero. Esta situación se da en 2 situaciones: cuando el bit del operando minuendo (operando A) es menor

al bit del operando sustraendo (operando B), o cuando la columna "le prestó uno" a la columna siguiente.

Recuperando el algoritmo del **sistema decimal**, cuando es necesario se pide a la columna de la izquierda una magnitud mayor, es decir, se pide una decena, que tiene relación con la base del sistema decimal. Análogamente, en el **sistema binario**, se pide a la columna izquierda una magnitud mayor, que resulta en 2 unidades, pues es un sistema de base 2.

En la Figura 3.4 se describen todos los casos posibles para calcular la resta en una columna, es decir bit a bit.
Tener en cuenta que la notación propuesta es semejante a la resolución manual que solemos realizar en un papel. Los valores resaltados en color verde hacen referencia a los préstamos entre columnas (como puede suceder con operadores cuyas cadenas tienen más de un bit). Los préstamos se denotan con un valor 2, dado que se prestan 2 unidades, puesto que es la base del sistema en el cual estamos operando.

Los primeros 4 casos, representan las situaciones más triviales de la operación, sólo el 3er caso requiere de un préstamo. Veamos: se necesita restar $0 - 1$, es decir que no alcanza las unidades del minuendo para la resta, por lo que se solicita un préstamo a la columna de la izquierda (notar la flecha), quedando ahora el minuendo en 2, $(0 + 2 = 2)$. De esta manera se puede realizar la resta entre ambos bits, $2-1 = 1$. Notar que en binario el valor 2 se escribe como 10.

Los últimos 4 casos contemplan situaciones con préstamos, ya sea entre la columna siguiente como la anterior. Analicemos el primer caso (los demás siguen la misma lógica).
Se quiere restar $0 - 0$, pero previamente la columna de la derecha nos ha solicitado un préstamo, y como no alcanza para prestar (pues 0), solicitamos un préstamo a la columna de la izquierda, por lo que anotamos el valor 2 $(0+2 = 2)$. Pero no olvidemos que tenemos un préstamo pendiente, así que en realidad la columna se queda con 1. Con el cual ya podemos concretar nuestra resta, quedando $1 - 0 = 1$.

Hasta aquí la explicación de cada caso por separado, pero al momento de automatizar el cálculo, se necesita un algoritmo que contemple todas las combinaciones posibles. Para lo cual contamos con el siguiente algoritmo que resulta en una generalización de los casos mencionados previamente. Tener en cuenta que, como para el caso de la suma, este algoritmo sólo evalúa columna a columna, bit a bit, utilizando variables para su resolución.

1. Comenzar con la columna menos significativa (extremo derecho de las cadenas). Al comenzar, el bit de préstamo, conocido como *carry/borrow* (**carry** para la suma y **borrow** para la resta), es 0 $(c = 0)$.

2. Sea a el bit del operando A en la columna actual.
   Sea b el bit del operando B en la columna actual.
   Sea c el bit prestado de la columna siguiente (de la izquierda) Sea t el resultado

3. Si $(a - c) \geq b$ entonces
   el resultado es $t = (a - c) - b$ y no hay un pedido de préstamo, quedando $c = 0$.
   Sino, en caso contrario:

   a) Se requiere un préstamo de la siguiente columna (la de la izquierda) quedando como resultado $t = (a + 2) - c - b$. Recordar que se piden 2 unidades (base binaria), por lo que se le suma dicha base al bit actual $(a + 2)$.

   b) Se actualiza el bit del borrow $c = 1$

4. Si quedan columnas, avanzar una hacia la izquierda y volver a realizar desde el paso 2.

Nota: tener en cuenta que si el operando A es menor al operando B, el resultado de la resta no sera un valor representable en dicho sistema.

Suponer por ejemplo que se necesita calcular la resta de las siguientes cadenas `101 - 011`. El algoritmo de la resta para este ejemplo se aplica como sigue.

Al comenzar, $c = 0$ y la columna menos significativa presenta: `1-1` , por lo que $a = 1$y $b = 1$. Se debe analizar si $(a - c) \geq b$. Como ocurre que $a - c = 1 - 0 = 1$ es mayor a $b = 1$ entonces $t = (a - c) - b = (1 - 0) - 1 = 1 - 1 = 0$ y no hay préstamo (sigue ocurriendo que $c = 0$). Gráficamente, el primer paso se ve como sigue:

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

En el segundo ciclo se tiene $(0 - 1)$ y entonces $a = 0$, $b = 1$ y $c = 0$ (no hubo préstamo desde la columna anterior). Analizando la condición: ¿$(a - c) \geq b$? se ve que no no se cumple pues $a - c = 0 - 0 = 0 < 1$ y entonces se requiere un préstamo. Así $t = (a+2) - c - b = (0+2) - 0 - 1 = 2 - 1 = 1$. Luego se actualiza $c = 1$, el cual se muestra en color rojo.

$$\begin{array}{rccc} & \color{red}{1} & & \\ & 1 & 0 & 1 \\ - & 0 & 1 & 1 \\ \hline & & 1 & 0 \end{array}$$

En el tercer ciclo se tiene $1 - 0$ y entonces $a = 1$, $b = 0$ y $c = 1$ (pues quedó actualizado del procesamiento de la columna anterior). Analizando la condición: ¿$(a - c) \geq b$? se ve que se cumple pues $a - c = 1 - 1 = 0 \geq 0$ y por lo tanto no es necesario solicitar un préstamo, quedando como resultado $t = (a - c) - b = (1 - 1) - 0 = 0 - 0 = 0$.

$$\begin{array}{rccc} & \color{red}{1} & & \\ & 1 & 0 & 1 \\ - & 0 & 1 & 1 \\ \hline & 0 & 1 & 0 \end{array} \leftarrow \text{cadena resultante}$$

Notar que el carry c, conocido como borrow en la resta, se visualiza en esta última columna, debido a que en este paso es donde se encuentra activado en 1.

El algoritmo finaliza obteniendo como cadena resultante `010`. Pero al igual que con la suma, es importante poder verificar el resultado. Para ello se deben interpretar las cadenas de ambos operandos, calcular el resultado esperado y compararlo con la interpretación de la cadena resultado.

Al interpretar las cadenas de cada operando se tiene que:

$$I_{bss(3)}(101) = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 = 1 + 4 = 5$$

$$I_{bss(3)}(011) = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 = 1 + 2 = 3$$

El resultado esperado es $5 - 3 = 2$, y su representación en BSS es

$$R_{bss(3)}(2) = 010$$

Esta cadena coincide con la cadena resultante del algoritmo. Esto permite concluir que el resultado es el correcto por lo que el algoritmo se aplicó correctamente.

3.1.4. Desplazamiento de cadenas

Como ya se vió en las secciones anteriores, la suma y resta en el sistema binario se realiza mediante un algoritmo de columnas de manera similar al sistema decimal, variando únicamente la base del sistema utilizado (en lugar de ser 10, es 2). Otra característica que es posible transpolar del sistema decimal al sistema binario, es el desplazamiento de cadenas.

Suponer que se tiene el valor decimal 17. Si multiplicamos dicho valor por 10 se obtiene como resultado el 170. Si se compara el 17 con el 170, es posible ver que este último se construye con la cadena 17 y un 0 a su derecha. Ahora bien, si se multiplicara el 17 por 1000, se obtendría 17000, es decir, la cadena 17 seguida de tres 0s a su derecha.

Al buscar una relación entre los valores utilizados en la multiplicación y los resultados obtenidos, se puede ver lo siguiente: Si se multiplica por 10, se añade a la cadena original, un 0 a la derecha. Esto se debe a que el valor 10 puede expresarse como 10^1 ($10^1 = 10$).
De la misma manera, al multiplicar por 1000, se añaden tres 0s a la derecha de la cadena original, dado que el valor 1000 se lo puede expresar como 10^3 ($10^3 = 1000$)

Es decir que al multiplicar un valor **n** por 10^m se obtiene como resultado el valor **n** con tantos 0s a su derecha como indica **m**. Esto representa un desplazamiento hacia la izquierda de la cadena que representa al valor original.

Dualmente, si se quiere conseguir un desplazamiento de **m** posiciones hacia la derecha, lo que se deberá realizar es **dividir** el valor en 10^m.
Por ejemplo, si se divide 250 en 10, el resultado es 25, es decir que se desplaza hacia la derecha el ultimo bit de la cadena original, "cayéndose o borrandose."

Esta manera de manipular las cadenas puede adaptarse al sistema binario, cambiando la escala 10^m por 2^m, es decir, usando potencias de la base del sistema binario.
En la tabla 3.4 se pueden ver más ejemplos de estas situaciones.

Cadena original	Interpretación	Desplazamiento	Representación	Cadena final
010	$I_{bss(3)}(010) = 2^1$	$2^1 * 2 = 4$	$R_{bss(3)}(4) = 100$	100
010	$I_{bss(3)}(010) = 2^1$	$2^1 \,\%2 = 1$	$R_{bss(3)}(1) = 001$	001
011	$I_{bss(3)}(011) = 2^0 + 2^1$	$3 * 2 = 6$	$R_{bss(3)}(6) = 110$	110
011	$I_{bss(3)}(011) = 2^0 + 2^1$	$3 \,\%2 = 1$	$R_{bss(3)}(1) = 001$	001

Tabla 3.4: Desplazamientos en cadenas binarias

3.2. Sistema Hexadecimal

Este sistema de numeración es un sistema posiciónal de base 16, es decir que su alfabeto tiene 16 símbolos. Este alfabeto se apoya parcialmente en el alfabeto decimal, pero agrega 6 símbolos adicionales, quedando de la siguiente manera: *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.*

Este sistema de numeración también contiene un conjunto de funcionalidades para poder operar con él. Y de la misma manera que en el sistema binario es posible:

1. Interpretar cadenas
2. Representar valores
3. Calcular su rango
4. Realizar cómputos aritméticos

Adicionalmente, se explicará un mecanismo de traducción entre cadenas binarias y cadenas hexadecimales denominado *agrupación de bits*.

3.2.1. Interpretación

Traspolando el mismo razonamiento que hicimos para el sistema binario, donde a cada dígito le cforresponde un peso segun su posición, se le asigna una determinada potencia de 16 comenzando desde la derecha (desde la posición 0). Entonces se multiplica el valor de cada dígito por su peso. Pero, ¿qué ocurre cuándo el dígito es una letra? Para ello el sistema establece una relación entre los dígitos {A,..,F} y la cantidad (o valor) que representan individualmente, como se describe a continuación:

Dígito	A	B	C	D	E	F
Valor	10	11	12	13	14	15

Consideremos los siguientes ejemplos:

- Queremos interpretar la cadena 128, entonces:

$$I_{hex(3)}(128) = 8 \times 16^0 + 2 \times 16^1 + 1 \times 16^2 = 8 + 32 + 256 = 296$$

- Ahora veamos otro ejemplo: 2A. Entonces, siguiendo la misma lógica quedaría:

$$I_{hex(2)}(2A) = A \times 16^0 + 2 \times 16^1 = 10 \times 16^0 + 2 \times 16^1 = 10 * 32 = 42$$

- Como último ejemplo, considerar la cadena `A3F`

$$I_{hex(3)}(A3F) = 15 \times 16^0 + 3 \times 16^1 + 10 \times 16^2 = 15 + 48 + 2560 = 2623$$

3.2.2. Representación

Replicando el algoritmo de representación del sistema binario, para representar valores mediante cadenas en sistema hexadecimal se deben realizar sucesivas divisiones por la base, que en este caso es 16, hasta obtener un cociente igual a 0 tomando cada resto como bits de la cadena.

Ejemplo: Se necesita representar el número 26 en hexadecimal:

1. Se divide el valor 26 por 16 hasta encontrar un cociente 0
2. Se construye la cadena tomando solo los restos, empezando por el último, y cuando el resto es mayor o igual a 10 se utiliza el caracter alfabético correspondiente (A..F).

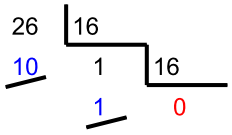


Figura 3.5: Representación del valor 26 en el sistema hexadecimal

Dado que de los restos es 10, entonces debemos traducirlo a la letra correspondiente aplicando la tabla de interpretación de hexadecimal presentada en la sección anterior. El valor **10** es equivalente a la letra `A`, quedando entonces `1A`. Esto quiere decir que el valor 26 en decimal se corresponde con la cadena `1A` en hexadecimal.

3.2.3. Rango

De la misma manera que en el sistema binario, para presentar el rango de un sistema hexadecimal, es necesario calcular el mínimo y el máximo número representable interpretando respectivamente la primera y la última cadena (en orden lexicográfico). De esta manera, el rango del sistema son todos los números comprendidos entre ambos límites. Supongamos el sistema hexadecimal de 2 dígitos:

El mínimo valor representable es el resultado de interpretar la cadena `00`, es decir:

$$I_{hex(2)}(00) = 0 \times 16^0 + 0 \times 16^1 = 0$$

El máximo valor representable es el resultado de interpretar la cadena `FF`

$$I_{hex(2)}(FF) = 15 \times 16^0 + 15 \times 16^1 = 255$$

(aplicando la tabla de interpretación de hexadecimal). Por lo tanto el rango de este sistema Hex(2) es: $[0, 255]$

Dígito	0	1	2	3	4	5	6	7
Cadena $BSS(4)$	0000	0001	0010	0011	0100	0101	0110	0111
	8	9	A	B	C	D	E	F
	1000	1001	1010	1011	1100	1101	1110	1111

Tabla 3.5: Relación entre cadenas $BSS(4)$ y dígitos del sistema hexadecimal

3.2.4. Operaciones aritméticas

En el sistema hexadecimal, siendo también posicional, la mecánica de la aritmética es análoga a los casos vistos antes, en el sentido que se considera el trabajo por columnas (desde la menos significativa a la mas significativa) y considerando la base de este sistema, que es 16 (ver algoritmos **??** y **??**)

1. Comenzar con la columna menos significativa.
2. Sea a el digito del operando A en la columna actual.
   Sea b el digito del operando B en la columna actual.
   Sea c el arrastre de la columna anterior.
   Considerar la suma entre los digitos de la columna: $t = a + b + c$
3. Analizar acarreo: ¿$t < 16$?
   - Si se cumple: el resultado intermedio (sea i) es igual a t ($i = t$) y no hay acarreo (c=0)
   - En caso contrario: $t \geq 16$ el resultado intermedio se calcula $i = t - 16$ y hay acarreo (c=1)
4. Generar el bit resultado:
   - Si i=10: el resultado final es r=A
   - Si i=11: el resultado final es r=B
   - Si i=12: el resultado final es r=C
   - Si i=13: el resultado final es r=D
   - Si i=14: el resultado final es r=E
   - Si i=15: el resultado final es r=F
   - sino: r = i
5. Volver al paso 2 con la siguiente columna hacia la izquierda

3.2.5. Agrupación de bits

Este es un método que establece una relación directa entre cadenas del sistema binario y cadenas del sistema hexadecimal, el cual permite convertir de manera directa cadenas en binario a cadenas en hexadecimal, sin pasar por la interpretación de la cadena original. Para esto, la cadena binaria se segmenta formando cuartetos de bits comenzando por el bit menos significativo (b_0).
Supongamos por ejemplo la cadena `1001011010100101`, que al ser segmentada se obtiene:

`1001 0110 1010 0101`.

Tener en cuenta que cada cuarteto es alguna de las combinaciones de 4 bits del sistema $BSS(4)$ y por lo tanto cada valor se encontrará dentro del siguiente rango [0.,15]. Considerando que dichos valores se pueden representar por un solo caracter hexadecimal, entonces se aplica la tabla **??** para convertir, uno a uno, los cuartetos de la cadena. En el ejemplo mencionado:

```
1001   0110   1010   0101
 9      6      A      5
```

Por lo tanto, las cadenas `96A5` y `1001011010100101` representan el mismo valor. Notar que no hizo falta obtener ese valor, dado que no se aplicó el proceso de interpretación. Sin embargo, lo utilizaremos para comprobar que son equivalentes:

$$I_{hex(4)}(\texttt{96A5}) = 5 \times 16^0 + 10 \times 16^1 + 6 \times 16^2 + 9 \times 16^3 =$$

Esta expresión se puede reescribir expresando los digitos hexadecimales como suma de potencias de 2:

$$= (4+1) \times 16^0 + (8+2) \times 16^1 + (4+2) \times 16^2 + (8+1) \times 16^3$$

Aplicando la propiedad distributiva:

$$(4 \times 16^0 + 1 \times 16^0) + (8 \times 16^1 + 2 \times 16^1) + (4 \times 16^2 + 2 \times 16^2) + (8 \times 16^3 + 1 \times 16^3)$$

Reemplazando los valores por sus equivalentes potencias de 2:

$$(2^2 \times (2^4)^0 + (2^4)^0) + (2^3 \times (2^4)^1 + 2 \times (2^4)^1) + (2^2 \times (2^4)^2 + 2 \times (2^4)^2 + 2^3 \times (2^4)^3 + (2^4)^3$$

Simplificamos los términos aplicando las propiedades de potencia:

$$(2^2 \times 2^0) + (2^0) + (2^3 \times 2^4) + (2 \times 2^4) + (2^2 \times 2^8) + (2 \times 2^8) + (2^3 \times 2^{12}) + (2^{12})$$

Aplicando la propiedad de potencias de igual base:

$$= 2^2 + 2^0 + 2^7 + 2^5 + 2^{10} + 2^9 + 2^{15} + 2^{12}$$

Aplicamos la propiedad conmutativa para ordenar los términos según las potencias de manera ascendente:

$$2^0 + 2^2 + 2^5 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{15}$$

Cuya expresión, por definicion, equivale a la interpretación de la cadena `1001011010100101` (Notar que no se incluyen los bits en cero):

$$Ibss(16)(\texttt{1001011010100101}) = 2^0 + 2^2 + 2^5 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{15}$$

Capítulo 4

Sistemas de numeración para números enteros

En apartados anteriores se abordó el problema de representar los números naturales en el sistema binario, detallando los mecanismos de interpretación, representación, rango y operaciones aritméticas. En el presente capitulo se realizará un análisis similar para el caso de los números enteros, teniendo en cuenta que los sistemas se implementarán en un sistema de cómputos donde sólo pueden almacenarse y procesarse dos estados: cero y uno.

Por otro lado, el sistema decimal que habitualmente usan las personas para comunicarse, utiliza un símbolo especial "−"para indicar cuando un número es negativo. Sin embargo esto no es trasladable directamente a un sistema de cómputos pues no es representable en las computadoras como tal, donde solo se almacenan los dígitos (o estados) cero y uno.

En las siguientes secciones se presentan tres posibles enfoques para abordar la representación de números negativos sin contar con un símbolo especial, cada uno con sus ventajas y limitaciones.

4.1. Signo - Magnitud

La idea detrás de este sistema es cubrir la incapacidad de escribir el signo '−' en el contexto de una computadora, indicando de alguna forma si está presente o no. Dicho en otras palabras, indicar mediante un bit la polaridad del valor. Por convención se suele usar el bit del extremo izquierdo de la cadena como indicador y se lo denomina *bit de signo* . Usando el bit de signo en estado 1 se indica que el número es negativo, y usando el bit en estado 0 se indica que es positivo. Los bits restantes de la cadena reciben el nombre de *magnitud* y su valor se determina con el mecanismo de interpretación del sistema binario sin signo ($BSS()$).

Por lo anterior, este sistema recibe el nombre **Signo-Magnitud** (SM). Cuando se restringe la cantidad de bits a n, se lo denota **SM(n)**, donde el bit de la izquierda es el signo, y la magnitud es de $n - 1$ bits.

Interpretación de cadenas en SM(n)

La idea general de este mecanismo es reutilizar el mecanismo de interpretación del sistema Binario Sin Signo pero aplicándolo sólamente sobre los bits de la magnitud.

Por ejemplo, si se considera la cadena `1010`, el primer paso para interpretarla es separar el bit de signo (en este caso `1`) de la magnitud (en este caso: `010`). El bit de signo se interpreta según lo indicado antes: si es `1` se indica un valor negativo, y si es `0` un valor positivo. En este caso la cadena comienza con 1, de modo que se trata de un negativo. Por otro lado, la magnitud se interpreta como BSS:

$$I_{BSS(3)}(010) = 0 * 2^0 + 1 * 2^1 + 0 * 2^2 = 2$$

De esta manera, la interpretación en SM se puede expresar en términos de la función de interpretación de BSS:

$$I_{SM(4)}(1010) = -1 \times I_{BSS(3)}(010) =$$
$$= -1 \times (0 * 2^0 + 1 * 2^1 + 0 * 2^2) = -1 \times 2 = -2$$

Representación de cadenas en SM(n)

Del mismo modo, la representación en signo magnitud se apoya sobre el mecanismo de representación del sistema Binario Sin Signo. Sin embargo, la representación del sistema BSS no permite representar números negativos, por lo que, en esos casos, es necesario transformar el valor a representar antes de usar la función de representación en BSS. Esta adaptación se consigue tomando el valor absoluto del valor dado para representar.

Suponiendo como ejemplo que se se necesita representar el valor -5 en SM(4). Este sistema destina 1 bit para el signo y 3 bits para la magnitud. El primer paso a resolver es la codificación del signo, y como en este caso el número es negativo, entonces el valor del bit de signo que va a tener la cadena resultante es **1**. A continuación se toma el valor absoluto del número y se lo representa como en BSS de tres bits, obteniendo como resultado `101`.

La cadena final en SM(4) se obtiene componiendo ambas partes, es decir:

$$R_{SM(4)}(\text{-}5) = 1R_{bss(3)}(|-5|) = 1101$$

De manera general, este mecanismo puede expresarse como la siguiente función por partes:

$$R_{SM(n)}(x) = \begin{cases} R_{BSS(n)}(x) & si \quad x > 0 \\ \mathbf{1}R_{BSS(n-1)}(|x|) & si \quad x \leq 0 \end{cases}$$

Rango del sistema SM(n)

Como se definió en apartados anteriores, el rango es el intervalo de números representables en un sistema con una cierta cantidad de bits (n bits).

El número mínimo que es posible representar en un sistema de signo magnitud será en todos los casos negativo, es decir que la cadena que lo representa comienza con 1. Además, su magnitud es la mas grande posible. Suponer la situación de un sistema signo-magnitud de 3 bits. Entonces el rango tendría

como mínimo al valor representado por la cadena `111`, y como máximo al valor representado por la cadena `011`. interpretando ambas cadenas se obtiene:

$$min = I_{SM(3)}(111) = -1 \times I_{BSS(2)}(11) = -(2^0 + 2^1) = -3$$

$$max = I_{SM(3)}(011) = I_{BSS(2)}(11) = 2^0 + 2^1 = 3$$

Es decir que el rango del sistema SM(3) es $[-3, 3]$.

Generalizando esta idea a un sistema signo-magnitud de n bits, el mínimo se obtiene:

$$I_{SM(n)}(\underbrace{1}_{s}\underbrace{1\ldots1}_{n-1}) = -1 \times I_{BSS(n-1)}(1\ldots1) = -(2^0 + \cdots + 2^{n-2}) = -(2^{n-1} - 1)$$

Dualmente, el número máximo se obtiene utilizando signo positivo, es decir 0; y nuevamente la mayor magnitud.

$$I_{SM(n)}(\underbrace{0}_{s}\underbrace{1\ldots1}_{n-1}) = I_{BSS(n-1)}(1\ldots1) = (2^0 + \cdots + 2^{n-2} = 2^{n-1} - 1$$

Por lo tanto el rango del sistema SM de n bits es $[-(2^{n-1} - 1), 2^{n-1} - 1]$.

Es interesante notar que en el rango de un sistema SM no se representan 2^n números distintos, sino que se representan $2^n - 1$. Volviendo al ejemplo del sistema SM(3), cuyo rango es: $[-3, 3]$, es posible analizar que este contiene sólo 7 números (y no 8): $\{-3, -2, -1, 0, 1, 2, 3\}$. Si se considera el sistema BSS de 3 bits, en cuyo caso se podían representar 8 números diferentes, surge la pregunta: ¿Cuál es el número que falta? La situación que se genera es producto de que el sistema posee una *doble representación del 0* , ya que tanto la cadena `000` como la cadena `100` representan dicho valor.

Esto trae consigo dos desventajas: la primera es el hecho de desaprovechar una cadena, y la segunda es que esta doble representación hace mas compleja la automatización de la aritmética en el sistema de cómputos, como se verá en el capítulo 6, al tener que considerar dos cadenas que representan el mismo valor.

Una característica que posee el rango de este sistema (y otros) es que, a diferencia del BSS(), el rango en SM() es un **rango simétrico**. Esto significa que, partiendo desde el 0, se tienen n cantidad de números positivos y negativos. Tomando el ejemplo anterior de un sistema SM(3), al ver los números posibles a representar se ve que se pueden representar 3 números positivos [1, 2, 3] y 3 números negativos [-3, -2, -1]

4.1.1. Aritmética

Suma

La suma en SM analiza los diferentes casos en función de los signos de las cadenas a sumar. Si las cadenas a sumar tienen el mismo signo (ambas negativas o ambas positivas), la suma se realizará sumando las magnitudes con

el algoritmo de BSS y replicando el signo de los operandos (que es el mismo) en el signo del resultado. Si las cadenas a sumar tienen diferente signo, se debe primero identificar qué cadena tiene la mayor magnitud para replicar su signo en el resultado y computar la magnitud resultado como una resta.

Mas formalmente, sean A y B los operandos (cadenas en SM(n)). Entonces sean s_A y s_B los signos de A y B respectvamente, y sean m_A y m_B las magnitudes.

a. $s_A = s_B$ (igual signo)

- Signo del resultado: $s_R = s_A = s_B$
- Magnitud del resultado: $m_R = m_A + m_B$ (suma en BSS).

b. $s_A \neq s_B$ (diferente signo): Si ocurre que $I_{BSS}(m_A) \geq I_{BSS}(m_B)$

- Signo del resultado: $s_R = s_A$
- Magnitud del resultado: $m_R = m_A - m_B$ (resta en BSS).

Considerar como ejemplo la suma `1101+1001`. Dado que el signo es el mismo en ambos operandos (`1`) y la magnitud se obtiene al sumar en BSS es `101 + 001 = 110`, entonces la cadena resultado de la suma en SM es `1110`.

Considerar otro ejemplo: `1101 + 0001`. La mayor magnitud es la de la cadena `1101`, ya que su magnitud es `101` (y esta representa el número 5) y es mayor que la magnitud `001` (que representa un 1). Es por esta razón que se toma como signo del resultado: $s_R = 1$. Entonces para determinar la magnitud del resultado se realiza la resta (en binario sin signo) entre las magnitudes: `101 - 001 = 100`. Por lo tanto el resultado final es `1100`.

Es posible analizar la correctitud del algoritmo como se hizo en el capítulo 3: interpretando los operandos y el resultado.

Resta

El algoritmo para calcular la resta entre las cadenas A y B, puede pensarse como una suma, aplicando la siguiente equivalencia:

$$A - B = A + (-B)$$

Es decir que la resta se resuelve en 2 pasos: primeramente se calcula el inverso del sustraendo (-B) y luego este se suma al minuendo (A). En el sistema SM, es posible construir el inverso de una cadena B a partir de invertir su bit de signo. Luego de ese paso, la operación se completa como una suma en Binario Sin Signo y se deben analizar los casos que se describieron en el apartado anterior.

Considerar como ejemplo la resta `1001 - 0101`, entonces como primer paso se invierte el segundo operando obteniendo `1101` y luedo la suma `1001 + 1101` se resuelve considerando los signos adecuadamente, según el algoritmo de la sección anterior. Entonces, dado que en este caso tienen igual signo, el signo de resultado será también `1` y su magnitud será el resultado de la suma (en BSS) de `001+101`. Por lo tanto, la cadena resultante es `1110`.

Para confirmar que lo anterior es correcto, al igual que en los ejemplos desarrollados anteriormente, se interpretan en SM(4) ambos operandos y el resultado

Operando A	$I_{SM(4)}(1001) = -1 \times I_{BSS(3)}(001) = -1 \times 2^0 = -1 \times 1 = -1$
Operando B	$I_{SM(4)}(0101) = I_{BSS(3)}(101) = 2^0 + 2^2 = 1 + 4 = 5$
Resultado esperado	$a - b = -1 - 5 = -6$,
Resultado obtenido	$I_{SM(4)}(1110) = -1 \times I_{BSS(3)}(110) = -1 \times (2^1 + 2^2) = -1 \times (2 + 4) = -6$

Tabla 4.1: Validación de la resta (ejemplo 1)

Operando A	$I_{SM(4)}(1101) = -1 \times I_{BSS(3)}(101) = -1 \times (2^0 + 2^2) = -1 \times (1 + 4) = -1 \times 5 = -5$
Operando B	$I_{SM(4)}(1001) = -1 \times I_{BSS(3)}(001) = -1 \times (2^0) = -1$
Resultado esperado	$a - b = -5 - (-1) = -4$,
Resultado obtenido	$I_{SM(4)}(1100) = -1 \times I_{BSS(3)}(100) = -1 \times (2^2) = -4$

Tabla 4.2: Validación de la resta (ejemplo 2)

obtenido para comparar con el resultado esperado, como se ilustra en la tabla 4.1, por lo que puede concluirse que el mecanismo fue aplicado correctamente.

Considerar como segundo ejemplo la resta `1101 - 1001`, entonces en primer lugar se invierte el segundo operando obteniendo `0001` para luego plantear la suma `1101 + 0001`. Dado que los operandos tienen diferente signo y la magnitud que representa al valor mas grande es `101`, el signo de resultado será `1` y su magnitud será el resultado de las resta (en BSS) de `101-001`. Por lo tanto, la cadena resultante es `1100`.

Nuevamente, se comprueba mediante las interpretaciones y la comparación con el resultado esperado, según se explica en la tabla 4.2. Allí puede verse que el resultado concide con aquel que se esperaba.

4.2. Complemento a 2

Este sistema de numeración fue diseñado para ser automatizado en un sistema de cómputos, y asumiendo la existencia de una computadora que sea capaz de realizar operaciones en el sistema BSS, se buscó manejar negativos con el mismo dispositivo, de manera que se pueda mantener la aritmética del sistema BSS. Este objetivo es interesante porque permite que una maquina no necesite incorporar nuevos elementos para sumar números naturales y enteros, y esto se traduce en abaratar costos.

La idea que subyace este sistema puede introducirse con un sistema decimal de un sólo dígito (denotado *dec(1)*) que permita representar negativos (sin usar el signo "-"). Para resolverlo, se ubican las **cadenas** del sistema *dec(1)* en el exterior de una rueda como se indica en la figura 4.1 , y se le asignan **valores** en la parte interna de la rueda de manera que cada cadena (de un sólo dígito) quede aparejada con el valor que se corresponde a su complemento a la base. En este escenario la base es 10, por lo que la cadena `9` se corresponde con el valor -1, pues:

$$10 + (-1) = 9$$

Esta forma de relacionar cadenas y valores tiene la particularidad que preserva las propiedades de las operaciones aritméticas, pues la suma entre cadenas sigue el mismo mecanismo que en decimal, aunque las cadenas estén representando otros números y esa es una gran ventaja.

Por ejemplo, dado que el sistema *dec(1)* es un sistema decimal restringido a un dígito, la suma `8+2` da como resultado la cadena `0` (con acarreo de 1).

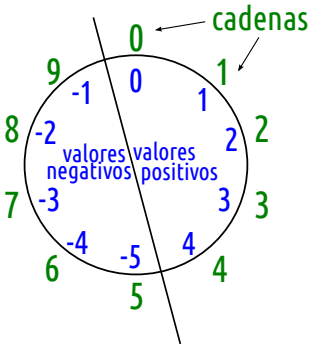


Figura 4.1: Correspondencia entre cadenas y valores en el sistema Dec(1)

Para validar el resultado de la cuenta, como en las situaciones anteriores, se deben interpretar las cadenas de los operandos y el resultado. En este escenario *dec(1)* se utiliza la correspondencia graficada en la Figura 4.1 para interpretar. Entonces se tiene que:

$$I(8) + I(2) = -2 + 2 = 0$$

Considerar un segundo ejemplo: `7+4`, donde la cadena resultado es `1` (con acarreo 1). Al validar esto se obtiene que

$$I(7) + I(4) = -3 + 4 = 1$$

Esta idea se puede trasladar al sistema binario, para lo cual deben pensarse las cadenas en un círculo ordenadas lexicográficamente, y luego asignarse números positivos en sentido horario y negativos en sentido anti-horario, como se muestra en la figura 4.2. De esta manera se consigue que algunas cadenas (las mas pequeñas en orden lexicográfico) queden asociadas a valores positivos y las otras a valores negativos (las mayores en orden lexicográfico). Este sistema es el *Complemento a 2*, y se denota $CA2(n)$.

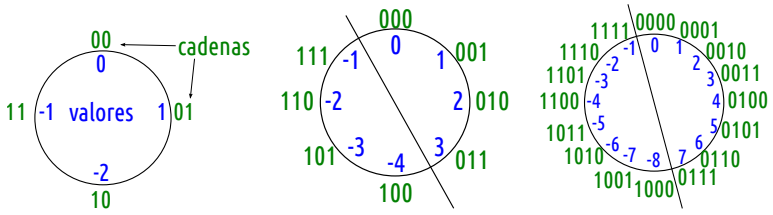


Figura 4.2: Correspondencia entre cadenas y valores en sistemas CA2

En este sistema también ocurre que las cadenas que empiezan en 0 se utilizan para representar los números positivos y las cadenas que comiencen en 1 se utilizan para los negativos. Es importante notar que no se destina un bit para signo como es el caso del sistema signo-magnitud, pues el bit mas significativo tiene peso a la hora de interpretar (como se explicará en la siguiente sección).

Previamente a definir los mecanismos de interpretación y representación, es necesario definir la operación *complemento()* sobre una cadena. El complemento

`complemento()`

de una cadena w es otra cadena w' que se obtiene a partir de invertir los bits de w y sumarle 1 al resultado. Por ejemplo, el complemento a 2 de la cadena `0001` es `1111` pues:

$$complemento(0001) = 0001 \rightarrow 1110 + 1 \rightarrow 1111$$

La operación *complemento* cumple con la siguiente propiedad:

$$w + w' = \underbrace{0 \ldots 0}_{n-1}$$

es decir, que al sumar una cadena y su complemento se obtiene como resultado la cadena de n ceros. Por ejemplo, con 4 bits: $0001 + 1111 = 0000$. Esto es importante porque justifica la asociación entre cadenas y valores como se ve en la Figura 4.2. En particular:

$$I(w) + I(w') = I(0000) = 0$$

Interpretación

Si bien tanto la interpretación y la representación pueden resolverse a través de una tabla que relaciona cada cadena con un valor, este enfoque enumerativo resulta impracticable cuando se tiene una cantidad grande de bits. Además, considerando que estos sistemas se deben automatizar en un sistema de cómputos, se busca operacionalizar el mecanismo de ambas funciones.

Como se anticipó, la interpretación de cadenas en CA2 debe considerar dos casos, determinados según cómo comienza la cadena. Si comienza con 0 (esto se denota $b_{n-1} = 0$), entonces se trata de un valor positivo, y en ese caso simplemente se interpreta como en un sistema $BSS(n)$. En caso contrario, si $b_{n-1} = 1$, se sabe que representa un valor negativo, en cuyo caso se aplica la operación *complemento*() a la cadena, para luego interpretar el resultado en $BSS(n)$, y finalmente agregarle el signo negativo. Esto se formaliza en la siguiente definición de la función $I_{CA2}()$:

$$I_{CA2(n)}(c) = \begin{cases} I_{BSS(n)}(c) & si \quad b_{n-1} = 0 \\ (-1) \times I_{BSS(n)}(complemento(c)) & si \quad b_{n-1} = 1 \end{cases}$$

Suponer el ejemplo de interpretación de la cadena `1111`. Dado que su primer bit (b_{n-1}) es 1 se entiende que representa un número negativo. Entonces su interpretación es como sigue:

$$I_{CA2(4)}(1111) = (-1) \times I_{BSS(4)}(complemento(1111))$$

Esto se resuelve en los siguientes pasos:

1. Se calcula su complemento a dos: $complemento(1111) = 0000 + 1 = 0001$, y reemplazando en la expresión anterior se obtiene:

$$(-1) \times I_{BSS(4)}(0001)$$

2. Se interpreta la cadena `0001` en BSS, y se obtiene como resultado final:

$$(-1) \times (1 \times 2^0) = -1$$

Representación

Respetando la dualidad entre las funciones de interpretación y representación, en esta sección se describe el proceso de representación distingue dos casos según si el valor es positivo (o cero), o si es negativo. Los números positivos se representan tal como se hace en binario sin signo y los números negativos se representan aplicando el **complemento a dos de la cadena** que representa su **valor absoluto**, como se ve en la siguiente definición de la función $R_{CA2}()$:

$$R_{CA2(n)}(x) = \begin{cases} R_{BSS(n)}(x) & si \quad x > 0 \\ complemento(R_{BSS(n)}(|x|)) & si \quad x \leq 0 \end{cases}$$

Por ejemplo, al representar el valor 3 en CA2(4), como este es positivo, se aplica directamente el mecanismo de BSS:

$$R_{BSS(4)}(3) = 0011$$

Por el contrario, al representar un valor negativo como -2, primero se debe representar el valor 2 en binario sin signo (es decir `0010`) y a continuación se toma el complemento a 2 de dicha cadena `0010` $\rightarrow$ `1101 + 1` $\rightarrow$ `1110`. Por lo tanto:

$$\begin{aligned} R_{CA2(4)}(-2) &= complemento(R_{BSS(4)}(|-2|)) \\ &= complemento(0010) = 1101 + 1 = 1110 \end{aligned}$$

Una vez mas, la representación se valida con la interpretación de la cadena resultante, como sigue:

$$\begin{aligned} I_{ca2(4)}(1110) &= -1 \times I_{BSS(4)}(complemento(1110)) \\ &= -1 \times I_{BSS(4)}(0010) = -1 \times 2 = -2 \end{aligned}$$

En lo anterior se puede apreciar que se obtuvo el valor -2, que era el valor original de la representación y por lo tanto se comprueba que el mecanismo fue aplicado correctamente.

Rango

Nuevamente, el cálculo del rango de un sistema de numeración se traduce en interpretar las cadenas que representan al máximo y mínimo valor. En otros sistemas se tomaron las cadenas en orden lexicográfico, pero en este caso se debe otro orden, que es el que se describe en la Figura 4.2. Es posible observar que las cadenas que representan el máximo y el mínimo siempre se ubican en el extremo inferior de la circunferencia:

- $ca2(2)$: cadena del mínimo $c_{min} = 10$ y del máximo $c_{max} = 01$
- $ca2(3)$: cadena del mínimo $c_{min} = 100$ y del máximo $c_{max} = 011$
- $ca2(4)$: cadena del mínimo $c_{min} = 1000$ y del máximo $c_{max} = 0111$

De lo anterior es posible detectar un patrón en cuanto a la estructura de las cadenas: las cadenas del valor máximo siempre comienzan con un dígito 0 y están seguidas por una cadena de unos, mientras que las cadenas del valor

Sistema	Mínimo	Máximo
CA2(2)	$I_{ca2(2)}(10)$ $= -1 \times I_{bss(2)}(complemento(10))$ $= -1 \times I_{bss(2)}(10) = -1 \times (2^1) = -2$	$I_{ca2(2)}(01) = 2^0 = 1$
CA2(3)	$I_{ca2(3)}(100)$ $= -1 \times I_{bss(2)}(complemento(100))$ $= -1 \times I_{bss(2)}(100) = -1 \times (2^2) = -4$	$I_{ca2(2)}(011) = 2^0 + 2^1 = 3$
CA2(4)	$I_{ca2(3)}(1000)$ $= -1 \times I_{bss(2)}(complemento(1000))$ $= -1 \times I_{bss(2)}(1000) = -1 \times (2^3) = -8$	$I_{ca2(2)}(0111) = 2^0 + 2^1 + 2^2 = 7$

Tabla 4.3: Rangos de los sistemas CA2

mínimo tienen un dígito 1 seguido de una cadena de ceros. La interpretación de estas cadenas se decribe en la tabla 4.3.

De manera general, el cómputo del rango puede expresarse como sigue:

$$[-2^{n-1}, 2^{n-1} - 1]$$

Es interesante notar que en este caso, la cantidad de números representables si es de 2^n, ya que no hay doble representación de ningún número.

Aritmética

La aritmética en CA2, por definición de complemento a la base, cumple con la propiedad de ser mecánicamente idéntica a la aritmética del sistema BSS. Es decir que tanto la suma como la resta se resuelven con los mismos algoritmos que se presentaron en el Capítulo 3. Para mostrar la validez de esta afirmación, es necesario comprobar la correctitud de la suma aplicando la interpretación en CA2 tal como se hizo anteriormente.

Suponer la siguiente operación de suma:

$$\begin{array}{r} 001 \\ +100 \\ \hline 101 \end{array}$$

Para comprobar si es válido en el contexto de un sistema Complemento a 2 se debe cumplir la siguiente propiedad:

$$I_{ca2}(001) + I_{ca2}(100) = I_{ca2}(101)$$

Por un lado se tiene que el valor de los operandos es:

$$I_{ca2}(001) = I_{bss}(001) = 1$$

$$I_{ca2}(100) = -1 \times I_{bss}(011 + 1) = -1 \times I_{bss}(100) = -1 \times 4 = -4$$

También se sabe que el resultado vale:

$$I_{ca2}(101) = -1 \times I_{bss}(010 + 1) = -1 \times I_{bss}(011) = -1 \times 3 = -3$$

Por lo tanto:

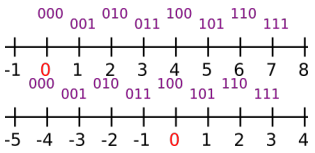


Figura 4.3: Comparación entre sistemas BSS(3) y Exc(3,4)

$$\begin{array}{r} I_{ca2}(001) = 1 \\ +I_{ca2}(100) = -4 \\ \hline I_{ca2}(101) = -3 \end{array}$$

Lo anterior demuestra que la operación de suma en BSS es válida también para el sistema CA2. De manera similar, la tabla 4.4 valida la siguiente operación de resta:

$$\begin{array}{r} 100 \\ -101 \\ \hline 111 \end{array}$$

Operando A	$I_{CA2(3)}(100) = -1 \times I_{BSS(3)}(complemento(100)) = -1 \times I_{BSS(3)}(100)$ $= -1 \times 2^2 = -4$
Operando B	$I_{CA2(3)}(101) = -1 \times I_{BSS(3)}(complemento(101)) = -1 \times I_{BSS(3)}(011)$ $= -1 \times (2^0 + 2^1) = -3$
Resultado esperado	$a - b = -4 - (-3) = -1$,
Resultado obtenido	$I_{CA2(3)}(111) = -1 \times I_{BSS(3)}(complemento(111)) = -1 \times I_{BSS(3)}(001)$ $= -1 \times (2^0) = -1$

Tabla 4.4: Validación de la resta (CA2(4))

4.3. Exceso

El sistema de exceso trabaja utilizando las reglas de interpretación y representación de BSS pero desplazando todas las interpretaciones sobre la recta numérica. Para trabajar en un sistema de exceso, se necesita conocer, además de la cantidad de bits n, el valor del desplazamiento Δ, denotándose $EX(n, \Delta)$. Considerar por ejemplo un sistema de $\Delta = 4$ con 3 bits; entonces las cadenas que en un escenario sin signo representan los valores [0, 1, 2, 3, 4, 5, 6, 7], al desaplazarse sobre la recta numérica 4 lugares hacia los negativos, representan los valores [-4, -3, -2, -1, 0, 1, 2, 3], es decir, cada cadena representará un valor a distancia 4 del valor que representa en BSS.

Exceso, o Δ, representa un *desplazamiento* relativo de las cadenas

La idea que atraviesa el sistema exceso es la de desplazar las cadenas sobre la recta numérica, con respecto a un sistema de base que es el BSS, y darnos la posibilidad de tener un rango no equilibrado con respecto al cero, a diferencia que en Signo-Magnitud y Complemento a 2, que poseen un rango equilibrado con respecto al 0.

Observando la figura 4.3 es posible comparar el sistema Excedido en 4 de 3 bits con el sistema BSS(3) y observar el espíritu de los sistemas excedidos.

Se puede observar al tomar cualquier cadena del sistema se cumple la siguiente propiedad: $I_{bss}(cadena) - 4 = I_{exc}(cadena)$

De manera aún mas general:

$$I_{bss}(cadena) - \Delta = I_{exc}(cadena)$$

Representación

Considerando la equivalencia mencionada entre ambos sistemas de numeración, la tarea de representar un valor se apoyará en los mecanismos conocidos del sistema binario sin signo. Para esto es necesario que los valores negativos sean desplazados hacia el sector positivo: es decir que al momento de representar un valor, se le sumará el valor Δ y luego representar *ese valor positivo* en BSS.

Por ejemplo, si el sistema es $Ex(4, 8)$ (es decir: Exceso de 4 bits con 8 de desplazamiento), y se quiere representar el valor -2, dado que este es negativo se debe primero convertir en un valor positivo sumándole el exceso, es decir 8: $-2 + 8 = 6$, para luego representar al 6 en BSS de 4 bits obteniendo la cadena `0110`

Entonces, la representación en exceso se puede formalizar como:

$$R_{exc}(x) = R_{bss}(x + \Delta)$$

Interpretación

Dualmente, al momento de interpretar se debe aplicar de manera opuesta la mencionada equivalencia entre sistemas. El primer paso será entonces usar las reglas de binario sin signo para interpretar la cadena y una vez que obtenido el valor numérico de la cadena, se le quita el exceso para obtener el verdadero número excedido (representado en la cadena).

Por ejemplo, si se trabaja en EX(4,8) (es decir Exceso de 4 bits con 8 de desplazamiento), y se necesita interpretar la cadena 0111, en primer lugar se la interpreta en binario sin signo, obteniendo el valor 7, y luego se le resta el desplazamiento: $7 - 8 = -1$. Por lo tanto $I_{exc}(0111) = -1$

Entonces, la interpretación en exceso se puede formalizar como:

$$I_{exc}(cadena) = I_{bss}(cadena) - \Delta$$

Rango

Es importante notar, habiendo comparado las rectas de la figura 4.3, que las cadenas en los sistemas excedidos respetan el mismo orden que las cadenas en el sistema binario sin signo. Por lo tanto, en un contexto de un sistema exceso de n bits, las cadenas que representan a los números mínimo y máximo son las mismas que en BSS: $\underbrace{00..00}_{n}$ y $\underbrace{11..11}_{n}$. Los valores que representan cada una dependen del valor del exceso Δ.

El mínimo es $I_{exc}(00..0) = I_{bss}(00..0) - \Delta = 0 - \Delta = -\Delta$, mientras que el máximo es $I_{exc}(11..1) = I_{bss}(11..1) - \Delta = 2^n - 1 - \Delta$. Por lo tanto, el rango es

$$[-\Delta, 2^n - 1 - \Delta]$$

Si bien el sistema de exceso permite tener rangos no equilibrados con respecto al valor cero, esto no significa que no pueda obtenerse un equilibrio en los mismos. Para este fin se le debe definir el valor del desplazamiento Δ como 2^{n-1}.

Por ejemplo, si se tiene un sistema de exceso con 4 bits, para equilibrar su rango con respecto al 0 deberemos tomar como desplazamiento el valor de 2^{4-1}, o sea, $2^3 = 8$, de modo tal que el sistema quedaría como un sistema $Ex(4, 8)$.

Aritmética

Suma

Este mecanismo, como en los anteriores, intentará reutilizar los métodos y algoritmos del sistema binario sin signo para así también economizar en el diseño de la computadora, como se retomará en los siguientes capítulos.

Si se suman dos cadenas excedidas tal como se hacía en BSS, y luego se interpreta el resultado, se obtiene un valor *demasiado excedido* (o desplazado en el sentido que corresponda a Δ) y por lo tanto se necesita **anular un desplazamiento**.

Suponer las cadenas C_A y C_B que representan en exceso los valores a y b respectivamente. Es decir que a ambos valores se le sumó Δ para ser representados. Si ambas cadenas se suman mediante el algoritmo de BSS se tiene lo que se ve en la tabla 4.5, y en la interpretación de la cadena C_R se ve claramente que el valor está muy desplazado ($2 \times \Delta$).

$$\begin{array}{cll} & C_A & I_{bss}(C_A) = (a + \Delta) \\ + & C_B & I_{bss}(C_B) = (b + \Delta) \\ \hline & C_R & I_{bss}(C_R) = (a + \Delta + b + \Delta) = (a + b + 2 \times \Delta) \end{array}$$

Tabla 4.5: Suma en exceso: abordaje incorrecto

Esto se corrige agregando un cómputo de resta para eliminar el Δ de más, como se ve en la tabla 4.6.

$$\begin{array}{cll} & C_A & I_{bss}(C_A) = (a + \Delta) \\ + & C_B & I_{bss}(C_B) = (b + \Delta) \\ \hline & C_R & I_{bss}(C_R) = (a + \Delta + b + \Delta) = (a + b + 2 \times \Delta) \\ - & \Delta & \\ \hline & C'_R & I_{bss}(C'_R) = (a + b + \Delta) \end{array}$$

Tabla 4.6: Suma en exceso: abordaje correcto

Suponer, por ejemplo, que se quiere sumar las cadenas 1001 y 0011 en un sistema Exceso(4, 10), y al sumar como en bss se tiene: $1001 + 0011 = 1100$

Al interpretar los operandos y el resultado obtenido en exceso se tiene:

$$I_{exc}(1001) = I_{bss}(1001) - 10 = 9 - 10 = -1$$

$$I_{exc}(0011) = I_{bss}(0011) - 10 = 3 - 10 = -7$$

$$I_{exc}(1100) = I_{bss}(1100) - 10 = 12 - 10 = 2$$

Es posible observar que el resultado obtenido no es el esperado (se esperaba obtener -8 y se obtuvo 2). La distancia a la que se encuentra el resultado obtenido del esperado es, en efecto, el valor de Δ (en este ejemplo: 10).

$$-8 = 2 - 10$$

Es decir, la cadena se encuentra *demasiado excedida*, se encuentra más a la derecha de lo que debería (dicho de otra manera, tiene un Δ adicional). Para solucionarlo, hace falta ajustar su desplazamiento, restando el Δ a la cadena que se obtuvo:

$$\begin{array}{rr} & 1100 \\ - & 1010 \\ \hline & 0010 \end{array}$$

Si se interpreta en el sistema Exceso(4,10) esta última cadena, se obtiene:

$$I_{exc}(0010) = I_{bss}(0010) - 10 = 2 - 10 = -8$$

En resumen, la suma en exceso consiste en:

1. Resolver la suma en BSS
2. Ajustar el desplazamiento (restando Δ)

Resta

Similarmente al caso de la suma, con el espíritu de economizar recursos y conocimiento, se intentará reutilizar las herramientas del sistema Binario Sin Signo. Si se analiza el resultado obtenido de una resta en comparación con el resultado esperado se puede concluir un mecanismo de ajuste tal como se hizo con la suma.

Suponer como ejemplo que se quiere resolver la siguiente resta en un sistema Exceso(8, 32):

$$\begin{array}{rr} & 10000100 \\ - & 01111011 \\ \hline & 00001001 \end{array}$$

Al interpretar las cadenas de los operandos se tiene que:

$$I_{exc}(10000100) = I_{bss}(10000100) - 32 = 132 - 32 = 100$$

$$I_{exc}(01111011) = I_{bss}(01111011) - 32 = 123 - 32 = 91$$

Al interpretar el resultado:

$$I_{exc}(00001001) = I_{bss}(00001001) - 32 = 9 - 32 = -23$$

Al igual que en la suma, es posible observar que el resultado obtenido (-23) no es el esperado (9), y nuevamente la distancia entre ambos valores es equivalente a Δ.

$$-23 = 9 - 32$$

También en este caso la cadena se encuentra *demasiado excedida*, es decir, se encuentra más a la izquierda de lo que debería. Para solucionarlo se debe anular (o revertir) un desplazamiento mediante la suma de Δ:

$$\begin{array}{rr} & 00001001 \\ + & 00100000 \\ \hline & 00101001 \end{array}$$

Al interpretar esta última cadena se obtiene:

$$I_{exc}(00101001) = I_{bss}(00101001) - 32 = 41 - 32 = 9$$

En resumen, la resta en exceso consiste en:

1. Restar las cadenas en BSS
2. Anular un desplazamiento (sumando Δ)

Capítulo 5

Lógica Digital

En este capítulo se trabajará sobre los conceptos de lógica digital, necesarios para entender el hardware real de la computadora. Esto es posible, en parte, gracias al enfoque de Von Neumann (ver Sección 6.1), quien propuso separar el hardware del software, motivando así a que la computadora se dividiera en varios niveles de abstracción. El software ya se explicó capítulos anteriores, mientras que en el presente capítulo nos enfocaremos en el hardware. Siendo que el mismo conforma el nivel más bajo de abstracción, se encuentra en la frontera entre la ciencias de la computación y la ingeniería electrónica.

Los elementos básicos con que se construyen todas las computadoras son los circuitos digitales y suelen ser realmente sencillos, por lo cual, iremos examinando dichos elementos que, a modo de análisis, se relacionan con el álgebra booleana. Luego, presentaremos un mecanismo para construir circuitos a partir de compuertas en combinaciones sencillas. Finalmente, se presentarán circuitos clásicos que se encuentran en la mayoría de las unidades de control y unidades aritmético-lógicas.

Pero en concreto, **¿a qué se le llama circuitos digitales?** Los *circuitos digitales* son dispositivos físicos que implementan una función lógica. Es decir que para un conjunto de valores lógicos de entrada, computa una única salida lógica.

En este escrito se estudian los circuitos digitales denominados combinacionales o combinatorios. La característica principal que radica en esta clasificación se debe a que es un circuito que recibe varias entradas y su salida está determinada de forma única por las entradas vigentes.

5.1. Diseño de circuitos

En esta sección se describe una metodología para el diseño de circuitos digitales a partir de un documento que especifique los requerimientos del circuito. En primer lugar, se deben analizar los requerimientos para describir mediante una **caja negra** la interfaz del circuito, en términos de sus entradas y salidas. A continuación se deben analizar los posibles casos que pueden encontrarse en las entradas, con el objetivo de elaborar una **tabla de verdad**. Esta tabla

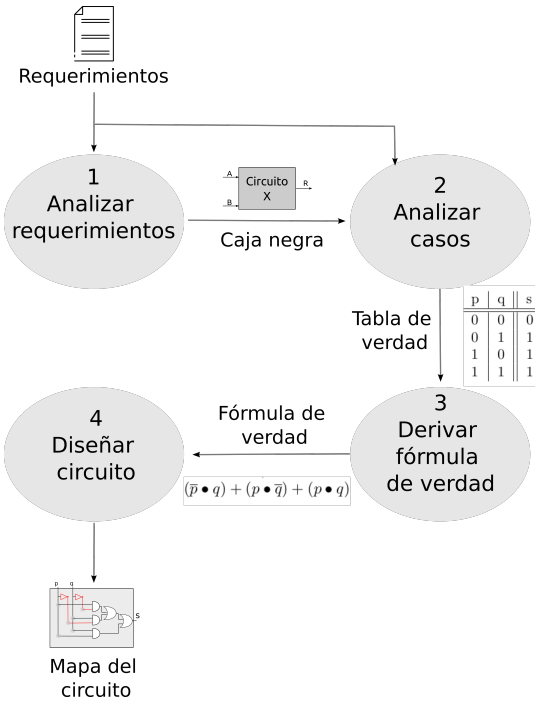


Figura 5.1: Diseño de un circuito digital

de verdad puede derivarse en una **fórmula de verdad**, que expresa la misma situación en términos de la lógica proposicional, a través de formas normales *suma de productos* o *productos de sumas*. Finalmente, basándose en la cooperación entre la lógica proposicional y la lógica de Boole, esa fórmula de verdad se traslada a un mapa de conexiones entre las entradas mediante **compuertas lógicas** para implementar esa fórmula.

Estos pasos se representan gráficamente en la figura 5.1, y en las siguientes secciones se explicará de manera más detallada cada uno de estos pasos.

5.1.1. Descripción de la interfaz del circuito

El primer paso del método propuesto se trata de formalizar la interfaz del circuito, que se debe expresar en términos del conjunto de entradas y el conjunto de salidas, detallando para cada una el nombre y tipo, es decir, que significado tienen. Los nombres de las entradas son los que se utilizan a la hora de realizar los siguientes pasos de este método. Estos conjuntos se describen gráficamente en un diagrama de *caja negra* , como se ejemplifica a continuación.

caja negra

Ejemplo: circuito *mayoría*, paso 1

Para ejemplificar la aplicación de estos conceptos en el diseño de un circuito concreto, se toma el siguiente requerimiento:

> *Se necesita un circuito de 3 entradas y una salida que compute la función "mayoría": si dos o más entradas valen 1, la salida debe*

Figura 5.2: Interfaz del circuito mayoría (caja negra)

A	B	C	Mayoría
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabla 5.1: Tabla de verdad para el circuito **mayoría**

valer 1, y un 0 en caso contrario.

Para entender lo que se pide, será útil dibujarlo como una caja negra, tal como muestra la Figura 5.2, donde pueden verse tres entradas independientes denominadas A, B y C, cada una de un bit, y una única salida denominada S.

5.1.2. Formalización de los casos: Tabla de verdad

El estudio de la lógica proposicional se apoya sobre un recurso formal denominado *tabla de verdad* cuyo objetivo es describir una función de verdad de manera exhaustiva, es decir: enumerando todas las posibles combinaciones entre las proposiciones de entrada y estableciendo el valor "de salida" para cada combinación. De ahora en más, se describirá el valor Falso como 0, y el valor verdadero como 1.

En el contexto del diseño de circuitos digitales, se considera la caja negra para plantear la estructura de la tabla y se revisan nuevamente los requerimientos para completarla, caso por caso.

Circuito *mayoría*, paso 2

Considerando el requerimiento, se tiene una tabla de verdad con 4 columnas (una por cada una de las 3 entradas y otra de salida), con 8 filas que representan los 8 posibles casos (pues $2^3 = 8$ siendo 3 la cantidad de entradas). Por ejemplo, si se analiza el caso donde A vale 1 y las demás entradas valen 0, entonces la salida debe ser 0 (pues no se cumple la definición planteada en el enunciado). Como otro ejemplo, en el caso en que B y C valen 1 pero A vale 0, la salida debe ser 1, porque se tienen 2 entradas en 1. Este análisis se realiza sobre todos los casos, dando lugar a la tabla de verdad que se muestra en el cuadro 5.1.

5.1.3. Fórmula de verdad

Para llevar a cabo el siguiente paso del método que se ilustró en la Figura 5.1, se necesita un mecanismo estructurado que permita deducir la fórmula de

verdad correspondiente a partir de la tabla de verdad construida anteriormente.

Para ello se cuenta con dos posibilidades:

1. Suma de productos (SoP)
2. Producto de sumas (PoS)

SoP El método SOP (*Sum of Product*) como lo dice su nombre, es una suma (disyunción) de términos que son productos (conjunciones) entre *literales*, es decir, entre una variable o su negación.

Literal

Para construir esta expresión, se debe extraer el término que describe cada caso de la tabla de verdad que verifica la fórmula, es decir, cada fila donde la salida vale 1.

Con cada uno de esos casos se describe un término con todas las variables, según aparezcan afirmadas o negadas. Por ejemplo:

p	q	s	
0	0	0	
0	1	1	$\rightarrow \overline{p} \bullet q$
1	0	1	$\rightarrow p \bullet \overline{q}$
1	1	1	$\rightarrow p \bullet q$

Finalmente, en este caso, los términos se componen con una disyunción: $(\overline{p} \bullet q) + (p \bullet \overline{q}) + (p \bullet q)$

PoS El método PoS (*Products of sum*) es una conjunción de disyunciones. A diferencia de la expresión anterior, describe la fórmula a partir de los casos donde no se cumple la proposición, con la siguiente idea: **"f vale cuando no ocurre el caso ... ni el caso..."**.

Por lo tanto en esta expresión, se deben tomar los casos donde la formula vale **0**, por ejemplo:

p	q	s	
0	0	0	$\rightarrow \overline{p} \bullet \overline{q}$
0	1	1	
1	0	0	$\rightarrow p \bullet \overline{q}$
1	1	1	

Luego, esos casos se niegan y se unen con conjunción, ya que no debe cumplirse ninguno de ellos: $\overline{(\overline{p} \bullet \overline{q})} \bullet \overline{(p \bullet \overline{q})}$

Finalmente, se pueden aplicar las leyes de Morgan, donde los términos negados se convierten en disyunciones: $(p + q) \bullet (\overline{p} + q)$

Circuito *mayoría*, paso 3

Considerando la tabla de verdad construida para el circuito "mayoría" en el paso anterior, que se ilustra en el cuadro el Cuadro 5.1, es posible llevar adelante el siguiente paso de la metodología propuesta. Entonces se traduce la tabla de verdad a la fórmula de verdad utilizando, en este caso, el método **SoP**, de la siguiente manera.

En primer lugar se describen los casos donde la salida vale 1, como se ve en el cuadro 5.2. En segundo lugar se compone la fórmula mediante la disyunción de los mencionados términos, obteniéndose la siguiente fórmula:

$$(\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$

A	B	C	Mayoría
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1 $(\neg A \bullet B \bullet C)$
1	0	0	0
1	0	1	1 $(A \bullet \neg B \bullet C)$
1	1	0	1 $(A \bullet B \bullet \neg C)$
1	1	1	1 $(A \bullet B \bullet C)$

Tabla 5.2: Casos positivos del circuito **mayoría**

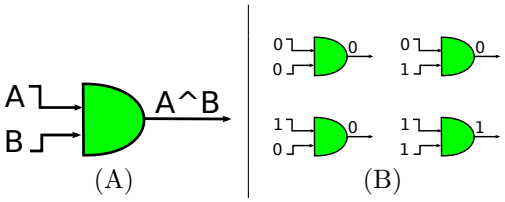


Figura 5.3: Compuerta AND

5.2. Mapa del circuito

compuertas lógicas

Para cumplir con el último paso se necesitará contar con los dispositivos atómicos que componen un circuito, que se denominan *compuertas lógicas* . Entonces, profundizando el concepto de circuito lógico mencionado al inicio del capítulo, se define a un circuito como: *una composición de compuertas que traduce un conjunto de entradas en una salida de acuerdo a una función lógica.* Esta salida se actualiza inmediatamente luego de proveerse las entradas.

Existe un medio gráfico para representar dispositivos (electrónicos, hidráulicos, mecánicos, etc.) que lleven a cabo funciones booleanas y que, en función de la combinación o combinaciones diseñadas, se obtendrán funciones más complejas.

5.2.1. Compuertas lógicas elementales

Las compuertas lógicas son dispositivos electrónicos que desarrollan las funciones lógicas elementales de conjunción, disyunción y negación.

Una compuerta AND implementa la función lógica de la conjunción tiene dos entradas y el resultado es un **1** sólo si ambas entradas son **1**. La notación gráfica se observa en la figura 5.3 (A) y los casos posibles se describen en la

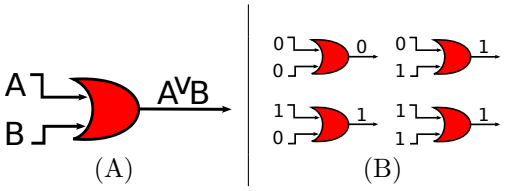


Figura 5.4: Compuerta OR

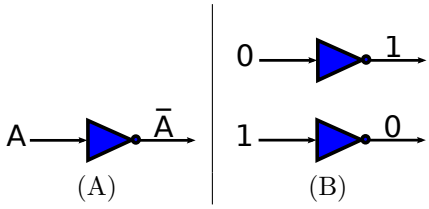


Figura 5.5: Compuerta NOT

figura 5.3 (B).

La compuerta OR es la que implementa una disyunción, y al igual que la compuerta anterior posee dos entradas. La salida de esta compuerta representa la operación lógica de una *suma* entre ambas entradas, es decir, basta que una de ellas sea 1 para que su salida sea también 1. La notación gráfica se observa en la figura 5.4 (A) y los casos posibles se describen en la figura 5.4 (B).

La compuerta NOT implementa un inversor, es decir, invierte el dato de entrada. Por ejemplo; si su entrada es 1 (nivel alto) se obtiene en su salida un 0 (o nivel bajo), y viceversa. Esta compuerta dispone de una sola entrada. La notación gráfica se observa en la figura 5.5 (A) y los casos posibles se describen en la figura 5.5 (B).

5.2.2. Compuertas lógicas adicionales

Además de las compuertas elementales para la conjunción, disyunción y negación, es común encontrar en la bibliografía otras compuertas que se describen en esta sección.

1. La compuerta **NAND** implementa la siguiente expresión lógica:

   $$a \uparrow b = \overline{a \bullet b}$$

   Esta expresión es una fórmula de verdad (que se apoya en el uso de una negación y una conjunción) que se sintetiza con el operador $\uparrow$. En su representación gráfica se reemplaza la compuerta NOT por un círculo a la salida de la compuerta AND. La notación gráfica se observa en la figura 5.6

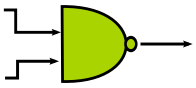


Figura 5.6: Compuerta NAND

2. La compuerta **NOR** es la dualidad de la anterior, implementando la expresión:

   $$a \downarrow b = \overline{a + b}$$

   También en este caso se introduce el operador $\downarrow$ que sintetiza la fórmula de verdad anterior. Como en el caso del NAND, se agrega un círculo a la compuerta OR y para representar la negación (ver Figura 5.7)

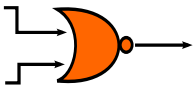


Figura 5.7: Compuerta NOR

3. La compuerta **XOR** implementa la expresión lógica:

$$a \oplus b = (\overline{a} \bullet b) + (a \bullet \overline{b})$$

Es una disyunción exclusiva, es decir que su salida será 1 si **una y sólo una** de sus entradas es 1. La notación gráfica se observa en la figura 5.8

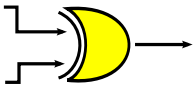


Figura 5.8: Compuerta XOR

5.2.3. Conexión de compuertas

La conexión de las compuertas puede dar lugar a un circuito poco legible si no se grafican las conexiones de manera ordenada. En este apartado se propone una técnica para ir conectando las compuertas siguiendo un orden espacial dentro del circuito de manera que quede organizado visualmente.

Como primer paso se debe replicar cada entrada del circuito de manera de disponer un cable de conexión para cada entrada y su negación, como puede observarse en la Figura 5.9 (A). Cada uno de esos cables representan los **literales de las fórmulas de verdad** y están disponibles para conectarse una o mas veces a las compuertas de cada **término**

El segundo paso es ir conectando a cada compuerta siguiendo el orden de los términos de la fórmula de verdad, en dirección a la salida del circuito (ver Figura 5.9 (B)). Como ejemplo, en la Figura 5.9 (C) se puede ver cómo conectar la línea e_1 y la línea $\overline{e_2}$ a una compuerta AND, siguiendo un término $(e_1 \bullet \overline{e_2})$. A continuación se conecta otra compuerta AND según un posible segundo término $(e_1 \bullet e_2)$, como se ve en la Figura 5.9 (D).

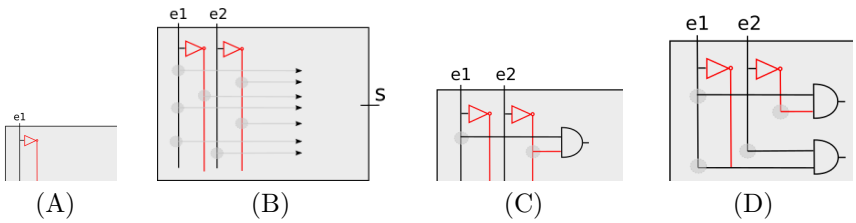


Figura 5.9: Conexión de entradas y compuertas

Finalmente, las salidas de cada compuerta AND (que representan los términos) se conectan entre si mediante compuertas OR, como puede verse en la Figura 5.10 que ilustra el circuito que implementa la función XOR, siguiendo la fórmula de verdad $(a \bullet \overline{b}) + (\overline{a} \bullet b)$.

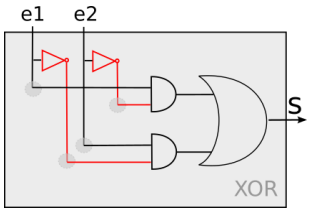


Figura 5.10: Circuito que implementa la función XOR

Diseño del circuito *mayoría*, paso 4

El último paso del método de diseño de circuitos consiste en conectar adecuadamente las compuertas lógicas que sean necesarias, respetando la función de verdad correspondiente.

Recuperando la fórmula de verdad construida en el paso anterior se tiene:

$$(\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$

El mapa de este circuito se completa siguiendo la precedencia que plantea la fórmula: en primer lugar cada término se representa con la conexión de 3 literales mediante 2 compuertas de tipo **and**. Luego las líneas que representan estos términos se componen mediante 3 compuertas de tipo **or**. Esto se describe en la figura 5.11.

5.3. Composición de circuitos

Hasta este punto se ha explicado el diseño de circuitos combinatorios a partir de compuertas elementales, pero el diseño de circuitos que resuelven problemas

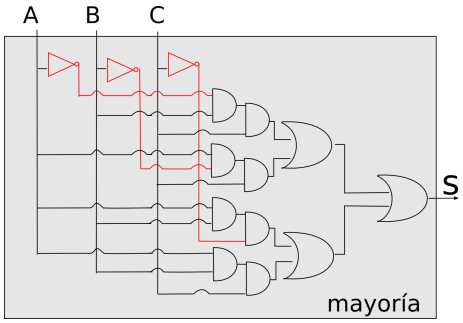
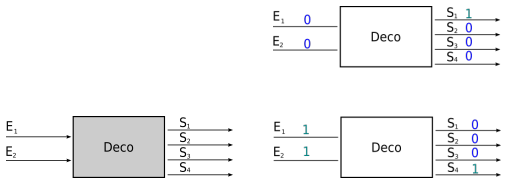
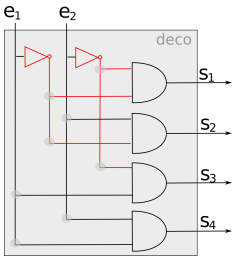


Figura 5.11: Mapa del circuito *Mayoría*



(A) caja negra (B) Ejemplos

Figura 5.12: Decodificador de N=2



e_1	e_2	s_1	s_2	s_3	s_4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(A) Tabla de verdad (B) Mapa

Figura 5.13: Decodificador de N=2

más complejos requieren un abordaje mediante descomposición. Es decir, pensar este problema como una composición de problemas, donde cada problema mas simple se resuelve en si mismo con un sub-circuito, que puede haber sido previamente diseñado como parte de otro problema. Para ello, en lugar de llevar a cabo los pasos descriptos en la sección anterior para diseñar circuitos desde cero, es posible (y recomendable) reutilizar circuitos que se hayan definido anteriormente, combinándolos de manera que en su conjunto resuelvan el problema mayor.

5.4. Circuitos estándares

En particular, en lo que sigue de esta sección, se explicará una selección de circuitos que han sido estandarizados en la literatura y cuya utilización es recurrente en distintos componentes de la computadora, en la CPU o el subsistema de memoria.

Decodificador

e_1	e_2	s_1	s_2	s_3	s_4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Tabla 5.3: Tabla de verdad para un decodificador de 2 bits

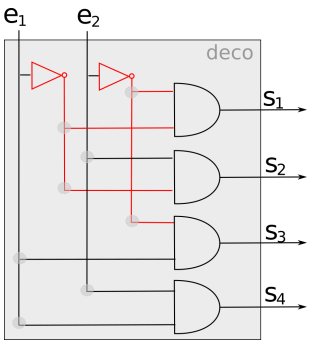


Figura 5.14: Decodificador

El objetivo de un decodificador es el de traducir un código de N bits de entrada en uno de 2^N valores.
Para esto tiene N entradas que representan una cadena de N bits, y selecciona (pone en 1), **una y sólo una** de las 2^N líneas de salida (ver Figura 5.13 (A)). Esto quiere decir que cada línea de salida será activada para una sola de las combinaciones posibles de entrada. Para ilustrar con ejemplos, en la Figura 5.13 (B) se ve como la combinación `00` activa la salida s_1 y la combinación `11` activa la salida s_4. A partir de esta definición, se debe completar una tabla de verdad con 2 columnas de entrada (y entonces 4 combinaciones o filas) y 4 columnas de salida. En el Cuadro 5.3 puede observarse que cada combinación de las entradas genera una única salida en 1 (cada fila tiene un solo caso en 1), y no hay salida que se ponga en 1 en mas de un caso (cada columna tiene un solo caso en 1).

Algo notable de este circuito es que tiene múltiples salidas y por lo tanto cada salida se describe con una fórmula de verdad independiente, es decir que se debe aplicar por separado el método SoP o PoS según el caso. De esta manera, las fórmulas de verdad para cada una de las salidas son:

$$s_1 = \overline{e_1} \bullet \overline{e_2}$$

$$s_2 = \overline{e_1} \bullet e_2$$

$$s_3 = e_1 \bullet \overline{e_2}$$

$$s_4 = e_1 \bullet e_2$$

Finalmente, se construye el circuito con el mecanismo desarrollado en la sección 5.2.3, como se muestra en la Figura 5.14.

Multiplexor

Un circuito multiplexor tiene por objetivo proyectar una de las entradas en la salida a partir de una configuración del control. Es una idea muy utilizada en cuando se tiene un recurso compartido (en este caso, el canal de salida) y se debe asignar a una demanda particular (una de las entradas). En la Figura 5.15 (A) se observa la estructura de un multiplexor de 2 entradas.

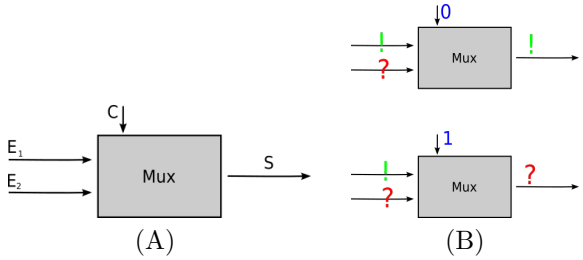


Figura 5.15: Multiplexor de N=1

c	e_1	e_2	s
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabla 5.4: Tabla de verdad para un multiplexor de 2 entradas

De manera general, el multiplexor recibe 2^N entradas de datos, una salida y N entradas de control que permiten seleccionar sólo una de todas las entradas. La entrada seleccionada "pasa por compuertas"(se encamina) hacia la salida.

Cada combinación de las entradas de control corresponde a una entrada de datos, y la salida final del multiplexor corresponderá al valor de dicha entrada seleccionada. En la Figura 5.15 (B) se muestra cómo la línea de control cuando vale 0 activa el paso de la entrada E_1 (con el símbolo "!") y cuándo vale 1 activa el paso de la entrada E_2 (con el símbolo "?")

Para elaborar la tabla de verdad se debe considerar que son 3 entradas, pues a pesar de que una de ellas se separa semánticamente no deja de ser un dato de entrada (ver Cuadro 5.4).

Por último, la fórmula de verdad correspondiente es:

$$S = (\overline{c} \bullet e_1 \bullet \overline{e_2}) + (\overline{c} \bullet e_1 \bullet e_2) + (c \bullet \overline{e_1} \bullet e_2) + (c \bullet e_1 \bullet e_2)$$

Para ensamblar el circuito se puede tomar uno de los siguientes criterios. Por un lado, es posible conectar las entradas como indica la fórmula de verdad anterior, y por otro es posible conseguir otra fórmula equivalente que simplifique el circuito final, en terminos de cantidad de compuertas así como de trabajo de diseño.

A continuación se desarrolla el segundo enfoque, para lo cual se aplica una de las leyes de equivalencia de la lógica proposicional:

$$S = (\overline{c} \bullet e_1 \bullet \overline{e_2}) + (\overline{c} \bullet e_1 \bullet e_2) + (c \bullet \overline{e_1} \bullet e_2) + (c \bullet e_1 \bullet e_2) =$$
$$= \overline{c} \bullet e_1 \bullet (\overline{e_2} + e_2) + c \bullet e_2 \bullet (\overline{e_1} + e_1)$$

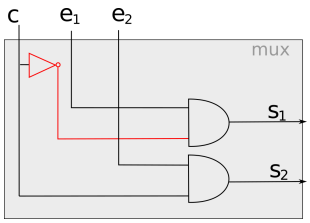


Figura 5.16: Multiplexor

$$= \overline{c} \bullet e_1 \bullet 1 + c \bullet e_2 \bullet 1$$
$$= \overline{c} \bullet e_1 + c \bullet e_2$$

A partir de esta última fórmula de verdad es posible construir un circuito mas simple como se muestra en la Figura 5.16.

Demultiplexor

El demultiplexor es un circuito complementario al multiplexor, en el cual se permite configurar por cuál salida se proyecta la entrada. Esto quiere decir que recibe sólo 1 línea de entrada, N líneas de control y 2^N líneas de salida. Este circuito encamina su única línea de entrada a una de sus 2^N líneas de salida dependiendo de los valores de las líneas de control. Es decir, si el valor binario en las líneas de control es k, se selecciona la salida k. Ver la figura 5.17.

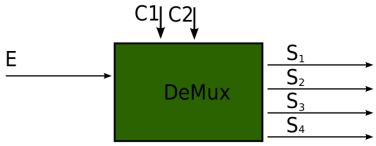


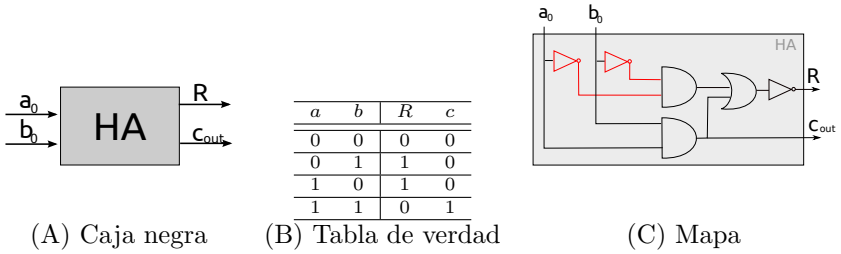
Figura 5.17: Demultiplexor

5.4.1. Circuitos aritméticos

Los denominados circuitos aritméticos resuelven tareas específicas relacionadas a las operaciones aritméticas entre cadenas binarias. Esas cadenas representan en forma binaria dentro del sistema de cómputos, los valores numéricos a operar. Por lo tanto, estos circuitos operan sobre dos cadenas binarias. La Unidad Aritmético-Lógica (ALU) de la CPU, se puede implementar mediante circuitos aritméticos, donde cada operación aritmética puede diseñarse con un circuito independiente.

Semisumador (*Half Adder*)

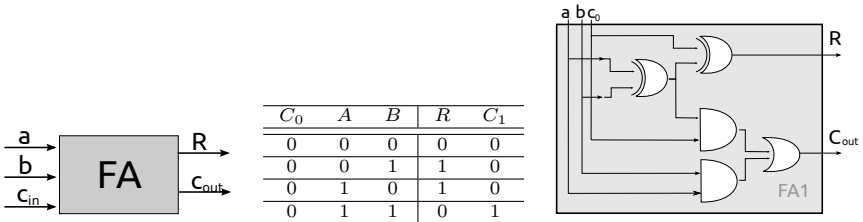
El objetivo de este circuito es el de sumar dos cadenas de un bit (BSS(1)), calculando el resultado (en BSS(1)) e indicando además en otra salida, si hubo



a	b	R	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(A) Caja negra (B) Tabla de verdad (C) Mapa

Figura 5.18: Circuito semisumador



C_0	A	B	R	C_1
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(A) Caja negra (B) Tabla de verdad (C) Mapa

Figura 5.19: Circuito sumador para BSS(1)

o no acarreo (*carry*). Por lo tanto recibe 2 entradas (los operandos a sumar) y obtiene 2 salidas (2 bits) que representan 1 bit que indica el resultado de la suma y 1 bit que indica si hubo acarreo (Ver su caja negra en la figura 5.18 (A)).

Como primer paso se elabora la tabla de verdad cómo se ve en la figura 5.18 (B) , y luego se extraen las dos fórmulas de verdad correspondientes al resultado y al acarreo. La fórmula del resultado R, por la cantidad de 1s en la tabla se puede obtener por PoS o SoP ya que se tienen dos casos con 1:

$$R = \overline{(\overline{a} \bullet \overline{b}) + (a \bullet b)} \text{ - fórmula por PoS}$$

$$R = (\overline{a} \bullet b) + (a \bullet \overline{b}) \text{- fórmula por SoP}$$

Sin embargo, se tomará la opción de PoS para unificar con la siguiente fórmula del acarreo (obtenida por SoP):

$$c = a \bullet b$$

A partir de lo anterior, el circuito del semisumador queda finalmente ilustrado en la Figura 5.18 (C).

Sumador (*Full Adder*)

El objetivo de un sumador es, como su nombre lo indica, sumar 2 cadenas en el sistema BSS(n). Para hacerlo se separa el problema en varios problemas

mas simples: sumar las n columnas que conforman las cadenas de n bits, dado que en cada columna se repite el mismo procedimiento. El hecho de sumar una columna es el trabajo del semisumador, pero teniendo en cuenta que también se necesita, como entrada, información sobre el carry de la columna anterior (la que está inmediatamente a la derecha), por lo que el semisumador, como tal, no es suficiente.

Por lo tanto, el sumador recibe 3 entradas, 2 de ellas corresponden a los operandos BSS(1), y la tercera que indica si hay acarreo pendiente. Al igual que el semisumador obtiene una salida para el resultado y otra para el acarreo resultante. Ver la caja negra en la figura 5.19 (A). La tabla de verdad para el sumador tiene 8 filas (las combinaciones de sus 3 entradas) y puede verse en la figura 5.18 (B). Notar que la primera mitad de la tabla (las primeras 4 filas) son análogas a lo que indica la tabla del semisumador, pues son los casos donde c_0 vale 0, es decir que no hay arrastre desde la columna anterior.

A partir de la tabla de verdad se derivan por SoP las siguientes fórmulas de verdad:

$$R = (\overline{C_0} \bullet \overline{A} \bullet B) + (\overline{C_0} \bullet A \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet \overline{B}) + (C_0 \bullet A \bullet B) =$$

$$C_1 = (\overline{C_0} \bullet A \bullet B) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet \overline{B}) + (C_0 \bullet A \bullet B)$$

La fórmula para R es posible simplificarla aplicando las identidades trigonométricas. En particular, aplicando la propiedad distributiva (dos veces)se obtiene la expresión:

$$R = \overline{C_0}(\overline{A} \bullet B + A \bullet \overline{B}) + C_0(\overline{A} \bullet \overline{B} + A \bullet B)$$

Luego, aplicando la definición de xor se obtiene la expresión:

$$R = \overline{C_0}(A \oplus B) + C_0(\overline{A} \bullet \overline{B} + A \bullet B)$$

Además, dado que es posible demostrar que $\overline{A} \bullet \overline{B} + A \bullet B$ es equivalente a $\overline{(A \oplus B)}$ (ver Lema 5.5.1) entonces:

$$R = \overline{C_0}(A \oplus B) + C_0(\overline{(A \oplus B)})$$

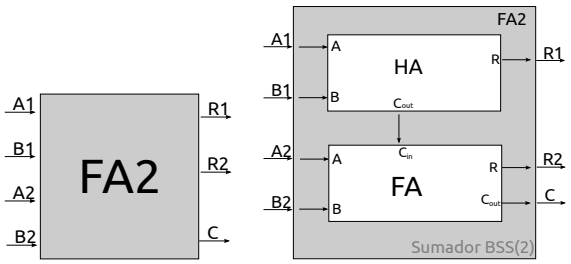
y a partir de esto último, aplicando nuevamente la definición de XOR se obtiene la fórmula de verdad para R: $C_0 \oplus (A \oplus B)$.

Análogamente es posible reducir la fórmula del bit de carry de salida (ver Lema 5.5.4) a la expresión: $A \bullet B + C_0 \bullet (A \oplus B)$

Finalmente, se construye el circuito del sumador para cadenas del sistema $BSS(1)$ como se muestra en la figura 5.19 (C).

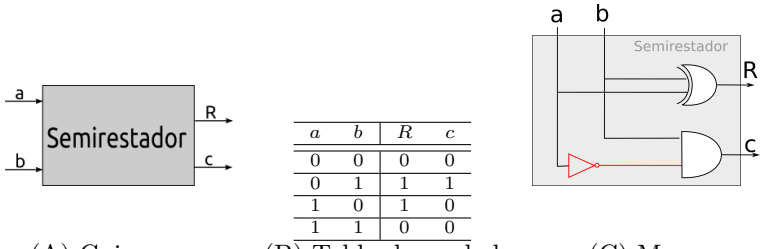
Sumador para cadenas BSS(2)

Contando con el semisumador y el sumador para $BSS(1)$, es posible entonces construir un sumador para cadenas $BSS(2)$, que debe tener la interfaz descripta en la Figura 5.20 (A). Esto se consigue ensamblando adecuadamente los dos circuitos presentados anteriormente en esta sección, como se describe en el mapa del circuito de la Figura 5.20(B). Es importante notar que esta idea puede extenderse para construir sumadores para una mayor cantidad de bits, replicando la idea de conectar en cascada los sumadores y usando un semisumador para la primer columna.



(A) Caja negra (B) Mapa

Figura 5.20: Circuito sumador para $BSS(2)$



a	b	R	c
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

(A) Caja negra (B) Tabla de verdad (C) Mapa

Figura 5.21: Circuito semirestador

Semirestador

De manera similar al caso de la suma, la operación de resta requiere la construcción de un semirestador y un restador. La interfaz del *semirestador* es muy similar a la del *semisumador* (ver Figura 5.21 (A)), pero para completar su tabla de verdad es necesario analizar los 4 casos de la resta de un bit:

caso 1	**caso 2**	**caso 3**	**caso 4**
$\begin{array}{r} 0 \\ -\ 0 \\ \hline 0 \end{array}$	$\begin{array}{r} c=1\quad 2 \\ 0 \\ -\ 1 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ -\ 0 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ -\ 1 \\ \hline 0 \end{array}$

Entonces la tabla de verdad se completa como se ve en la Figura 5.21 (B), y las fórmulas de verdad para las salidas son:

$$R = (\overline{A} \bullet B) + (A \bullet \overline{B}) = A \oplus B$$

$$c = \overline{A} \bullet B$$

El circuito del *semirestador* se construye conectando las compuertas según se indica en las fórmulas anteriores, quedando graficado como se muestra en la Figura 5.21(C).

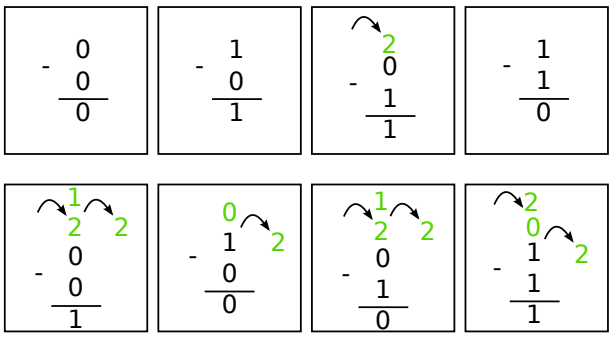
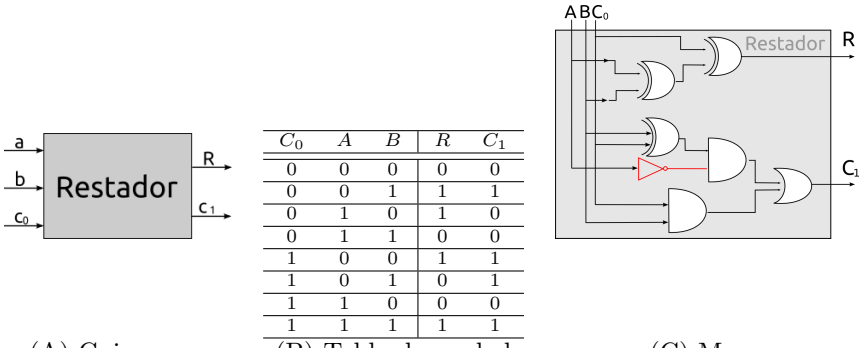


Figura 5.22: Casos posibles en una resta con carry



C_0	A	B	R	C_1
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

(A) Caja negra (B) Tabla de verdad (C) Mapa

Figura 5.23: Restador BSS(1)

Restador

Análogamente, el circuito *restador* tiene similitudes con el circuito sumador, y su interfaz es la misma (ver Figura 5.23 (A)). La figura 5.22 describe los casos posibles entre los operandos y el borrow anterior y se usó como referencia para construir la tabla de verdad que se describe en la Figura 5.23 (B). De esta tabla se deducen la expresión para R:

$$R = C_0 \oplus (A \oplus B)$$

Es importante notar que esta expresión es una simplificación como se hizo para el caso del sumador pues en ambos casos a la salida R le corresponde la misma SoP. Por otro lado, la fórmula de verdad del *carry* (o *borrow*) C_1 es cómo sigue:

$$C_1 = (\overline{C_0} \bullet \overline{A} \bullet B) + (C_0 \bullet \overline{A} \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet B)$$

Y esta expresión se puede reescribir a expresiones mas simples como se muestra en los Lemas 5.5.2 y 5.5.3.

Finalmente, el mapa del circuito se ensambla como se muestra en la Figura 5.23 (C).

5.5. Anexos

Lógica proposicional

En el segundo paso del método presentado se utilizan las reglas de la lógica proposicional que permiten definir los problemas en términos de proposiciones.

Las proposiciones son frases en lenguaje natural que pueden ser verdaderas o falsas, y pueden ser simples, como por ejemplo A="hoy es viernes" o B="hoy está lloviendo", o pueden ser complejas, (composición de proposiciones simples) como por ejemplo: "hoy es viernes **y** hoy está lloviendo".

Se utilizará la lógica proposicional para las operaciones lógicas entre las proposiciones. Se utilizan letras minúsculas (p, q, r) para simbolizar las proposiciones y conectivos lógicos para combinarlas (ver tabla 5.5.

Función booleana	símbolo	Ejemplo de uso
Conjunción	$\bullet$	$A \bullet B$
Disyunción	$+$	$A + B$
Negación	$\neg$	$\neg A$

Tabla 5.5: Simbología utilizada para las funciones lógicas

Considerar el siguiente ejemplo con las proposiciones p, q y r definidas de la siguiente manera:

- p = "está lloviendo"
- q = "el sol está brillando"
- r = "hay nubes en el cielo"

Simbolizamos las siguientes frases:

1. Está lloviendo y el sol está brillando: $p \bullet q$
2. Está lloviendo o hay nubes en el cielo: $p + r$
3. No está lloviendo, o el sol no está brillando y hay nubes en el cielo: $\overline{p}+(\overline{q}\bullet r)$

La suposición fundamental del cálculo proposicional consiste en que los valores de verdad de una proposición construida a partir de otras proposiciones quedan completamente determinados por los valores de verdad de las proposiciones originales (o "de entrada").

En el cuadro 5.6 se detallan las tablas de verdad de las operaciones lógicas elementales. La tabla de la *conjunción* indica que, el conectivo lógico "y" (and) sólo será verdadero cuando ambas proposiciones p y q sean verdaderas. En el caso de la *disyunción* , la tabla indica que, si al menos una de las proposiciones es verdadera, la proposición formada por el conectivo "o" (or) será verdadera. Por ultimo, en el caso de la *negación* , si la proposición p es verdadera, su negación será falsa y viceversa (not).

Conjunción

p	q	$p \bullet q$
0	0	0
0	1	0
1	0	0
1	1	1

Disyunción

p	q	$p + q$
0	0	0
0	1	1
1	0	1
1	1	1

Negación

p	$\overline{p}$
0	1
1	0

Tabla 5.6: Tablas de verdad de las operaciones lógicas elementales

5.5.1. Lemas

En este apartado se incluyen las demostraciones usadas en los apartados anteriores.

Lemma 5.5.1 *En este lema se demuestra que* $\overline{(a \oplus b)} = (\overline{b} \bullet \overline{a}) + (a \bullet b)$*, mediante las propiedades (identidades lógicas) indicadas.*

$$\begin{aligned}
\overline{(a \oplus b)} &= \overline{(\overline{a} \bullet b) + (a \bullet \overline{b})} \\
\textit{Ley de De Morgan} &= \overline{(\overline{a} \bullet b)} \bullet \overline{(a \bullet \overline{b})} \\
\textit{Ley de De Morgan} &= (\overline{\overline{a}} + \overline{b}) \bullet \overline{(a \bullet \overline{b})} \\
\textit{Ley de De Morgan} &= (\overline{\overline{a}} + \overline{b}) \bullet (\overline{a} + \overline{\overline{b}}) \\
\textit{Ley de doble negación} &= (a + \overline{b}) \bullet (\overline{a} + b) \\
\textit{Ley distributiva} &= ((a + \overline{b}) \bullet \overline{a}) + ((a + \overline{b}) \bullet b) \\
\textit{Ley distributiva} &= (a \bullet \overline{a}) + (\overline{b} \bullet \overline{a}) + ((a + \overline{b}) \bullet b) \\
\textit{Ley distributiva} &= (a \bullet \overline{a}) + (\overline{b} \bullet \overline{a}) + (a \bullet b) + (\overline{b} \bullet b) \\
\textit{Ley de contradicción} &= (0) + (\overline{b} \bullet \overline{a}) + (a \bullet b) + (0) \\
\textit{Ley de identidad} &= (\overline{b} \bullet \overline{a}) + (a \bullet b)
\end{aligned}$$

Lemma 5.5.2 *En este lema se demuestra que la Suma de Productos del Carry de salida del restador (apartado 5.4.1),* $(\overline{C_0} \bullet \overline{A} \bullet B) + (C_0 \bullet \overline{A} \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet B)$*, es equivalente a la expresión* $\overline{A} \bullet (C_0 \oplus B) + (C_0 \bullet B)$*, mediante las propiedades (identidades lógicas) indicadas.*

$$\begin{aligned}
&(\overline{C_0} \bullet \overline{A} \bullet B) + (C_0 \bullet \overline{A} \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet B) \\
\textit{Ley distributiva} &= \overline{A} \bullet (\overline{C_0} \bullet B + C_0 \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet B) \\
\textit{Ley distributiva} &= \overline{A} \bullet (\overline{C_0} \bullet B + C_0 \bullet \overline{B}) + (C_0 \bullet B) \bullet (\overline{A} + A) \\
\textit{Ley del medio excluido} &= \overline{A} \bullet (\overline{C_0} \bullet B + C_0 \bullet \overline{B}) + (C_0 \bullet B) \bullet (1) \\
\textit{Ley de identidad} &= \overline{A} \bullet (\overline{C_0} \bullet B + C_0 \bullet \overline{B}) + (C_0 \bullet B) \\
\textit{definición de XOR} &= \overline{A} \bullet (C_0 \oplus B) + (C_0 \bullet B)
\end{aligned}$$

Lemma 5.5.3 *Este lema presenta otra equivalencia para la Suma de Productos del Carry de salida del restador (apartado 5.4.1):* $\overline{A} \bullet B + C_0 \bullet (A \bullet \overline{B})$*, mediante las propiedades (identidades lógicas) indicadas.*

$$(\overline{C_0} \bullet \overline{A} \bullet B) + (C_0 \bullet \overline{A} \bullet \overline{B}) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet B) =$$
por ley dist. en 2 y 4:
$$= (\overline{C_0} \bullet \overline{A} \bullet B) + C_0 \bullet [(\overline{A} \bullet \overline{B}) + (A \bullet B)] + (C_0 \bullet \overline{A} \bullet B) =$$
por lema 5.5.1: $= (\overline{C_0} \bullet \overline{A} \bullet B) + C_0 \bullet [\overline{A \oplus B}] + (C_0 \bullet \overline{A} \bullet B) =$
por ley asociativa: $= (\overline{C_0} \bullet \overline{A} \bullet B) + (C_0 \bullet \overline{A} \bullet B) + C_0 \bullet [\overline{A \oplus B}] =$
por ley distributiva en términos 1 y 2 $= \overline{A} \bullet B \bullet (\overline{C_0} + C_0) + C_0 \bullet \overline{A \oplus B} =$
por ley del medio excluido $= \overline{A} \bullet B \bullet (1) + C_0 \bullet \overline{A \oplus B} =$
por ley de identidad $= \overline{A} \bullet B + C_0 \bullet \overline{A \oplus B} =$
por lefinición del or exclusivo $= \overline{A} \bullet B + C_0 \bullet (\overline{A} \bullet B + A \bullet \overline{B}) =$
por ley distributiva $= \overline{A} \bullet B + C_0 \bullet (\overline{A} \bullet B) + C_0 \bullet (A \bullet \overline{B}) =$
por ley distributiva $= \overline{A} \bullet B \bullet (1 + C_0) + C_0 \bullet (A \bullet \overline{B}) =$
por ley de identidad en $+ = \overline{A} \bullet B \bullet 1 + C_0 \bullet (A \bullet \overline{B}) =$
por ley de identidad en $\bullet = \overline{A} \bullet B + C_0 \bullet (A \bullet \overline{B})$

Lemma 5.5.4 *En este lema se demuestra que* $(\overline{C_0} \bullet A \bullet B) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet \overline{B}) + (C_0 \bullet A \bullet B) = A \bullet B + C_0 \bullet (A \oplus B)$*, mediante las propiedades (identidades lógicas) indicadas.*

$$(\overline{C_0} \bullet A \bullet B) + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet \overline{B}) + (C_0 \bullet A \bullet B)$$
Ley distributiva (terminos 1 y 4) $= (\overline{C_0} + C_0) \bullet A \bullet B + (C_0 \bullet \overline{A} \bullet B) + (C_0 \bullet A \bullet \overline{B})$
Ley distributiva(terminos 3 y 4) $= (\overline{C_0} + C_0) \bullet A \bullet B + C_0 \bullet (\overline{A} \bullet B + A \bullet \overline{B})$
Ley del medio excluido $= (1) \bullet A \bullet B + C_0 \bullet (\overline{A} \bullet B + A \bullet \overline{B})$
Ley de identidad $= A \bullet B + C_0 \bullet (\overline{A} \bullet B + A \bullet \overline{B})$
Definición XOR $= A \bullet B + C_0 \bullet (A \oplus B)$

Capítulo 6

Q1: Repertorio de instrucciones

6.1. Arquitectura de Von Neumann

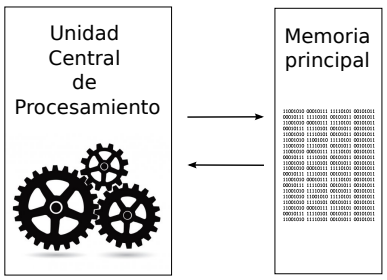


Figura 6.1: Arquitectura conceptual de John Von Neumann

Los sistemas de cómputo tienen como objetivo la resolución de problemas computables, algunos de ellos elementales y otros mucho mas complejos. Los problemas computables son aquellos para los cuales se puede diseñar un algoritmo que lo resuelve. Recordemos que un *algoritmo* es una secuencia de pasos/órdenes/instrucciones ordenadas y finitas que pertenecen a un lenguaje formal. Si ese lenguaje está pensado para un sistema de cómputos que lo *comprende*, entonces los algoritmos son ejecutables por un autómata, a los cuales se los define como *programas.* Así, el sistema de cómputos (o computadora), resuelve los problemas computables mediante la **ejecución de programas**.

algoritmo

programas.

Como se desarrolló en la sección 2.1, estos sistemas de cómputos se pensaron en un principio para resolver los problemas de cada área de manera puntual, hasta que John Von Neumann propuso una computadora de *propósito general.* Este enfoque, reconociendo la complejidad en la operación de sus antecesores, propone separar el soporte físico, que traduce las órdenes en acciones mecánicas (o eléctricas), de las órdenes (o instrucciones) propias del lenguaje. Lo que hoy se conoce como *Arquitectura de Von Neumann* (ver figura 6.1) tiene como compo-

propósito general.

Arquitectura de Von Neumann

nentes principales un dispositivo denominado *Unidad Central de Procesamiento,* CPU (por sus siglas en inglés *(Central Processing Unit*), y un dispositivo denominado *Memoria principal.*
La CPU es una unidad que se alimenta de las instrucciones en un orden determinado y las ejecuta individualmente, mientras que la Memoria es un espacio de almacenamiento que contiene tanto las instrucciones a ejecutar (los programas) como sus datos. Esta propuesta marca un momento de inflexión histórico muy importante que separa lo sucedido hasta ese momento, donde las computadoras eran diseñadas y operadas por las mismas personas que formaban parte de un grupo muy selecto de la ciencia, de lo ocurrido con posterioridad: el surgimiento de nuevas disciplinas. Por un lado se comenzaron a diseñar computadoras de propósito general (cada vez más eficientes, más pequeñas y económicas), y por otro lado, se comenzaron a construir lenguajes para cada modelo, que permiten diseñar programas que resuelven problemas de distintas índoles.
En resumen, esta propuesta provocó el surgimiento de la programación y de la industria del *software* como una actividad independiente al diseño y fabricación de computadoras (o *hardware).*

Unidad Central de Procesamiento,

Memoria principal.

software

hardware).

A continuación veremos las dos unidades fundamentales que conforman la CPU: la **ALU** (del inglés *Arithmetic Logic Unit* ó unidad aritmético-logica) y la **UC** (Unidad de control).

La ALU es un circuito lógico digital encargado de realizar las operaciones aritmeticas (suma, resta, multiplicación y división) y las operaciones lógicas (disyunción, conjunción y negación) entre dos cadenas binarias, que dependiendo del sistema de numeración a utilizar representarán los operadores correspondientes.

La Unidad de Control (UC) es el circuito digital principal, responsable de la ejecución de los programas. Para lo cual, se requiere implementar el ciclo de ejecución de instrucciones, que será detallado en el apartado 6.4.

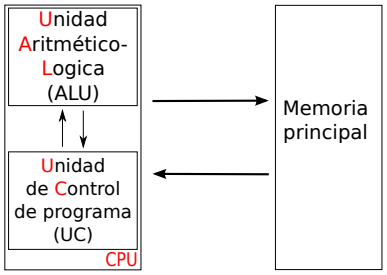


Figura 6.2: Diagrama de la arquitectura de Von Neumann que incorpora las unidades de la CPU

La arquitectura con la que trabajaremos se denomina **Arquitectura Q,** y se basa, de una manera más simplista, en la arquitectura de Von Neumann.
Para comprender las secciones subsiguientes es importante destacar que en la arquitectura Q, la CPU, además de las unidades, cuenta con un conjunto de registros que permiten almacenar información necesaria para ejecutar los pro-

gramas. Éstos se dividen en 2 tipos: **registros de uso específico**, reservados enteramente para el uso de la CPU, y **registros de uso general**, accesibles para quienes programan. En los sucesivos capítulos ahondaremos sobre su evolución, donde se irán presentando los diferententes elementos que se incorporan a cada versión. Cada versión se la identifica con un número, iniciando por el 1, de esta manera comenzaremos con Q1 (ver figura 6.3).

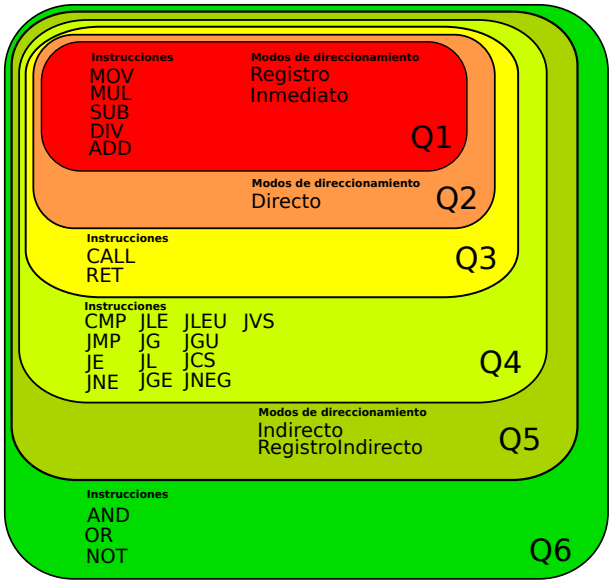


Figura 6.3: Capas de la Arquitectura Q

6.2. Repertorio de instrucciones

La UC ejecuta programas siguiendo una lista de instrucciones, las cuales deben estar disponibles en alguna codificación que la computadora sea capaz de interpretar y traducir para obtener un resultado. Como se ve en el capítulo 5, las computadoras actuales están construidas con circuitos digitales, y por lo tanto, necesitan que la mencionada codificación sea provista en términos de cadenas binarias. Los detalles que se explican en esta sección pueden no parecer importantes cuando se está programando con lenguajes de alto nivel como Gobstones o Java donde las características de la arquitectura no son tan *visibles*, pero es necesario conocer cómo opera la computadora internamente para administrar bien los recursos al momento de programar. El funcionamiento del procesador (CPU) está determinado por las instrucciones que ejecuta. Estas instrucciones se denominan **instrucciones máquina** o instrucciones del computador. Al conjunto de instrucciones distintas que puede ejecutar el procesador se lo denomina **repertorio de instrucciones**.

Un punto de encuentro entre las personas que diseñan computadoras y las personas que programan es el **conjunto o repertorio de instrucciones**. Desde

el punto de vista de quien diseña, el conjunto de instrucciones máquina constituye la especificación o requisitos funcionales del procesador. Es decir que diseñar el procesador es una tarea que, en buena parte, implica construir circuitos que cumplan con tales requisitos. Desde el punto de vista de quien programa, las instrucciones del lenguaje ensamblador son las herramientas disponibles para expresar los programas. Además, debe conocer la estructura de los espacios de almacenamiento, es decir tanto de los registros como de la memoria, los tipos de datos que acepta la máquina, y el funcionamiento (o restricciones) de la ALU. Así, cada instrucción debe contener la información que necesita el procesador para su ejecución. Dichos elementos son:

- **Código de operación:** especifica la operación a realizar (suma, resta, movimiento, etc). La operación se indica mediante un código binario denominado código de operación, o de manera abreviada, *codop* .
- **Referencia a los operandos de entrada:** la operación puede implicar uno o más operandos que son necesarios para la operación.
- **Referencia al resultado:** la operación puede producir un resultado por lo que puede ser necesario indicar donde se almacenará.
- **Referencia a la siguiente instrucción:** indica al procesador de dónde captar la siguiente instrucción tras completarse la ejecución de la instrucción actual

La siguiente instrucción a captar está en memoria principal o, en el caso de un sistema de memoria virtual en memoria principal o en memoria secundaria (disco). En la mayoría de los casos, la siguiente instrucción a captar sigue inmediatamente a la instrucción en ejecución y en tales casos no hay referencia explícita a la siguiente instrucción. Cuando sea necesaria la referencia explícita debe suministrarse la dirección de memoria principal o de memoria virtual.

Por otro lado, los operandos y el resultado pueden estar en alguna de las tres áreas:

- **Memoria principal o virtual**: como en las referencias a instrucciones siguientes, debe indicarse la dirección de memoria principal o memoria virtual.
- **Registro del procesador**: Salvo raras excepciones, un procesador contiene uno o más registros que pueden ser referenciados por instrucciones máquina. Si sólo existe un registro, la referencia a él puede ser implícita. Si existe más de uno, cada registro tendrá asignado un número único y la instrucción debe contener el número del registro deseado. Estos registros se denominan *Registros de uso general* y normalmente son circuitos secuenciales con capacidad de almacenar una cadena de unos cuantos bits.
- **Dispositivo de entrada/salida**:la instrucción debe especificar el módulo y dispositivo de Entrada/Salida para la operación. En el caso de E/S asignadas en memoria, se dará otra dirección de memoria principal o virtual.

6.3. Ciclo de vida de los programas

Originalmente las personas escribían sus programas en *código máquina,* es decir, mediante códigos binarios que hacían que la programación sea una tarea muy engorrosa y propensa a errores. Por ello, la evolución de la programación propuso utilizar representaciones simbólicas para las instrucciones máquina, donde estos códigos, denominados *codops*, se representan mediante abreviaturas, denominadas *nemotécnicos* que indican la operación en cuestión (los nombres, a modo de estándar, se encuentran en inglés).

Ejemplos de la arquitectura Q:

`MOV`	Almacena una copia
`ADD`	Sumar
`SUB`	Restar
`MUL`	Multiplicar
`DIV`	Dividir

Los operandos también suelen representarse simbólicamente. Por ejemplo, la instrucción `ADD R,Y` se utiliza para sumar el valor contenido en la variable **"Y"** con el valor contenido en la variable **"R"**, que en este caso se trata de un registro. Pero esta representación requirió el desarrollo de un dispositivo que tradujera dicho código simbólico al lenguaje de la computadora. Este lenguaje simbólico se denomina *código fuente,* y el mencionado dispositivo es el *ensamblador,* quien además, carga el programa en memoria principal. A este proceso de traducir del código fuente a código máquina se lo denomina **ensamblar**.

código fuente,

ensamblador,

Es raro encontrar, hoy en día, a personas que programen en lenguaje máquina. La mayoría de los programas actuales se escriben en un lenguaje de alto nivel o, en ausencia del mismo, en el lenguaje ensamblador (Assembler). Luego éste se ensambla y se carga en memoria, aplicando alguna política de administración de memoria, de manera que cuando se necesita dicho programa, se ejecuta **el ciclo de ejecución de una instrucción** tantas veces como sea necesario, a partir de la acción concreta de un/a usuario/a. Ver figura 6.4.

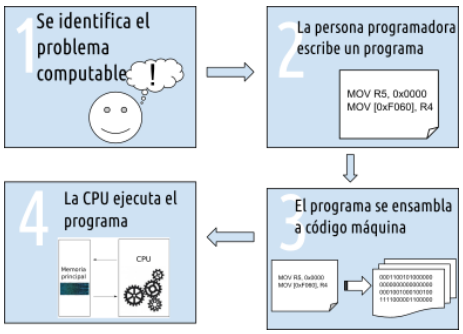


Figura 6.4: Ciclo de Vida de un programa

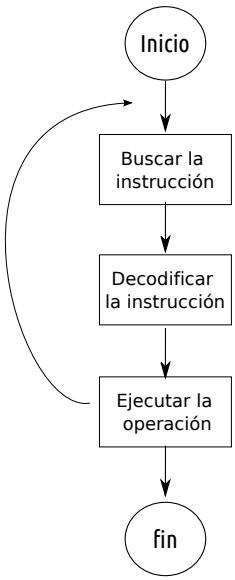


Figura 6.5: Ciclo de ejecución de una instrucción

6.4. Ejecución de un programa

ciclo de ejecución de una instrucción

Recordemos que la ejecución de un programa es la ejecución ordenada de cada una de las instrucciones que lo componen, y la ejecución de cada una sigue el *ciclo de ejecución de una instrucción* .

El ciclo de ejecución de una instrucción en Q1 consta de las siguientes 3 etapas: **Búsqueda de la instrucción**, **Decodificación** y **Ejecución de la operación** (ver Figura 6.5).

1. La **búsqueda de instrucción** se encarga de recuperar el código máquina completo de la instrucción y ponerlo disponible en uno de los *registros de uso específico* de la CPU. Si bien este registro toma diferentes nombres en las muchas arquitecturas disponibles en el mercado, la manera general de nombrarlo es *Instruction Register* , IR, que significa Registro de instrucción.

2. La **decodificación** debe ser capaz de identificar la operación que resuelve dicha instrucción y si ésta requiere de operandos para su posterior ejecución

3. La **ejecución** normalmente se traduce en que la UC delegue a otro circuito la realización de ésta tarea, como ser el caso de las instrucciones con operaciones aritméticas que son delegadas a la ALU.

Por ejemplo, para escribir el programa que resuelva la siguiente operación: `A=A-B`, se necesita poder escribir una instrucción semejante a: `restar VALOR-DE-A, VALOR-DE-B`. De esta manera surge la necesidad de tener una cadena binaria que represente dicha instrucción para que pueda ser entendida por la unidad de control, quien le indicará a la ALU que debe realizar una resta, y por lo tanto,

activar el circuito Restador. Esta representación se denomina **código máquina** y debe incluir en cada instrucción la siguiente información:

- Qué operación es (suma, resta,etc)
- Cuáles son los operandos
- Dónde se guarda el resultado

En este punto, es razonable hacerse la pregunta: ¿Qué son A y B? Estas son variables (como las que se especifican en matemáticas) que almacenan un valor cada una. Para tal fin, en una computadora, los valores de las variables deben ser mantenidos en los registros de uso general. En este caso necesitaremos un registro para A y otro para B, y el resultado se almacena en el registro A: `A=A-B`.

Ahora veamos un ejemplo con una suma entre el registro A y B: `A = A + B`.

- **Búsqueda de instrucción**: como primera etapa del ciclo de ejecución de la instrucción, la UC busca la instrucción, realizando una operación de lectura. Así es que obtiene la cadena correspondiente, en código máquina, y la copia en el registro IR (Instruction Register).
- **Decodificación:** la UC decodifica la instrucción contenida en el registro IR, donde identifica, mediante el código de la operación (codop), que en este caso la operación representa una suma. Luego, delega en la ALU la operación junto con ambos operandos. Cada operando recibe un nombre dentro de la instrucción. Se denomina *Operando Destino* al operando donde se alamacenará el resultado de la operación, en este caso el *operando A*, y *Operando Origen*, al segundo operando de la operación, en este caso el *operando B*.
- **Ejecución:** la ALU ejecuta la operación suma, activando el circuito sumador (FullAdder) y almacenando el resultado en el registro A.

6.5. Características de la arquitectura Q1

Como se anticipó en la figura 6.3, la arquitectura Q1 tiene las siguientes características:

- **Registros de uso general:** cuenta con 8 (ocho) registros de uso general de 16 bits cada uno, denominados: `R0`, `R1`, `R2`, `R3`, `R4`, `R5`, `R6` y `R7`. Recordar que son registros accesibles para quienes programan, es decir que son variables disponibles para usar en los programas.
- **Instrucciones:** cuenta con 5 (cinco) instrucciones de 2 operandos: `ADD` para la suma, `SUB` para la resta, `MUL` para la multiplicación, `DIV` para la división entera y `MOV` para copiar el valor de un operando a otro.
- **Operandos:** los operandos pueden ser referencias a variables, mediante el uso de los registros, o a valores constantes.
- **Constantes:** Las constantes también debe tener 16 bits, y se escriben en hexadecimal con la sintaxis: `0xhhhh`, donde 'h' es un caracter hexadecimal. Recordemos que al primer operando se lo denomina *operando destino* ,

operando destino

por ser además donde se almacenará el resultado, y al segundo *operando origen* .

operando origen

- **El sistema de numeración:** los sistemas soportados por esta arquitectura son BSS y CA2, considerando una ALU como se describe en el capítulo 5.

Así como contamos con la arquitectura Q como soporte de bajo nivel, es importante mencionar que para programar utilizaremos el **lenguaje Q**, el cual respeta el repertorio de instrucciones mencionadas en la arquitectura. De esta manera, el lenguaje irá evolucionando conforme cada versión de la arquitectura.

Ejemplo de una instrucción, sintácticamente válida, en el lenguaje Q1 es: `MOV R5, 0x0002`. Lo que realiza esta instrucción es copiar la **cadena constante** `0x0002` al registro R5, que es una **variable**. Es decir que luego de ejecutarse dicha instrucción, el registro R5 tendrá la cadena hexadecimal 0002, que representa el valor 2.
Otro ejemplo puede ser: `ADD R5, R3`. Suma el contenido de la **variable R3** con el contenido de la **variable R5** y almacena el resultado en R5 (operando destino). Es decir que si por ejemplo R5 tiene la cadena `0x0004` y R3 tiene la cadena `0x0002`, entonces, luego de ejecutarse la instrucción, R5 tendrá la cadena 0x0006 (0x0004 + 0x0002 = 0x0006).
Notar que en el primer ejemplo la instrucción tiene como operando destino una variable y como operando origen una constante, mientras que en el segundo ejemplo, ambos son variables.

6.5.1. Ensamblar instrucciones en Q1

Si bien escribimos programas en código fuente, para que sean ejecutados por una computadora, deben estar escritos en código máquina. Por lo cual es necesario establecer, en la arquitectura, las reglas de traducción **código fuente-código máquina** que debe aplicar un **ensamblador**. Estas reglas se conocen como el **formato de instrucción**, el cual define la organización de los bits dentro de una instrucción en términos de las partes que la componen, tanto sobre la cantidad de bits destinados a cada parte, como el orden en que se deben escribir.
Como parte del formato de una instrucción, se deben incluir campos con información sobre **la operación a ejecutar** y **los datos (operandos)**, indicando, mínimamente el código de la operación y un mecanismo para acceder hasta el/los operando/s. A este mecanismo lo denominamos *modos de direccionamiento* . Esta información es importante, dado que los datos puede provenir de diferentes orígenes, y por tanto, de no contar con un modo de direccionamiento, la UC no sabría dónde ni cómo acceder a ellos.
Q1 cuenta con 2 modos de direccionamientos: **Inmediato** y **Registro**.

- **Modo de direccionamiento Inmediato:** es el mecanismo más sencillo, dado que el operando está presente en la propia instrucción. Este modo puede utilizarse para definir y utilizar constantes, o para fijar valores iniciales de variables (inicializar).
- **Modo de direccionamiento Registro:** este mecanismo indica que el operando se encuentra en uno de los registros de la CPU, reduciendo así

Modo	Codificación	Sintaxis
Inmediato	`000000`	`0xHHHH`
Registro	`100rrr`	`Rx`

Donde `rrr` es una codificación (en 3 bits) del número de registro.

Tabla 6.1: Códigos de los modos de direccionamiento en Q1

el tiempo de acceso. Otra ventaja importante es que, en el mismo campo, además se indica de qué registro se trata. La desventaja principal es que el espacio de almacenamiento es limitado, pues se espera que la cantidad de registros de uso general no sea muy grande.

El formato de instrucción de Q1 contempla instrucciones de 2 operandos, y el tamaño de cada campo está expresado en bits.

`Cod_Op (4b)`	`Modo Destino(6b)`	`Modo Origen(6b)`	`Origen(16b)`

Este formato codifica en primer lugar, el código de la operación (campo Cod_Op) que ocupa 4 bits. En segundo lugar los modos de direccionamientos correspondientes a cada operando (inmediato o registro) que ocupan 6 bits cada uno. Y finalmente el operando origen, que ocupa 16 bits.
El campo Cod_Op se completa siguiendo la codificación indicada en el cuadro 6.2, donde se detallan las instrucciones de Q1 con su código de operación y el efecto esperado. El efecto se utiliza para indicar qué resultado se obtuvo y dónde se almacenó el mismo luego de ejecutar la instrucción. La sintaxis utilizada es una flecha que apunta a la variable de almacenamiento. Ejemplo: R1 $\leftarrow$ 0x0000. En este ejemplo el registro R1 finalizó con la cadena 0x0000.
Los 6 bits de cada modo de direccionamiento se codifican considerando el cuadro 6.1.
Por último el campo Origen es opcional, pues es necesario sólo cuando el operando origen tiene el modo de direccionamiento Inmediato. Las instrucciones que tienen en modo destino, operandos del tipo Inmediato son consideradas como inválidas por el procesador, dado que no es posible almacenar un dato en una constante.

Nota: es importante destacar que la operación `DIV` es una división entera. Por otro lado, a diferencia del resto de las operaciones y dada la naturaleza de la multiplicación, el resultado de la operación `MUL` ocupa 32 bits en lugar de 16, por lo que la arquitectura Q, destinará el registro R7 para almacenar los 2 bytes mas significativos y el operando Destino para los 2 bytes menos significativos (1 byte = 8 bits).

En el cuadro a continuación, ver figura 6.3, veremos de ejemplo cómo quedan ensambladas 2 instrucciones en que respetan su formato de instrucción.
Notar que el código máquina se presenta en dos notaciones, binaria y hexadecimal, dado que la versión en hexadecimal, además de ser mas compacta, es mas legible.

Operación	Cod Op	Efecto
MUL	0000	{R7, Dest} ← Dest * Origen
MOV	0001	Dest ← Origen
ADD	0010	Dest ← Dest + Origen
SUB	0011	Dest ← Dest - Origen
DIV	0111	Dest ← Dest % Origen (% denota la división entera)

Tabla 6.2: Codigos de operación y efecto de las instrucciones en Q1

Código fuente	Código máquina	
	Binario	Hexadecimal
`MOV R5, 0xFF00`	`0001 100101 000000 1111 1111 0000 0000`	`1 9 4 0 F F 0 0`
`ADD R5, R3`	`0010 100101 100011`	`2 9 6 3`

Tabla 6.3: Ejemplos de instrucciones ensambladas

6.5.2. Ejecución de programas Q1

Como se indicó previamente, la ejecución de un programa en Q1, es la ejecución de un ciclo de ejecución por cada instrucción del programa (ver Figura 6.5). A modo de integración, en esta sección se llevará a cabo una simulación de la ejecución de un programa de ejemplo, ya ensamblado (en código máquina). Para ello, se asume que actualmente `R5=0x0001` y `R3=0x0001`.

```
1940 000F
2963
0940 0002
```

Código 6.1: Programa en código máquina hexadecimal

⁂

Primer ciclo

Búsqueda de Instrucción (BI): se lee `IR=1940 000F`

Decodificación: primero desempaquetamos: `0001 1001 0100 0000 0000 0000 0000 1111`. Se obtiene que la operación es MOV hacia R5 con un inmediato

0001	100101	000000	0000000000001111

Ejecución: (efecto) `R5←0x000F`

En la ejecución del primer ciclo, luego de la BI, la UC lee el código máquina del registro IR que contiene la cadena: `1940 000F`. Al decodificar la instrucción, la UC identifica:

- Mediante el codop `0001`, que se trata de una operación de copiado de datos (MOV).

- El modo de direccionamiento origen, que se trata de un modo Inmediato, y que el modo de direccoinamiento destino es un Registro, puntualmente la variable R5.
- El campo Origen, que se trata de la cadena constante `0x000F`.

El código fuente de la instrucción es: `MOV R5, 0x000F`.
Finalmente se ejecuta la etapa de Ejecución de la operación, cuyo efecto de la instrucción, ocasiona que la cadena que antes tenía R5 se reemplace con la cadena `0x000F`. Quedando como efecto: R5 ← 0x000F (valor 15).

⁂

Segundo ciclo

BI: se lee `IR=2963`

Decodificación: primero desempaquetamos: `0010 1001 0110 0011`. Se obtiene que la operación es un ADD entre R5 y R3

0010	100101	100011	--

Ejecución: (efecto) `R5 ← 0x0010 (0x000F+0x0001)`

En la ejecución del segundo ciclo, luego de la BI, la UC lee el código máquina de `IR=2963`. De la misma manera que antes, en la etapa de Decodificación, mediante el formato de instrucción identifica que se trata de una operación de suma entre dos registros, R5 y R3, y que el destino del resultado debe ser R5, es decir que el campo Origen no se utiliza en este caso. En código fuente la instrucción es: `ADD R5, R3`.
Para ejecutar esta operación será necesario delegar el cómputo a la ALU, cuyo operación resulta en la suma de las cadenas `000F` y `0001` (15+1), obteniendo el resultado `0010` (16), el cual se almacena en el registro R5. Quedando como efecto: R5 ← 0x0010.

⁂

Tercer ciclo

BI: se lee `IR=0940 0002`

Decodificación: primero desempaquetamos: `0000 1001 0100 0000 0000 0000 0000 0010`. Se obtiene que la operación es un MUL entre R5 y un inmediato

0000	100101	000000	0000000000000010

Ejecución: (efecto) `R5 ← 0x0020 (0x0010 * 0x0002)`

Al comenzar el tercer ciclo, la UC lee el código máquina de `IR=0940 0002`. Al decodificar esta tercera instrucción identifica la operación de multiplicación entre la cadena almacenada en R5 y la cadena constante `0x0002`. En código fuente la instrucción es: `MUL R5, 0x0002`. También en este caso se delega el cómputo a la ALU, con los valores de entrada cuyas cadenas son `0x0010` (16) y `0x0002` (2), arrojando como resultado (valor de salida) la cadena `0x0020` (32) (16*2), la cual se almacena en el registro R5. Quedando como efecto: R5 ← 0x0020.

Capítulo 7

Q2: Memoria principal y subsitema de buses

Hasta ahora hemos visto que la ejecución de programas en Q1 sigue el ciclo de ejecución mencionado en la figura 6.5, y que las instrucciones de los programas, junto con sus datos, se encuentran en la memoria principal.
A continuación se explicará la evolución de la arquitectura Q a la versión Q2, la cual incluye en el ciclo de ejecución, el acceso a memoria principal, así como también la interconexión entre ésta y la CPU.

7.1. Características de la memoria principal

- La memoria principal se compone de un conjunto de *celdas,* donde cada una es un dispositivo que almacena una cadena de bits, siempre y cuando esté alimentada eléctricamente. Cada celda de la arquitectura Q ocupa 16 bits.
- El conjunto de celdas es homogéneo, es decir que todas las celdas tienen la misma capacidad, expresada en cantidad de bits.
- Cada celda se identifica mediante una *dirección,* a través de la cual es posible acceder para su lectura o e scritura. Lo que la lleva a ser la *unidad direccionable más pequeña* .
- A la memoria principal se la conoce también como memoria RAM (Memoria de acceso aleatorio), dado que el acceso a una celda tiene un costo, en tiempo, igual a cualquier otra. Mas detalle sobre cómo esto es posible se da en la sección 7.5.2.
- Las celdas se agrupan en lo que se denomina *palabras.* En general, el tamaño de palabra coincide con la cantidad de datos que se leen en la misma operación. En la arquitectura Q2, una palabra es equivalente a una celda, pues, tanto variables como las operaciones aritméticas manejan 16 bits.

En la figura 7.1 se ilustra una memoria principal de 4 celdas, cuyas direcciones, que se muestran a su izquierda en color rojo, son: 00,10,10 y 11. Notar que cada dirección es de 2 bits, lo que permite direccionar 4 celdas (2^2).

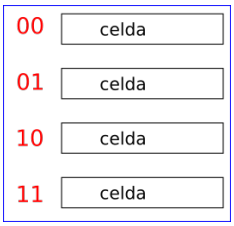


Figura 7.1: Memoria de 4 celdas

7.2. Memoria Principal e instrucciones

Además de las etapas mencionadas en Q1, el ciclo de ejecución de Q2 deberá contemplar el dispositivo desde el cual leer la instrucción y/o los operandos así como también dónde escribir el resultado. Así, la UC deberá:

1. Buscar la instrucción alojada en la memoria principal o desde un dispositivo de entrada y salida.
2. Decodificar la instrucción para determinar que operación ejecutar.
3. Buscar el/los datos necesario, que a partir de ahora, pueden estar en memoria.
4. Ejecutar la instrucción realizando una operación lógica o aritmética.
5. Almacenar el resultado donde corresponda, que también puede ser en memoria.

En este apartado se detallará cómo participa la memoria principal como almacenamiento de las instrucciones, pero también de datos u operandos que se utilizan en ellas (pasos 3 y 5 mencionados anteriormente).

7.2.1. Operaciones sobre la memoria

La memoria admite 2 (dos) operaciones sobre sus celdas: **lectura** y **escritura**.

Para resolver la **lectura de una celda**, la UC le envía a la memoria principal una señal de lectura, y la dirección de la celda a leer. Luego, ésta pone a disposición de la UC, el contenido de la celda correspondiente, es decir el dato, y activa otra señal de control para indicar su finalización.

Para realizar una **escritura en una celda**, la UC le envía a la memoria principal una señal de escritura, la dirección de la celda donde escribir y el dato a escribir. A partir de esto, se actualiza el contenido de la celda correspondiente con el dato recibido y activa una señal de control para indicar su finalización.

7.2.2. Interconexión

Como se mencionó, la memoria principal y la unidad de control deben comunicarse para intercambiar información. Estos circuitos se comunican a través de un medio de transmisión compartido entre ambos que se denomina *Bus del Sistema* .

Bus del Sistema

La información que se necesita transmitir incluye datos desde y hacia la memoria, así como direcciones hacia la memoria, además de las señales de control. Es por esto que el bus del sistema está formado por 3 (tres) buses para cumplir con cada objetivo. Los buses son: **Bus de direcciones**, **Bus de datos** y **Bus de control** (ver figura 7.2).

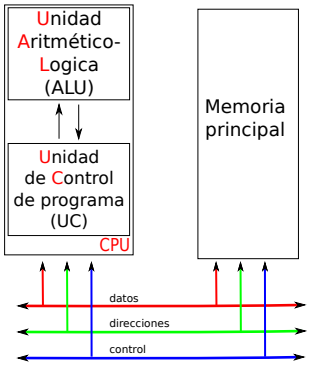


Figura 7.2: Buses del sistema

Cada bus tiene una determinada cantidad de líneas, lo que hace referencia al ancho del bus, y cada línea transmite un bit a la vez. De esta manera, el ancho del bus determina cuántos bits se pueden transmitir en paralelo.

En el caso del **bus de direcciones,** la cantidad de líneas determina el conjunto de direcciones de las celdas de memoria, denominado *espacio direccionable* . Por ejemplo, si se cuenta con m bits para determinar las direcciones de la memoria, entonces la cantidad máxima de combinaciones, y por lo tanto de celdas a direccionar, es 2^m celdas.

Por otro lado, el ancho del **bus de datos** determina el tamaño de las palabras, pues implica la cantidad de información (operandos o instrucciones en código máquina) que puede transmitirse en paralelo.

En cuanto al **bus de control**, éste sólo cuenta con una línea a modo de señal para indicar qué tipo de operación le solicita la UC a la memoria principal (lectura o escritura).

7.2.3. Modos de direccionamiento

Como ya hemos visto, en Q1 se utilizan los registros como variables en un programa. Pero en Q2, estas posibilidades se extienden, dado que también se pueden utlizar las celdas de memoria principal, accesibles a través de sus direcciones, como variables para los operandos de una instrucción. De esta manera, será necesario contar con un mecanismos que permita hacer referencia al operando/s que involucra la celda en cuestión. Así es que se incorpora un nuevo

Estado inicial ⇒			Estado final ⇒		
	...			...	
0x000A	29C8		0x000A	29C7	
0x000B	A0A0		0x000B	A0A0	
	...			...	

Tabla 7.1: Mapas de memoria para `SUB [0x000A], 0x0001`

Estado inicial ⇒			Estado final ⇒		
	...			...	
0x0004	0019		0x0004	0019	
0x0005	4000		0x0005	4019	
	...			...	

Tabla 7.2: Mapas de memoria para `ADD [0x0005], [0x0004]`

modo de direccionamiento, denominado **Modo de direccionamiento directo**. En este modo de direccionamiento, el campo *operando* contiene la dirección efectiva del operando en cuestión.

Este mecanismo fue común en las primeras generaciones de computadoras y se encuentra aún en diversas arquitecturas, pero la limitación más evidente es que el tamaño de dicho campo restringe la cantidad de bits destinados a una dirección y por lo tanto al espacio direccionable. En la sección 7.3 se verá porqué también restringe el tamaño de las constantes.

7.3. Lenguaje Q2: alto nivel

Directo

La sintaxis para indicar el modo de direccionamiento *Directo* , implica escribir entre corchetes la dirección del operando. Por ejemplo: `SUB [0x000A], 0x0001`. El efecto de esta instrucción es el de decrementar en 1 el valor que tenía la celda cuya dirección es `000A`.

mapas de memoria principal

En el cuadro 7.1 se muestran 2 (dos) *mapas de memoria principal* , que identifican sólo un segmento de la misma, para ilustrar el estado inicial (antes de ejecutar la instrucción) y el estado final luego de la ejecución de la instrucción. Es imporante notar que cuando se utiliza el modo de direccionamiento directo en el operando destino, el efecto se denota con la dirección de la celda entre corchetes. En particular en esta instrucción, el efecto se denota: `[000A]` ← `0x29C7`.

Como segundo ejemplo, considerar la instrucción: `ADD [0x0005], [0x0004]`, cuyo resultado se muestra en el cuadro 7.2.

Notar que en el segundo ejemplo, ambos operandos tienen el modo de direccionamiento directo. Por lo que ambos valores se encuentran en la memoria principal, y por tal motivo debemos recurrir a un mapa de memoria para comprender su efecto. En términos sintácticos el efecto se denota: `[0005]` ← `0x4019`. Lo cual resulta de sumar la cadena 0x4000 (valor alojado en la celda con dirección 0x0005) con la cadena 0x0019 (valor alojado en la celda con dirección 0x0004).

7.4. Arquitectura Q2: bajo nivel

Según la especificación de Q2, el modo de direccionamiento directo tiene la codificación `001000`. De esta manera, siguiendo el formato de instrucción correspondiente, el código máquina de la instrucción del primer ejemplo: `SUB [0x000A], 0x0001` es como se indica a continuación:

CodOp	Modo direcc. destino	Modo direcc. origen	Operando destino	Operando origen
0011	001000	000000	0000000000001010	0000000000000001
SUB	DIRECTO	INMEDIATO	000A	0001

7.5. Arquitectura Q2: Registros de la CPU

Para llevar a cabo estas tareas es claro que el procesador debe almacenar algunos datos temporalmente. Por ejemplo será necesario almacenar la posición de la última instrucción, para así determinar dónde buscar la próxima. Además necesita almacenar información temporalmente mientras una instrucción está en proceso de ejecución. En otras palabras, el procesador necesita una pequeña memoria interna.

Como ya se ha mencionado en otros capítulos, el procesador cuenta con un conjunto de registros clasificados en dos tipos. Recordemos cada uno de ellos:

- **Registros de uso general** o visibles a quien programa: facilitan la definición de variables del programa para minimizar las referencias a memoria principal.
- **Registros de uso específico** o de control y estado: Son utilizados por la unidad de control para controlar el funcionamiento del procesador y de programas privilegiados del sistema operativo durante la ejecución de los programas.

Hay diversos registros del procesador que se emplean para controlar su funcionamiento. La mayoría de ellos, en la mayor parte de las máquinas no son visibles al usuario, pero alguno de ellos puede ser visible por ciertas instrucciones máquina ejecutadas en un modo privilegiado o de sistema operativo.

Naturalmente, diferentes arquitecturas tendrán distinta organización de registros y usarán distinta terminología, pero esencialmente se necesitan los siguientes registros para la ejecución de las instrucciones. Los cuales son parte de nuestra arquitectura Q2:

- **Contador del programa (*Program Counter* - PC)**: alnmacena la dirección de la instrucción a buscar/leer.
- **Registro de instrucción (*Instruction Register*- IR)**: almacena la instrucción buscada más recientemente. Muchas veces este registro tiene mas capacidad que los demás, a los fines de *almacenar la instrucción completa*, siendo que una instrucción puede ocupar más de una celda de memoria.

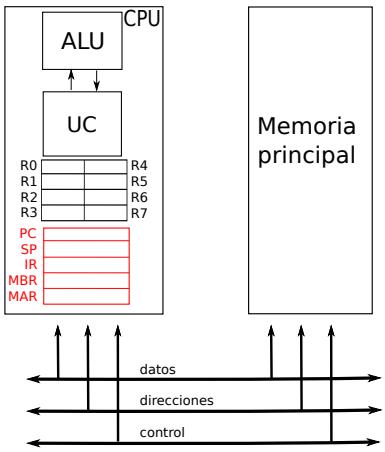


Figura 7.3: Arquitectura Q con detalle de registros de uso específico (en color rojo) y registros de uso general

- **Registro de dirección de memoria (*Memory Address Register* - MAR)**: almacena la dirección de una celda de memoria.
- **Registro amortiguador de memoria (*Memory Buffer Register* - MBR)**: almacena el dato contenido en una celda de memoria leído recientemente, o el dato a escribir en una celda de memoria.
- **Puntero de pila (*Stack Pointer* - SP)**: almacena la dirección del tope de pila (ver apartado 8.2)

No todos los procesadores tienen registros internos designados como MAR o MBR pero se necesita algún mecanismo de almacenamiento intermedio equivalente mediante el cual se de salida a los bits que van a ser transferidos por el bus de sistema. En la figura 7.3 se ven todos los registros de la arquitectura Q2.

Muchos diseños de procesadores incluyen un registro o un conjunto de registros, conocidos a menudo como palabra del estado de programa o *program status word* (PSW), que contiene información de estado. Típicamente contiene códigos de condición, incluyendo a menudo los siguientes bits de estado:

- El signo del resultado de la última operación
- Indicador de resultado cero o nulo
- Indicador de acarreo en la última operación (suma o resta)
- Indicador de igualdad entre operadores
- Indicador de desbordamiento aritmético
- Habilitación de interrupciones
- Indicador de modo supervisor (para permitir ciertas instrucciones privilegiadas)

En algunos diseños es posible encontrar otros registros relativos a estado y control. Puede existir un puntero a bloque de memoria que contenga información de estado adicional (por ejemplo, bloques de control de procesos). En las máquinas que utilizan interrupciones vectorizadas puede existir un registro de vector de interrupciones. Si se utiliza una pila para llevar a cabo ciertas funciones, se necesita un puntero a pila del sistema. En un sistema de memoria virtual se utiliza un puntero a una tabla de páginas. Por último, pueden utilizarse registros para el control de operaciones de E/S. Estos son comparables con el concepto de los flags de la arquitectura Q que se verá mas adelante en el capitulo 9.

7.5.1. Funcionamiento de la memoria principal

Como se dijo en la sección 7.2.1, la memoria principal es un **circuito** formado por **celdas** a las que se acceden a través de los buses para realizar 2 operaciones: lectura y escritura. En esta sección se revisarán dichas operaciones a partir de los registros y los buses mencionados. Como esquemas genéricos podemos analizar las figuras 7.4 y 7.5 respectivamente.

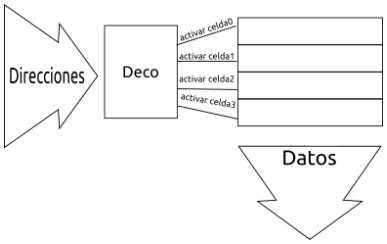


Figura 7.4: Memoria conectada a través de los buses

La operación de **lectura** ocurre cuando la UC necesita leer un dato de una celda de memoria principal. Para esto, la UC almacena en el registro **MAR** la dirección de la celda a leer y luego activa una **señal de lectura** en el **bus de control**, que detallamos como R/$\overline{W}$=1. Luego, la memoria principal activa un circuito decodificador (ver figura 7.5) que permite elegir la celda correspondiente, para luego copiar su contenido en el **bus de datos**. Finalmente, el dato se almacena en el registro **MBR** de la CPU, y mediante el bus de control activará una señal que indica que el dato está disponible.

La operación de **escritura** ocurre cuando la UC necesita almacenar un dato en una determinada celda de memoria principal. Para esto, la UC almacena el dato en el registro **MBR** y su dirección correspondiente en el registro **MAR**. Luego activa una **señal de escritura** en el bus de control para indicar el tipo de operación, que detallamos como R/$\overline{W}$=0. También en este caso, la memoria activa un circuito decodificador que permite elegir la celda correspondiente, en base a la información enviada por los buses (el dato a escribir en el bus de datos, y la dirección donde escribirlo en el bus de direcciones). Finalmente almacena el dato en la celda correspondiente de la memoria. Por último, mediante el bus de control se indica a la UC que la operación ha finalizado.

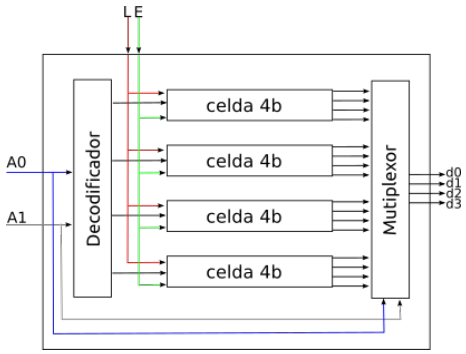


Figura 7.5: Circuito de una memoria de 4 celdas

7.5.2. Ciclo de ejecución de una instrucción (Q2)

En la presente sección veremos el ciclo de ejecución de una instrucción en Q. A partir de extender la arquitectura, donde, además de los registros, es posible utilizar las celdas de la memoria principal como variables en un programa (operandos), será necesario ampliar el ciclo de ejecución provisto en Q1. Para ello, se añaden 2 nuevas etapas: **Búsqueda de operandos (BO)**, y **Almacenamiento de resultado (AR)**. Pero, como no todos los operandos se encuentran en memoria (lo cual depende de su modo de direccionamiento), será necesario evaluar dicha condición para saber si amerita realizar dicha etapa. Así, el ciclo de ejecución de una instrucción en Q2 (y de acá en adelante), será el que se muestra en la figura 7.6.

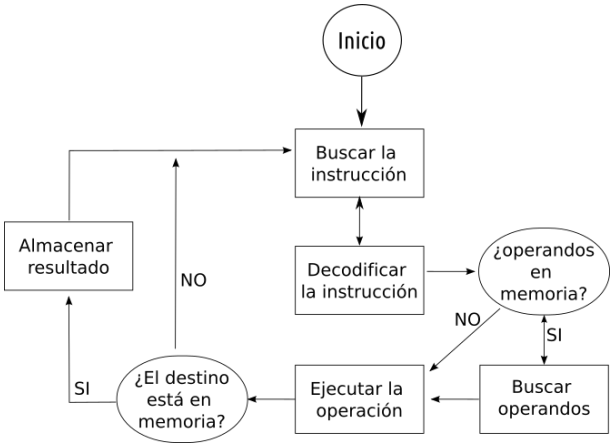


Figura 7.6: Ciclo de ejecución de una instrucción

Para comprender mejor este proceso, será necesario comprender el ciclo teniendo en cuenta los registros de uso específico y los accesos a memoria. A diferencia del proceso mencionado en Q1, se explicará el ciclo de manera completa y generalizada.

Antes de iniciar este proceso, se asume que el registro PC contiene una dirección de memoria. Para lo cual:

1. **Búsqueda de la instrucción (BI):** se solicita la lectura de la celda indicada por PC. Luego, éste se incrementa en 1, quedando preparado para la próxima lectura. Ver cómo intervienen los registros y buses del sistema en la operación de lectura (sección 7.5.1) . Finalmente, cuando el dato está disponible en el MBR, se copia al registro IR (pues se trata de una instrucción).

2. **Decodificación de la instrucción:** en base al formato de instrucción de la especificación de Q, la UC decodifica la cadena almacenada en IR, es decir, que decodifica qué tipo de operación es y dónde buscar sus operandos. Siendo que una instrucción puede ocupar más de una celda, es posible que no baste con una única lectura. Por lo que, se vuelve a la etapa de BI (ver flecha de ida y vuelta entre ambas etapas) concatenando en IR las cadenas correspondientes. Y así sucesivamente hasta tener la instrucción completa en el IR, ocurriendo un mini ciclo entre ambas.

3. **Búsqueda de operandos (BO):** en función de los modos de direccionamiento, la UC evalúa si alguno de los operandos se encuentran en memoria o no. En caso negativo, avanza a la próxima etapa, pero en caso afirmativo, la UC solicita una lectura de la celda correspondiente. Para ello realiza los siguientes pasos:

   a) Copia la dirección del operando en cuestión, que obtiene del IR, al registro MAR.

   b) Envía a la memoria principal la dirección correspondiente por el bus de direcciones, y la señal de lectura por el bus de control: R/$\overline{W}$=1.

   c) Cuando recibe de la memoria principal el dato por el bus de datos, lo copia al registro MBR

4. **Ejecución de la operación:** con la instrucción completa en el IR y sus respectivos operandos, la UC puede ejecutar la instrucción. De tratarse de una operación aritmética o lógica, delega la operación a la ALU, la cual activa el circuito correspondiente para retornar el resultado (salida).

5. **Almacenamiento de resultados (AR):** en base al modo de direccionamiento del operando destino, la UC deberá evaluar si dicho operando debe almacenarse en memoria. En caso negativo, actualiza el registro (de uso general) correspondiente en la CPU y finaliza con este ciclo. En caso afirmativo la UC solicita una operación de escritura a la memoria principal (ver la operación de escritura). Para ello realiza los siguientes pasos:

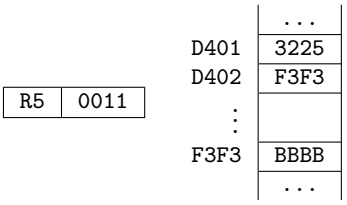
   a) Copia la dirección del operando destino, que obtiene del IR, al registro MAR.

   b) Copia el resultado arrojado por la ALU (salida) en el registro MBR.

   c) Envía a la memoria principal el dato a escribir por el bus de datos, la dirección donde escribir por el bus de direcciones, y la señal de escritura por el bus de control: R/$\overline{W}$=0.

7.5.3. Ejemplo del ciclo de ejecución de una instrucción

A continuación se verá, a modo de ejemplo, cómo se aplica el ciclo de ejecución de una instrucción específica, a partir de un mapa de memoria. En base a la explicación de la sección anterior, se mencionarán las cadenas resultantes que intervienen en cada etapa. El mapa de memoria es el siguiente:



La instrucción está ensamblada a partir de la celda con dirección `0xD401`, por lo que `PC=D401`. El estado inicial de las celdas de memoria y el registro R5 es el que se muestra en el mapa, y a partir del cual aplicaremos el proceso del ciclo de ejecución de una instrucción.

1. **Búsqueda de instrucción (BI)**: se solicita la lectura de la celda indicada por `PC=D401`, por lo que se copia a MAR, quedando `MAR = D401`, así que el bus de dirección transporta la dirección D401, y por el bus de control viaja la señal R/$\overline{W}$=1. Se actualiza `PC=D402`. Finalmente, la memoria envía por el bus de datos, la cadena 3225. Por lo tanto queda `IR=3225`.
   A modo de resumen, iremos registrando en una bitácora los accesos a memoria principal, detallando la información que viaja por los buses en cada etapa:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
BI	R/$\overline{W}$=1	D401	3225

2. **Decodificación de la instrucción**: en base al formato de instrucción, la UC decodifica la cadena almacenada en `IR=3225`. Obteniendo que:

Codop(4b)	Modo direc. Destino (6b)	Modo direc. Origen(6b)
0011	001000	100101

   La UC decodifica que el modo de direccionamiento del operando destino es **Directo**, por lo que falta la lectura de otra celda para completar la instrucción.
   A los fines de comprensión de nuestro ejemplo, es importante notar que, hasta el momento, el código máquina corresponde a la instrucción (en código fuente): `SUB [0x????], R5`.
   Notar que no es necesario actualizar la bitácora, dado que en esta etapa no se accede a memoria.

3. **Completar la instrucción - BI**: se solicita la lectura de la celda indicada por `PC=D402`, se copia la dirección al MAR, quedando `MAR=D402`. Por el bus de direcciones viaja la cadena D402, y por el bus de control la señal R/$\overline{W}$=1. Se actualiza `PC=D403`. Se recibe por el bus de datos la cadena F3F3, quedando finalmente `IR=3225F3F3`.

Se actualiza la bitácora, dado que se accedió a memoria para completar la instrucción:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
BI	R/$\overline{W}$=1	D402	F3F3

4. **Completar la decodificación**: con esta nueva lectura puede concluirse que:

Codop	Modo Direc. Dest	Modo Direc. Origen	Operando Dest(16b)
0011	001000	100101	1111001111110011

De esta manera la instrucción completa en código fuente es:
`SUB [0xF3F3], R5`

5. **Búsqueda de operandos (BO):** se evalúan los operandos en memoria. Como el operando destino se encuentra en memoria, pues tiene modo de direccionamiento directo, la UC solicita la lectura de la celda F3F3. Para ello realiza:

   a) Copia la dirección F3F3 del registro IR al registro `MAR=F3F3`.

   b) Envía la dirección a la memoria, por lo que el bus de direcciones transporta la cadena F3F3 y la señal R/$\overline{W}$=1 por el bus de control.

   c) Desde la memoria, recibe por el bus de datos, la cadena BBBB y la copia en el MBR, quedando `MBR=BBBB`.

   Se actualiza la bitácora con el acceso a memoria por la búsqueda del operando destino:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
BO	R/$\overline{W}$=1	F3F3	BBBB

6. **Ejecución de la instrucción**: como se trata de una resta, la UC le delega la operación a la ALU, activandose el circuito Restador, al cual le suministra los dos operandos:

   a) Operando destino: la cadena `BBBB` (desde el registro MBR)

   b) Operando origen: la cadena `0011` (desde el registro R5)

   Así es que se realiza la resta entre ambos operandos, cuyo cálculo auxiliar es: `0xBBBB - 0x0011 = 0xBBAA`.

   No se actualiza la bitácora, puesto que en dicha etapa no se accede a memoria principal.

7. **Almacenamiento de resultado (AR)**: se evalúa el modo de direccionamiento del operando destino, que en este caso es un directo, por lo que será necesario acceder a la memoria para almacenar el resultado. Para ello realiza:

a) Copia la dirección F3F3 del registro IR al MAR, quedando `MAR=F3F3`

b) Copia el resultado arrojado por la ALU en el registro MBR, quedando `MBR=BBAA`

c) Envía a la memoria el dato a escribir, quedando en el bus de datos la cadena BBAA, en el bus de direcciones la cadena F3F3, y la señal de escritura R/$\overline{W}$=0 en el bus de control.

Se actualiza la bitácora con el acceso correspondiente a la operación de escritura del resultado de la resta:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
AR	R/$\overline{W}$=0	F3F3	BBAA

8. Comienza un nuevo ciclo con la siguiente instrucción indicada por PC

A modo de resumen general, en la bitácora a continuación se muestran todos los accesos realizados a memoria principal:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
BI	R/$\overline{W}$=1	D401	3225
BI	R/$\overline{W}$=1	D402	F3F3
BO	R/$\overline{W}$=1	F3F3	BBBB
AR	R/$\overline{W}$=0	F3F3	BBAA

Esta **bitácora** se utilizará para detallar el contenido de los buses en cada acceso a memoria principal (operación de lectura o escritura) al llevar a cabo el ciclo de ejecución de una instrucción.

Esta bitácora tiene como propósito, detallar los accesos a memoria principal para elaborar conclusiones sobre las características de los programas e incluso, por ejemplo, para analizar su eficiencia. Así es que, en la bitácora del ejemplo, podemos concluir que la instrucción ocupa 2 celdas (pues hay 2 BI) y que se alteró un dato de una de las celdas (pues hay un AR). Así como también que esta única instrucción, requiere un total de 4 accesos a memoria, donde 3 son de lectura y 1 de escritura.

Capítulo 8

Q3: Rutinas

8.1. Modularización y reuso

A menudo un problema complejo se transforma en algo mas abordable si se lo descompone en problemas más pequeños y simples. Esta descomposición recibe el nombre de **modularización**. Esta técnica es muy utilizada al momento de programar, dado que nos permite descomponer el problema en pequeños programas que resuelven una parte específica.

Como parte del lenguaje de programación Q, se denomina **rutina** a un programa que tiene una estructura específica y que se espera usar más de una vez, aprovechando así la modularización. Por lo que una rutina puede contener **subrutinas**, que son rutinas contenidas, como parte de la descomposición del problema (una sub parte).

Modularizar un programa permite:

- Resolver problemas pequeños, simples e independientes (cada uno una parte específica), como una porción del problema grande o complejo.
- Si hay problemas que comparten las mismas soluciones, se puede **reutilizar** dicha solución.
- En caso de cometer un error, es más sencillo detectarlo y corregirlo en un problema simple y pequeño, que dentro de todo el problema grande o complejo.

A modo de resumen, cuando se tiene un problema grande o complejo conviene definir una rutina modularizada que invoque `n` subrutinas (en el orden correspondiente) para que cada una resuelva, de manera independiente, una porción específica del problema, y que en su totalidad resuelvan el problema general. Para ello, será necesario definir cada subrutina como una rutina independiente.

Así, el uso de rutinas requiere llevar a cabo dos tareas, el **encapsulamiento** y la **invocación**.

El **encapsulamiento** implica definir el código que cumple con la función de la rutina, teniendo en cuenta el alcance de la misma (que debe ser acotado), para lo cual será necesario analizar y definir qué parte del problema general resuelve. El alcance de la rutina está delimitado por un inicio y un fin. Se inicia con una **etiqueta** que la identifique inequívocamente (su nombre), ubicada en la primera instrucción, de manera tal que la rutina pueda ser invocada mediante la misma. Mientras que para finalizarla, debemos incluir como última instrucción, la instrucción especial `RET`, de manera que al finalizar su ejecución vuelva a la rutina que la invocó.

Veamos un ejemplo de un problema muy sencillo. Se desea incrementar en uno el valor del registro R0. Para ello definimos la rutina `inc` como:

```
1 inc: ADD R0, 0x0001
2        RET
```

La rutina está delimitada, en su inicio por la etiqueta `inc`, para que al momento de su invocación se pueda utilizar dicho nombre, y en su fin, con la instrucción `RET`.

Rutina

La **invocación** implica *"llamar"* a la rutina (subrutina dentro de la rutina principal), para que resuelva la parte del problema correspondiente. De esta manera, en el lugar desde donde se la invoca, se ejecutará la porción de código correspondiente. La instrucción que utilizaremos para invocar a una rutina es `CALL`, a la cual le debemos agregar la etiqueta de la rutina en cuestión.

Siguiendo con el ejemplo anterior, a continuación se muestra cómo invocar a la rutina `inc`:

```
1 CALL inc
```

Al ensamblarse, las etiquetas se traducen en inmediatos

La etiqueta `inc` se traducirá con el modo de direccionamiento `inmediato` cuyo valor será la dirección de memoria a partir de la cual está ensamblada la rutina, así, su primera instrucción será la siguiente a ejecutar.
Por ejemplo, si la rutina `inc` se encuentra ensamblada a partir de la celda con dirección `0x568A`, la invocación anterior equivale a la siguiente:

```
1 CALL 0x568A
```

A continuación veremos un ejemplo de reuso. Si necesitamos incrementar 3 veces un valor, y sabemos que contamos con una rutina que resuelve dicho problema, pero para un problema más chico (incrementa 1 vez), es conveniente que la rutina a definir, invoque 3 veces a la rutina que ya tenemos definida. Así, la rutina que llamaremos `agregar`, invocará 3 veces a la subrutina `inc`, aprovechando de esta manera, el concepto de **reuso** y **modularización**. Veamos:

```
1 agregar: CALL inc
2          CALL inc
3          CALL inc
4          RET
```

8.1.1. Generalización de rutinas: pasaje de parámetros

Considerar la siguiente rutina `promEntre20y30`, que calcula el promedio entre los valores 20 y 30, y cuyas cadenas en hexadecimal son 0x0014 y 0x001E respectivamente:

```
1 PromEntre20y30: MOV R0, 0x0014
2                 ADD R0, 0x001E
3                 DIV R0, 0x0002
4                 RET
```

La rutina principal, también llamada rutina cliente, que utiliza esta rutina, podría ser:

```
1 ActualizarProm: CALL PromEntre20y30
2                 MOV R3, R0
3                 RET
```

Se ve claramente que la rutina `promEntre20y30` sólo sirve para calcular específicamente el promedio de los valores 20 y 30, por lo que, en caso de querer un promedio con valores diferentes, deberíamos definir otra rutina con dichos valores, y así sucesivamente. Por lo tanto, se puede concluir que esta manera de programar no es práctica. Necesitamos una rutina que nos permita calcular el promedio de cualquier par de valores, generalizando así la solución del problema. Para lograr esto, la rutina debe ser **parametrizable**, es decir, contar con **parámetros** para suministrar los valores que se necesiten cada vez.

Haciendo una analogía con las matemáticas, lo que se busca hacer es pasar de una expresión tal como:

$$x = \frac{20 + 30}{2}$$

a otra tal que:

$$f_{Prom}(a, b) = \frac{a + b}{2}$$

donde `a` y `b` son los argumentos de la función f_{Prom}

Se denomina **pasaje de parámetros** al hecho de establecer valores específicos a variables (en Q son registros o celdas), que serán utilizados dentro de la rutina invocada. Así, la rutina que recibe dichos parámetros (variables), los utilizará para realizar el cálculo necesario. Volviendo a la analogía con el lenguaje matemático, el pasaje de parámetros es la evaluación de la función con valores específicos: $f_{Prom}(20, 30)$.

En el lenguaje Q3, el pasaje de parámetros no cuenta con una sintaxis específica, sino que cada rutina debe informar qué variables se usan como parámetros.

Suponer ahora que la rutina recibe los valores a promediar en los registros R1 y R2, definiendo de esta manera una versión general de la misma (para cualquier par de valores). Por lo tanto, ahora la vamos a nombrar simplemente, `Prom`, con el siguiente código:

```
1 Prom: MOV R0, R1
2       ADD R0, R2
3       DIV R0, 0x0002
4       RET
```

Notar que la cadena 0x0002 no se pasa como parámetro, dado que para calcular un promedio entre 2 valores, siempre se deberá dividir por 2, siendo éste una constante.

Tener en cuenta que, en este algoritmo se está utilizando la variable de retorno (R0) como variable auxiliar, para realizar los cálculos temporales, de manera de no alterar el valor original de los parámetros a promediar (R1 y R2).

Finalmente se debe actualizar el código de la rutina `ActualizarProm`, para que siga resolviendo el problema original (el promedio entre 20 y 30), pero que invoque a la rutina generalizada `Prom`. Así es que, antes de su invocación, será necesario pasarle los parámetros correspondientes para que pueda seguir cumpliendo con su propósito. La rutina queda de la siguiente manera:

```
1 ActualizarProm: MOV R1, 0x0014   //1er Parametro (valor 20)
2                 MOV R2, 0x001E  //2do Parametro (valor 30)
3                 CALL Prom       //Calcula el promedio
4                 MOV R1, R0      //Actualiza el promedio
5                 RET
```

8.1.2. Documentación de las rutinas

Para usar apropiadamente cualquier rutina es importante contar con información sobre qué hace, qué parámetros necesita, cómo afecta al cambio de estado (registros, celdas de memoria, etc) y de qué manera dispone el o los resultados. Esa información se conoce con el nombre de **documentación de la rutina** y se compone de tres campos:

- **Requiere**: se utiliza para informar los parámetros (valores de entrada) que requiere la rutina al ejecutarse, indicando la/s variable/s (registro o celda) y qué información almacena cada una.
- **Retorna**: se utiliza para informar en qué variable se retorna el/los resultado/s (valores de salida) y en qué consiste cada uno.
- **Modifica**: se utiliza para informar qué variables quedan modificadas previniendo posibles *efectos colaterales* post ejecución de la rutina. Excepto la variable en la cual se retorna, dado que se menciona en el campo correspondiente, y por lo cual lógicamente se va a modificar. De esta manera evitamos ser redundantes.

Si son usados adecuadamente, los campos Modifica y Retorna deben ser disjuntos, pues el primero sólo se utiliza para informar cuando la rutina necesitó utilizar *variables adicionales* para el cálculo del resultado esperado.

La documentación permite que cualquier persona pueda invocar a una rutina sólo conociendo su etiqueta (nombre) y su documentación, sin necesidad de conocer su código fuente (encapsulamiento).

La documentación de una rutina es un **contrato** entre quien la programa (servicio) y quienes la utilizan (cliente). Esto implica que el servicio se compromete a que el cliente obtenga los resultados esperados (según modifica y

retorna), si las especificaciones escritas en la documentación son respetadas a la hora de invocar la rutina (según el requiere).

Siguiendo con el ejemplo del promedio, la documentación que acompaña la rutina podría ser:

Prom	
Requiere	en R1 y R2 dos valores a promediar, donde la suma de ambos se pueda representar en 16 bits
Retorna	en R0 el promedio (entero) entre los valores.
Modifica	–

En este caso, la rutina sólo modifica el registro R0, que es en donde retorna el resultado, por lo que no es necesario agregarlo en el campo `Modifica`, dado que sería redundante.

Recordemos el código de la rutina `ActualizarProm`:

```
1 ActualizarProm: MOV R1, 0x0014   //1er Parametro (valor 20)
2                 MOV R2, 0x001E   //2do Parametro (valor 30)
3                 CALL Prom        //Calcula el promedio
4                 MOV R3, R0       //Actualiza el promedio
5                 RET
```

Ahora veamos su documentación:

ActualizarProm	
Requiere	–
Retorna	en R3 el promedio (entero) entre los valores 20 y 30
Modifica	R0, R1, R2

En este caso, el campo **Requiere** no contiene información, puesto que no hay pasaje de parámetros, ya que `ActualizarProm` no requiere variables a las cuales asignarles valores **previamente a su invocación** (en alguna rutina cliente), como sí es el caso de la subrutina `Prom`. En el campo **Retorna** se informa que el registro R3 será el destino del resultado del promedio. En el campo **Modifica**, se informan las variables R0, R1 y R2. Por un lado R0 se informa por ser una variable modificada por la subrutina `Prom`. Y por otro lado, también se informan los registros R1 y R2 por ser los parámetros de dicha subrutina. Tener en cuenta que R3 no se incluye en el modifica, dado que ya se informa en el campo Retorna.

8.2. Rutinas en bajo nivel: CALL y RET

El objetivo de la programación mediante rutinas es el de abstraer a quien programa, de la ubicación de dichas rutinas en memoria principal a través del encapsulamiento.
La figura 8.1 describe un ejemplo donde una rutina principal (caja de la izquierda) invoca a 2 subrutinas: `rutinaA` y `rutinaB`. A su vez, `rutinaA`, también invoca a una subrutina, `rutinaB`. Por otro lado, en el caso de `rutinaB`, como nos interesa focalizar qué sucede cuando se ejecuta la instrucción `RET`, no se mencionan las instrucciones que se necesitarían previamente.
En ese ejemplo se destacan 2 puntos a considerar.

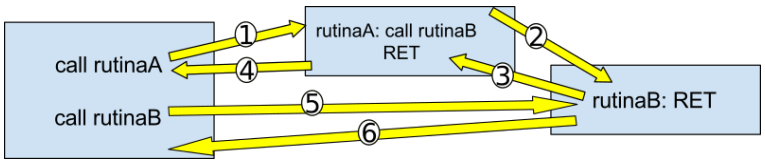


Figura 8.1: flujo de ejecución entre 3 rutinas de ejemplo

1. Cuando una rutina invoca a una subrutina, y ésta a su vez, invoca a otra, se provoca una "anidación de invocaciones" que habrá que manejar de alguna manera para no perder el orden de ejecución correspondiente.

2. Al finalizar las ejecuciones de la rutina `rutinaB` se debe retornar a dos lugares distintos y dependerá de la ubicación de la última instrucción `CALL` que se ejecutó. En la primera ocasión, la rutina `rutinaB` es invocada dentro de rutina `rutinaA` y en la segunda desde la rutina principal, por lo que el punto de retorno varía.

Teniendo en cuenta estas situaciones, será necesario contar con algún mecanismo que le permita a la UC administrar las distintas invocaciones, para conocer cada punto de retorno correspondiente, y como consecuencia, poder ejecutar las instrucciones en el orden en que fueron programadas.

Para entender este mecanismo y como funciona, volvamos al ejemplo anterior y analicemos su ciclo de ejecución.
Pero antes, recordemos que las rutinas son ensambladas y alojadas en memoria por el ensamblador, el cual, no necesariamente lo hace en ubicaciones contiguas de memoria.

Siguiendo el ejemplo anterior, una posibilidad de "mapa de memoria"sería

		...
(rutina principal)	0x9999	CALL
	0x999A	rutinaA
	0x999B	CALL
	0x999C	rutinaB
		⋮
(rutinaA)	0xAAAA	CALL
	0xAAAB	rutinaB
	0xAAAC	RET
		⋮
(rutinaB)	0xBAA0	RET
		...

Sobre este mapa de memoria tener en cuenta 2 cuestiones:
Primero, que las instrucciones no se detallan en código máquina como corresponde, sino que se describen en un lenguaje legible y coloquial con fines explicativos.

Y segundo, que las instrucciones que conforman una rutina, sí se encuentran almacenadas de manera consecutiva, pero no así cada bloque de código correspondiente a las diferentes rutinas. Notar que la `rutinaA` no comienza en la dirección 0x999D, sino en otra cualquiera, como ser 0xAAAA

Como ya sabemos, se utiliza el registro `PC` para ejecutar las instrucciones en el orden correspondiente. Es por ello que, luego de completar la etapa de búsqueda de instrucción, el registro PC queda preparado para la próxima instrucción, pero, al momento de ejecutar la instrucción `CALL`, el valor de PC se altera con la dirección de la celda dónde se encuentra la primera instrucción de la rutina invocada. A este efecto se lo denomina **desvío del flujo**, dado que cambia el flujo (rumbo) de ejecución secuencial que se sostiene en las instrucciones de 2 operandos.
Analicemos cómo sería, a grandes razgos, el ciclo de ejecución para el ejemplo dado:

1. `PC=0x9999`
2. Se busca la instrucción en la celda con dirección 0x9999 y al decodificarla la UC interpreta que se trata de un `CALL rutinaA`. Teniendo en cuenta que la instrucción ocupa dos celdas de memoria (mini ciclo entre la etapa de BI y la Deco), obtenemos que PC se incrementó en 2, quedando `PC=0x999B`, preparado para ejecutar la próxima instrucción de `rutina main`.
3. Como no hay operandos en memoria, se avanza a la etapa de ejecución. Al tratarse de una instrucción `CALL`, se debe desviar el flujo, alterando PC para que apunte a la dirección donde se encuentra la primera instrucción de `rutinaA`, quedando `PC=0xAAAA`. Pero, además, se debe contar con algún mecanismo para **guardar el valor anterior de PC, que era** `0x999B`, y así poder restituir el flujo al retornar de `rutinaA`.
4. Comienza un nuevo ciclo con la búsqueda de la instrucción en la dirección `0xAAAA` apuntada por PC. Al completar la etapa de decodificación, el valor de PC resulta en `PC=0xAAAC`, pues se trata de un `CALL` que ocupa 2 celdas. Así, la UC interpreta que se trata de la instrucción `CALL rutinaB`.
5. Nuevamente no hay operandos en memoria, por lo que se pasa a la etapa de ejecución del `CALL`. Así, nuevamente se debe desviar el flujo, quedando `PC=0xBAA0`. Y se debe **guardar el valor previo de PC**, que era `0xAAAC`.
6. Comienza un nuevo ciclo con la búsqueda de la instrucción en la dirección `0xBAA0` apuntada por PC. En la etapa de decodificación, la UC interpreta que se trata de un `RET`, por lo que el valor de PC se incrementa en 1 (pues ocupa una celda), quedando `PC=0xBAA1`.
7. Se ejecuta la instrucción `RET`, lo que implica que finalizó `rutinaB`. Ahora, entonces, será necesario retornar a la instrucción que continúa según el orden programado. Por lo que será necesario utilizar el mecanismo de guardado que fuimos aplicando en los ciclos anteriores. Así es que **se altera PC con el último valor guardado**, restituyendo a `PC=0xAAAC`.
8. Comienza un nuevo ciclo con la búsqueda de la instrucción en la dirección `0xAAAC` apuntada por PC. En la etapa de decodificación, la UC interpreta

que se trata nuevamente de un `RET`, por lo que el valor de PC se incrementa en 1 (pues ocupa una celda), quedando `PC=0xAAAD`.

9. Se ejecuta la instrucción `RET`, lo que implica que finalizó `rutinaA`, y que habrá que retornar a la próxima instrucción de la rutina que se está ejecutando. Se utiliza el mecanismo de guardado, **alterando PC con el último valor guardado**, así pues `PC=0x999B`.

10. Comienza un nuevo ciclo con la búsqueda de la instrucción en la dirección `0x999B` apuntada por PC. Al completar la etapa de decodificación, el valor de PC resulta en `PC=0x999D`, pues el `CALL` ocupa 2 celdas. Así, la UC interpreta que se trata de la instrucción `CALL rutinaB`.

11. Se ejecuta la instrucción `CALL` (no hay operandos en memoria), por lo que se debe desviar el flujo, quedando `PC=0xBAA0`. Y se debe **guardar el valor previo de PC**, que era `0x999D`.

12. Comienza un nuevo ciclo con la búsqueda de la instrucción en la dirección `0xBAA0` apuntada por PC. En la etapa de decodificación, la UC interpreta que se trata de un `RET`, por lo que el valor de PC se incrementa en 1 (pues ocupa una celda), quedando `PC=0xBAA1`.

13. Se ejecuta la instrucción `RET`, lo que implica que finalizó `rutinaB`. Entonces hay que retornar a la siguiente instrucción utilizando el mecanismo de guardado. Así es que **se altera PC con el último valor guardado**, quedando `PC=0x999D`.

El desafío entonces es darle a la UC un mecanismo que permita llevar cuenta de los valores acumulados de PC para ir restituyéndolos en orden inverso. La solución no puede ser tener un registro de respaldo de PC porque como vemos en los pasos 3 y 5, se necesitan mantener dos valores (y podrían ser aún más) por lo que esta opción no es viable.

8.2.1. La estructura de pila

El desvío del flujo que se produce al ejecutar cada instrucción `CALL`, lleva a la necesidad de contar con una estructura de datos (mecanismo) que permita guardar todos los valores de PC que, posteriormente, se deben restituir con cada `RET`, para que puedan ser consumidos de manera inversa a la que fueron guardados. A esta estructura se la denomina **Pila** y está caracterizada por dos operaciones: **Apilado** (*push*) y **Desapilado** (*pop*).

Consideremos una situación de ejemplo para explicar las operaciones de apilado y desapilado mencionadas.

1. Al comenzar se considera que la pila está vacía, y se denota: {}

2. Luego, al apilar el valor 7 ($push(7)$), la pila queda con ese único valor: {7}

3. Si a continuación se apila el valor 4, el estado de la pila será: $\{4, 7\}$

4. Si a continuación se desapila un valor (siempre es el último apilado), la pila queda: {7}

Este concepto se implementa utilizando **un sector de la memoria principal**, de manera que se reserva un conjunto de direcciones para su administración, iniciando desde la `0xFFEF` y decreciendo en las direcciones de memoria. La celda disponible para apilar un valor es denominada como **tope de pila**, y el registro de uso específico que se usará para indicarlo se denomina **SP** (*Stack pointer*, puntero de pila). Para el caso inicial, donde la pila está vacía, `SP=0xFFEF`. El SP se irá decrementando cuando apila ("sube" por la pila, apuntando a la dirección previa), o se irá incrementando cuando desapila ("baja" por la pila, apuntando a la dirección siguiente).

Con esta idea, es momento de revisar el funcionamiento esperado (efecto) para el `CALL` y el `RET` que se describe en la especificación dela arquitectura, teniendo en cuenta que con cada `CALL` se debe **Apilar** y con cada `RET` se debe **Desapilar**.

Para el caso de la instrucción `CALL`, se necesita apilar el valor que tiene PC justo antes del desvío del flujo, y luego reemplazar PC con la dirección de inicio de la rutina que se está invocando. Particulamente, se realizan los siguientes pasos:

(I) Se copia el valor de PC en la celda con dirección indicada por **SP**, quedando `[SP]<- PC` (apila)

(II) Se decrementa **SP** (hay un elemento más en la pila), por lo que habrá que "subir" en la pila. Quedando `SP <- (SP-1)`

(III) Se actualiza el valor de PC con la dirección de inicio de la rutina, por lo que `PC <- Origen` (operando origen).

Para el caso de la instrucción `RET` se necesita desapilar el último valor apilado de PC. En particular se realizan los siguientes pasos:

(I) Se incrementa **SP** (hay un elemento menos en la pila), por lo tanto habrá que "bajar" en la pila. Quedando: `SP <- (SP+1)`

(II) Se copia a PC el valor de la celda indicada por **SP**, quedando `PC <- [SP]` (desapila)

En la Figura 8.2 se explica gráficamente los cambios que ocurren en la pila durante la ejecución de la rutina principal (de la sección anterior 8.2) que invoca a las subrutinas (`rutinaA` y `rutinaB`).

8.3. Simulación revisada de la ejecución

Durante la ejecución de un programa, además de los cambios en los registros y en las celdas de memoria (como se explicó en el apartado 7.5.3), se realizan cambios, que dependerán de las instrucciones, en los registros de uso específico PC, IR y SP, así como también en la pila.

Por ejemplo, suponer la ejecución del siguiente bloque de código que está ensamblado a partir de la celda con dirección `0x0000`:

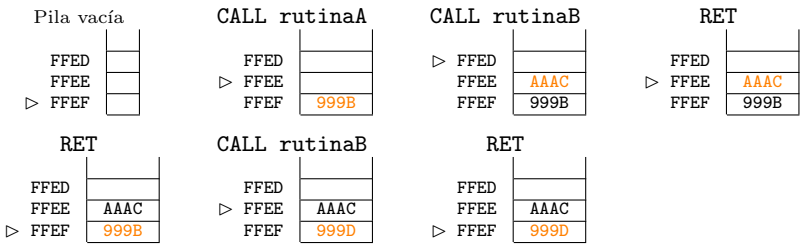


Figura 8.2: Cambios en la memoria principal durante la ejecución de `rutina principal`. En **naranja** se indica qué celda fue leida o escrita según el caso.

```
1 MOV  [0x4567], 0x0008
2 MOV  [0x5678], 0x000A
3 CALL promedio
```

También se asume que la rutina **promedio** esta ensamblada a partir de la celda con dirección `0x0123`, siendo el siguiente su código fuente:

```
1 promedio: MOV R3, [0x4567]
2           ADD R3, [0x5678]
3           DIV R3, 0x0002
4           RET
```

Además, se considera que el valor inicial del registro `PC=0x0000` y que la pila está vacía. Para seguir, paso a paso los cambios en los registros de uso específico y la pila, se utilizará la siguiente tabla, la cual resumirá dichos cambios por cada etapa del ciclo de ejecución de cada instrucción. Notar que en esta ocasión pondremos el foco sólo en los registros PC, IR y SP, dejando de lado MAR y MBR.

Búsq. de inst.			Decod. de inst.			Búsq. de Op.		Ejecución			Alm. res.
PC	IR	PC-bi	operación	modo destino	modo origen	Destino	Origen	Pila	SP	PC-ex	

Donde **PC-bi** es el valor de PC luego de la *búsqueda de instrucción*, **PC-ex** es el valor de PC y **SP** es el valor del registro SP luego de la etapa de *ejecución de la operación*, y la **Pila** es la secuencia de valores apilados (en orden invertido con respecto a su inserción en la pila).

El primer ciclo de la simulación considera:

- **Búsqueda de instrucción:** PC comienza con el valor `0x0000`. IR contiene el código máquina de la instrucción buscada. En este caso, la primera instrucción de la rutina es un `MOV [0x4567], 0x0008`, que ocupa 48 bits (3 celdas), y el código máquina correspondiente es `1800 4567 0008`. Inmediatamente después de la búsqueda de instrucción, se incrementa PC con la dirección de la siguiente instrucción a ejecutar, quedando `PC = 0x0003`, pues como se puede observar en el registro IR, la instruccion ocupa 3 celdas.

- **Decodificación:** se identifica la operación a ejecutar y sobre qué operandos. En este caso se trata de un MOV, que es una instrucción de dos

Búsq. de inst.			Decod. de inst.			Búsq. de Op.		Ejecución			Alm. res.
PC	IR	PC-bi	operación	modo destino	modo origen	Destino	Origen	Pila	SP	PC-ex	
0000	1200 4567 0008	0003	MOV	Dir	Inm	[4567]	0008	{}	FFEF	0003	[4567]<-0008
0003	1200 5678 000A	0006	MOV	Dir	Inm	[5678]	000A	{}	FFEF	0006	[5678]<-000A
0006	B000 0123	0008	CALL	--	Inm	--	0123	{0008}	FFEE	0123	--
0123	18C8 4567	0125	MOV	Reg	Dir	R3	[4567]	{0008}	FFEE	0125	R3 <-0008
0125	28C8 5678	0127	ADD	Reg	Dir	R3	[5678]	{0008}	FFEE	0127	R3 <-0012
0127	78C0 0002	0129	DIV	Reg	Inm	R3	0002	{0008}	FFEE	0129	R3 <-0009
0129	C000	012A	RET	--	--	--	--	{}	FFEF	0008	--

Tabla 8.1: Tabla de simulación de la rutina

operandos, cuyos modos son directo para el destino e inmediato para el origen.

- **Búsqueda de Operandos:** el operando destino es la celda `4567`, y el operando origen es el valor inmediato `0008`. Esta situación provoca un único acceso de lectura a memoria por el operando destino.
- **Ejecución de la operación:** la instrucción `MOV` no tiene efecto adicional. Es decir que no altera SP, y por ende, tampoco la pila.
- **Almacenamiento de resultado:** el almacenamiento del resultado en la celda `4567` se describe con la notación de efecto: `[4567]<-0008`

Esta información se incluye en la primera fila de la tabla como se muestra en la tabla 8.1.
Es importante notar que las columnas **PC-ex**, **SP** y **pila** se modifican sólo en los casos de instrucciones como `CALL` y `RET` que son las que modifican dichos registros.

La ejecución de las siguientes instrucciones de este ejemplo que se agregan al cuadro, aplican el mismo análisis detallado para la primera instrucción.

Simulación: ejemplo 2

Como segundo ejemplo suponer el siguiente bloque de código que se encuentra ensamblado a partir de la celda `0008`:

```
1
2  CALL rutinaA
3  CALL rutinaB
4
5  rutinaB: RET
6
7  rutinaA: CALL rutinaB
8           RET
```

También se asume que **rutinaA** está ensamblada a partir de la celda con dirección `0x0123`, y **rutinaB** está ensamblada a partir de la celda con dirección `0x0678`. Por último, considerando que el valor inicial del registro `PC = 0x0008` y que la pila está vacía. Así, la primera entrada de la tabla se completa de la siguiente manera:

- **Búsqueda de la instrucción:** PC comienza con el valor `0x0008`. IR contiene el código máquina de la instrucción buscada: `CALL rutinaA`, que ocupa 32 bits (2 celdas), y el código máquina correspondiente es `IR = B000 0123`. Luego de la búsqueda de instrucción, `PC = 0x000A`.
- **Decodificación:** se identifica la operación a realizar y sobre qué operandos. En este caso es un `CALL`, que se trata de una instrucción de un sólo operando origen, dado que el operando destino es el registro PC, y por lo tanto está implícito (la instrucción no debe indicarlo).
- **Búsqueda de operandos:** el operando origen en este caso, será el nuevo valor que tendrá PC , quedando `PC = 0x0123`, y se deduce del valor de la etiqueta `rutinaA`.
- **Ejecución de la operación:** tal como indica el **efecto de la instrucción CALL**, se tiene que: `[SP]<- PC; SP<-SP-1; PC<-Origen`, se llevan a cabo 3 pasos:

  1. Apilar PC: la pila debe almacenar el valor de retorno para PC, es decir el valor de PC luego de la BI.
  2. Decrementar SP: el valor del registro SP se actualiza para que apunte a una celda disponible, es decir que toma el valor `0xFFEE`.
  3. Actualizar PC: finalmente, el valor de PC luego de la ejecución, se carga con el valor `Origen`, y como se trata de una etiqueta (modo de direccionamiento inmediato), dicho valor se obtiene del código máquina de la instrucción, es decir del operando que está en el IR.

- **Almacenamiento de resultado:** dado que no se tiene un operando destino en memora (o este está implícito y es el regisro PC), entonces esta etapa no se lleva a cabo.

Estos datos se escriben en la primera fila de la tabla como se muestra en el cuadro 8.2. En la siguiente instrucción `CALL rutinaB` se desvía nuevamente el flujo y se apila otro valor de retorno para PC, como se muestra en la segunda fila de la tabla. En la tercera instrucción `RET`, se restituye el flujo a la última rutina que ejecutó un `RET`, en este caso, se trata de `rutinaA` y para ello, es necesario desapilar el tope de la pila y copiarlo en PC (ver columna PC-ex), como se muestra en la tercera fila de la tabla.

Para finalizar la simulación, se repite el proceso descripto con el resto de las instrucciones en el orden del **flujo de ejecución**, incluyendo la invocación a subrutinas (y sus instrucciones) hasta la última instrucción del programa principal. La tabla completa con la simulación de ejecución del programa puede verse en la tabla 8.2.

Búsq. de inst.			Decod. de inst.			Búsq. de Op.		Ejecución			Alm. res.
PC	IR	PC-bi	operación	modo destino	modo origen	Destino	Origen	Pila	SP	PC-ex	
0008	B0000123	000A	CALL	--	Inm	--	0123	{000A}	FFEE	0123	--
0123	B0000678	0125	CALL	--	Inm	--	0678	{000A,0125}	FFED	0678	--
0678	C000	0679	RET	--	---	--	---	{000A}	FFEE	0125	--
0125	C000	0126	RET	--	---	--	---	{}	FFEF	000A	--
000A	B0000678	000C	CALL	--	Inm	--	0678	{000C}	FFEE	0678	--
0678	C000	0679	RET	--	---	--	---	{}	FFEF	000C	--

Tabla 8.2: Simulación completa del bloque de código dado

Capítulo 9

Q4: estructura condicional

9.1. Introducción y motivación

En la tarea de programación puede surgir la necesidad de bifurcar el flujo de ejecución del programa, es decir, ejecutar una acción u otra, dependiendo de una condición dada.

Exceptuando la invocación de subrutinas, las rutinas que construimos hasta ahora se ejecutan de manera secuencial, donde cada instrucción es leída según su dirección en la memoria y, como ya sabemos, la posición de memoria a leer es la que almacena el registro PC. También se sabe que el valor de PC se incrementa en función del tamaño de cada instrucción, por lo que si se quiere evitar que la próxima instrucción a ejecutar sea siempre la misma, se necesita un mecanismo que permita modificar el valor del PC de manera condicional. Este mecanismo está dado por las instrucciones conocidas como **Saltos**.

Una condición que debe poder expresarse en cualquier lenguaje es la **condición por igualdad** entre los valores de dos variables o entre una variable y una constante.

Por ejemplo, suponer que se desea almacenar el valor 10 en el registro R6, pero sólo si el contenido de R0 es igual a 1. Así podemos expresar el algoritmo en `pseudo-código` (mezcla de sintaxis entre el lenguaje natural y un lenguaje de programación, en nuestro caso el lenguaje Q). Este tipo de expresión nos permite analizar bien el problema para poder encarar el programa que lo resuelva. Veamos:

Problema de ejemplo en pseudo-código

```
Si (R0 = 1) entonces
    R6 <- 0x000A (efecto)
finalizar
```

Lo anterior puede expresarse en términos de un lenguaje ensamblador como sigue:

```
      (Comparar el contenido de R0 con 1)
      {si es igual --> MOV R6, 0x000A}
      {finalizar}
```

Q3 no alcanza para resolver este tipo de algoritmos, es por eso que la arquitectura y el lenguaje `Q` propone una nueva versión: `Q4`, el cual incluye varias **instrucciones de saltos** y una instrucción de **comparación**.

Instrucción	Descripción
JE	Igual / Cero
JNE	No igual
JLE	Menor o igual
JG	Mayor
JL	Menor
JGE	Mayor o igual
JLEU	Menor o igual sin signo
JGU	Mayor sin signo
JCS	Carry / Menor sin signo
JNEG	Negativo
JVS	Overflow

Tabla 9.1: Saltos condicionales

Los **Saltos** son instrucciones que permiten **desviar el flujo**, modificando el valor del `PC` para que la rutina continúe su ejecución en una instrucción distinta a la siguiente posición de memoria. La instrucción donde se desea continuar la ejecución, se debe señalar con una etiqueta. Es decir que cada salto se acompaña con una etiqueta que comunica la acción a ejecutar al realizar el salto, lo cual enriquece la legibilidad del código.

Los saltos disponibles se clasifican en términos de alto nivel como **condicionales o incondicionales**:

- **Saltos Condicionales:** el desvío del flujo se lleva a cabo durante la ejecución del salto, **solo si se cumple una determinada condición**. Es decir que dependen directamente de una condición.
- **Saltos Incondicionales:** el desvío del flujo se lleva a cabo siempre que se ejecute el salto. Es decir que no dependen de una condición.

Las instrucciones que incorpora Q4 para los saltos condicionales son **JE, JNE, JLE, JG, JL, JGE, JLEU, JGU, JCS, JNEG** y **JVS**. Cada una de estas evalúa una condición distinta (ver el cuadro 9.1) y tienen la siguiente sintaxis: `Jxx etiqueta`. En términos de arquitectura (bajo nivel), la intención es modificar el PC si la condición del salto se cumple, de lo contrario no tiene efecto alguno.

La instrucción para el salto incondicional es **JMP** y tiene la sintaxis: `JMP etiqueta` y al ejecutarse, la UC almacena en PC la dirección de memoria donde se encuentra dicha `etiqueta`.

Tener en cuenta que si bien los saltos desvían el flujo, al igual que la instrucción `CALL`, el desvío se realiza **dentro del alcance de la rutina**, es decir dentro de los límites que la definen (su etiqueta y el RET). Mientras que el `CALL` desvia el flujo fuera del alcance de la rutina, invocando a una rutina diferente (subrutina) que tiene su propia definición (etiqueta y RET).

Volviendo al ejemplo anterior, el uso de las instrucciones queda de la siguiente manera:

```
                    (Comparar el contenido de R0 con 1)
                    si es igual ---> asignarValor
                    saltar a salir
asignarValor ---> MOV R6, 0x000A
                    salir
```

A partir de este análisis podremos ir definiendo la rutina en pseudo-código de la siguiente manera:

```
  asignarValor10: Si (R0 es igual a 1) entonces
                  JE asignarValor
                  JMP fin
    asignarValor: MOV R6, 0x000A
                  fin: RET
```

Luego de comparar, que en este caso se trata de una igualdad, debemos utilizar el salto condicional `JE` para continuar con la instrucción correspondiente a dicha bifurcación (la asignación del valor 10 a R6). Pero también se debe contemplar el caso en que R0 no sea igual a 1, mediante una instrucción que permita finalizar la rutina. Es por este motivo es que se utiliza el salto incondicional `JMP fin`, ya que no necesitamos evaluar una condición, sino, simplemente finalizar, ejecutando el `RET`.

Analicemos ambos casos:

- **Caso verdadero:** R0 = 0x0001, por lo que al ejecutarse el salto, la evaluación del `JE` evalúa verdadero, así que desvía el flujo, saltando a la instrucción correspondiente a la etiqueta `asignarValor`, en este caso el `MOV R6, 0x000A`. Luego, continúa el fiujo secuencial y ejecuta la instrucción `RET`, finalizando así la rutina. Efecto final: `R6 <- 0x000A`. Notar que no se ejecutó el salto incondicional (`JMP`).
- **Caso falso:** R0 = 0x0002, por lo que ejecutarse el salto, la evaluación del `JE` evalúa falso, así que no se devía el flujo, y se ejecuta la siguiente instrucción: el `JMP fin`. Al ejecutarse, salta directamente a la instrucción que se encuentra junto a la etiqueta fin, es decir el `RET`. Efecto: R6 no se altera. Notar que no se ejecutó la instrucción `MOV R6, 0x000A`.

Por último, queda resolver qué instrucción utilizar para realizar la comparación. Así es que, Q4 incorpora una instrucción que permite comparar dos valores de 16 bits para, a posteriori, se pueda concluir si son iguales o no.
La instrucción que se encarga de esta tarea se denomina **CMP**, la cual se incluye entre las **instrucciones de dos operandos**, y cuyo objetivo es comparar ambos operandos.

Para comprender bien cómo funciona la instrucción `CMP`, vamos a expresar la igualdad en términos de una resta:

$$A = B \iff A - B = 0$$

De esta manera, el problema se reduce a indicar si el resultado es cero.

Así es que, el `CMP` tiene como objetivo comparar dos operandos por medio de una resta, previo a la ejecución de un salto condicional. Es decir que ambas instrucciones trabajan en conjunto. El efecto de esta instrucción es restar

el operando destino con el operando origen. Pero cabe destacar que la resta, se lleva a cabo sólo a los fines de comparar, es por esto que su resultado se descarta y **no se modifica el destino** (a diferencia de las demás instrucciones de dos operandos, como sucede con la operación `SUB`).

Finalmente la rutina, escrita en el lenguaje Q4, queda de la siguiente manera:

```
1  asignarValor10: CMP R0, 0x0001
2                  JE asignarValor
3                  JMP fin
4    asignarValor: MOV R6, 0x000A
5                  fin: RET
```

En este ejemplo, la comparación que se realiza es el contenido del registro R0 con la cadena 0x0001, que representa al valor 1. Para los casos mencionados anterioremente realiza:

- **Caso verdadero:** 0x0001 - 0x0001
- **Caso falso:** 0x0001 - 0x0002

9.2. Funcionamiento de los saltos en bajo nivel

A las diferencias ya mencionadas entre el `CALL` y los saltos, es importante destacar que al ejecutarse el salto `JMP`, la UC almacena en el registro `PC`, la dirección de memoria correspondiente a la `etiqueta` que lo acompaña, pero a diferencia del `CALL`, no altera la pila.

Por otro lado, en térmoinos de arquitectura (bajo nivel), los saltos se pueden clasificar de dos manera en función de cómo modifican el registro `PC`:

- **Salto Absoluto:** el nuevo valor del PC se expresa en términos de una dirección de memoria específica (`JMP`).
- **Salto Relativo:** el nuevo valor del PC se expresa en términos de un desplazamiento con respecto a la siguiente instrucción, indicando la cantidad de celdas que debe "saltar".

Ensamblado de CMP

La instrucción `CMP` es una instrucción de dos operandos, tal y como las instrucciones aritméticas que ya conocemos:

Operación	Cod Op	Efecto
CMP	0110	Dest - Origen

Y se ensambla mediante el siguiente formato de instrucción:

Cod_Op (4b)	Modo Destino(6b)	Modo Origen(6b)	Destino(16b)	Origen(16b)

Por ejemplo, se cuenta con la instrucción `CMP R0, 0x4567`. El código máquina resultante es `0110 100000 000000 0100 0101 0110 0111`. Y su versión comprimida en hexadecimal es: `6800 4567`.

Ensamblado del salto incondicional

La instrucción `JMP`, al igual que el `CALL` es una instrucción de **1 operando origen**:

Operación	Cod Op	Efecto
JMP	1010	PC ← Origen

Y su formato de instrucción es el siguiente:

`Cod_Op (4b)`	`Relleno(000000)(6)`	`Modo Origen(6b)`	`Origen(16b)`

El **JMP** es un **salto absoluto**, por lo cual, el valor del **operando origen** debe ser la dirección de memoria donde se desea saltar (siguiente instrucción a ejecutar). Esta dirección en el código fuente, se expresa como una etiqueta, es por esta razón que al igual que el `CALL`, el `JMP` ocupará dos celdas: la primera, contiene el código de operación junto con el relleno y el modo de direccionamiento del origen (en caso de una etiqueta es **inmediato**), y la segunda celda contiene la dirección de memoria a la que se quiere saltar (etiqueta), la cual representa el operando origen de la instrucción.

Ensamblado del saltos condicionales

Los saltos restantes, es decir **los condicionales**, son **saltos relativos**. Este tipo de saltos no reemplazan el valor del PC con la dirección de memoria de la instrucción a donde se quiere bifurcar ("saltar"), sino que se calcula, mediante un desplazamiento, dónde se debe saltar. Así, al valor del PC se le suma el valor del desplazamiento, expresado como un número en $CA2(8)$. Se utiliza **CA2** porque al disponer de números negativos es posible realizar saltos tanto hacia una instrucción posterior como una instrucción anterior.

El formato de instrucción es el siguiente:

`Prefijo (1111)`	CodOp (4)	Desplazamiento(8)

Donde los primeros cuatro bits se corresponden con la cadena binaria `1111`, la cual funciona como prefijo del `CopOp`. Si **al evaluar la condición de salto** (ver Cuadro 9.2) ésta valúa verdadero, se le suma al PC el valor del **desplazamiento**, representado en $CA2(8)$, en caso contrario la instrucción no altera PC.

Así, la ejecución de los saltos condicionales dentro del **ciclo de ejecución**, se hace presente en la etapa de **Ejecución de la operación**. Es decir que la instrucción de salto siempre se ejecuta, y al evaluarse se concluye si se deberá deviar o no el flujo, y por ende alterar o no el registro PC. Ver la Figura 9.1.

Al utilizarse el sistema de numeración $CA2()$ se implica que el valor puede ser positivo o negativo. En el primer caso, permite "avanzar" el PC o mejor dicho desviar el flujo hacia una instrucción mas adelante en el código de la rutina (o posterior en la memoria). En el segundo caso (valores negativos) permite "retroceder" el PC para desviar el flujo hacia una instrucción mas atrás en el código (o anterior en la memoria).

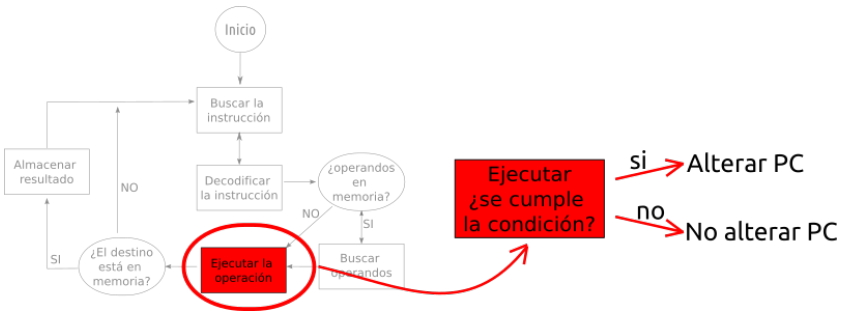


Figura 9.1: Ciclo de ejecución de instrucción

Por ejemplo, suponer la siguiente rutina:

```
1 main:  CMP R0, 0x0001
2        JE fin
3        MOV R5,0x0002
4   fin: RET
```

El código máquina de la instrucción `JE` se encontrará en la tabla 9.2 de códigos de operación de la arquitectura Q.

Para ensamblar la instrucción `JE fin`, primero habrá que calcular su desplazamiento. En este caso, se trata del valor 2, puesto que la instrucción a "saltar"(`MOV R5, 0x0002`) ocupa 2 celda. Por lo tanto, a PC habrá que sumarle dicho valor. Así, el código máquina resulta en: `1111 0001 00000010`.

Nota: la cadena para el desplazamiento se deberá calcular representando dicho valor en CA2(.)

Suponer un segundo ejemplo de rutina (que no resuelve un problema dado), la cual se encuentra ensamblada a partir de la celda con dirección `0x00AA`:

```
1 main:   JMP seguir
2 arriba: JMP fin
3 seguir: CMP R1,R0
4         JE arriba
5         MOV R5, R6
6 fin:    RET
```

El mapa de memoria, sólo a fines explicativos, es:

		...
(main)	0x00AA	JMP
	0x00AB	seguir
	0x00AC	JMP
	0x00AD	fin
	0x00AE	CMP R1, R0
	0x00AF	JE arriba
	0x00B0	MOV R5, R6
	0x00B1	RET
		...

Codop	Op.	Descripción
0001	JE	Igual / Cero
1001	JNE	No igual
0010	JLE	Menor o igual
1010	JG	Mayor
0011	JL	Menor
1011	JGE	Mayor o igual
0100	JLEU	Menor o igual sin signo
1100	JGU	Mayor sin signo
0101	JCS	Carry / Menor sin signo
0110	JNEG	Negativo
0111	JVS	Overflow

Tabla 9.2: Codigo de operación y condiciones de cada salto condicional

El código máquina de la instrucción `JE arriba` consta de un desplazamiento de `-4`. Esto se debe a que al ejecutarse la instrucción (etapa de ejecución), el PC quedó `PC=00B0`, dado que está preparado para buscar la próxima instrucción `MOV R5, R6`. Si la intención es sumarle el desplazamiento (**d**), tal que luego quede `PC=00AC` (donde está la instrucción etiquetada como "arriba"), entonces debe ocurrir que `0x00B0 + Rca2(16)(d) = 0x00AC`.

Veamos el cálculo auxiliar para ensamblar la instrucción:

$$R_{ca2(8)}(-4) = complemento(R_{bss(8)}(4)) = complemento(00000100) =$$

$$= 11111011 + 1 = 11111100$$

.
Nota: tener en cuenta que la representación se realiza con 8 bits, correspondientes al formato de instrucción. Pero para comprobar si es correcto al sumarlo con PC, se debe completar los bits faltantes para lograr una cadena de 16 bits.

Siendo que ya conocemos el desplazamiento, la instrucción queda ensamblada como: `1111 0001 1111 1100` (0xF1FC).

Ahora, con esta idea se debe comprobar si la suma entre la dirección de PC y la cadena del desplazamiento, efectivamente se corresponde con la dirección de la celda a donde se desea "saltar".
Para ello, realizamos el cálculo correspondiente (`0x00B0` + $R_{ca2(16)}(-4)$) y comparamos si se obtuvo la cadena `0x00AC`.

$$\begin{array}{rr} & 0000\ 0000\ 1011\ 0000 \\ + & 1111\ 1111\ 1111\ 1100 \\ \hline & 0000\ 0000\ 1010\ 1100 \end{array}$$

La cadena resultante se comprime en hexadecimal, obteniendo finalmente la cadena deseada `0x00AC`. Así comprobamos que el desplazamiento calculado es correcto.

9.2.1. Cómo se implementa una comparación

Como se dijo en la sección 9.1, a la hora de resolver el desafío de comparar dos valores y concluir si son o no iguales, es posible reformular el cálculo en términos de una resta, aprovechando el circuito restador con el que ya cuenta la arquitectura.
Así, la comparación se resuelve como: $A = B \iff A - B = 0$, de manera que problema se reduce a concluir si el resultado es cero o no.
Con la misma idea de comparar 2 valores por igual, se los puede comparar por mayor (o menor):

$$A < B \iff A - B < 0$$

Donde esta nueva versión del problema se traslada a determinar si el resultado es negativo o no. En un sistema en $CA2(e)$ste problema puede resolverse con una señal que se encienda cuando el bit más significativo del resultado del cálculo $(A - B)$ es 1. Sin embargo, éste bit no indica nada desde el punto de vista del sistema binario sin signo, y por lo tanto debemos pensar en otra forma de determinar si el minuendo (A) es menor al sustraendo (B). Para indicar esto se dispone del bit de acarreo (en inglés *carry*) o del préstamo si se trata de una resta (en inglés *borrow*) que la ALU genera como parte del resultado de la operación.
Así, cuando la ALU ejecuta una instrucción aritmética, además de realizar la operación y generar un resultado, calcula un **conjunto de indicadores que caracterizan dicho resultado**. Estos indicadores se denominan **flags** (banderas) y son bits que se almacenan conjuntamente en un registro de la CPU.

En el caso de la arquitectura Q se cuenta con 4 flags, donde cada uno indica una situación distinta:

Flag Z (Zero)

Se activa (toma el valor 1) cuando el resultado de una operación resulta en la cadena completa de ceros. Es útil para la comparación por igual (A=B) mencionada en la primera situación.

Ejemplo en una resta:

$$\begin{array}{rr} & 111 \\ - & 111 \\ \hline & 000 \end{array} \quad \text{Z=1}$$

Ejemplo en una suma:

$$\begin{array}{rr} & 111 \\ + & 001 \\ \hline & 000 \end{array} \quad \text{Z=1}$$

Flag N (Negative)

Se aciva (toma el valor 1) cuando el bit más significativo del resultado es 1. Es útil para concluir si un valor es menor a otro en el contexto de CA2, como se describió en la segunda situación.

Ejemplo en una resta:

$$\begin{array}{rr} & 100 \\ - & 001 \\ \hline & 011 \end{array} \quad \text{N=0}$$

Ejemplo en una suma:

$$\begin{array}{rr} & 101 \\ + & 001 \\ \hline & 110 \end{array} \quad \text{N=1}$$

Flag C (Carry)

Se activa (toma el valor 1) cuando la resta o la suma tuvieron acarreo o borrow respectivamente. Es útil para concluir si un valor es menor a otro en el contexto de BSS, como se describió previamente.

Ejemplo en una resta:
$$\begin{array}{rr} & 100 \\ - & 001 \\ \hline & 011 \end{array} \quad \text{C=0}$$

Ejemplo en una suma:
$$\begin{array}{rr} & 110 \\ + & 011 \\ \hline & 001 \end{array} \quad \text{C=1}$$

Es importante ver que el Carry indica una situación de error en el contexto del sistema BSS (ver figura 9.2) pero no implica un error en el caso de un sistema CA2 (ver figura 9.3).
Nota: tener en cuenta que las cruces o tildes hacen referencia al error o no en el resultado, y no al cálculo del carry (C).

Ejemplo de BSS: en la primera suma, el carry no se activa `(C=0)` por lo que el resultado es correcto. Sin embargo, en los cálculos erróneos, en la suma, el carry sí se e activa `(C=1)`, debido a que el resultado se encuentra fuera de rango. Y para la resta, el resultado (-2) no es representable en el sistema BSS.

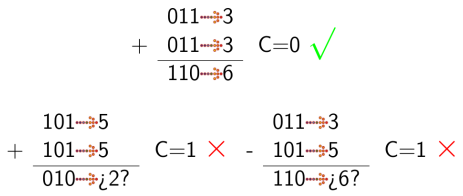


Figura 9.2: Significado del flag Carry en un contexto BSS

Ejemplo de CA2: se observan las 4 combinaciones, donde hay 2 casos que el carry se activa, y en uno el resultado es correcto pero en el otro es incorrecto. Asimismo ocurre cuando el carry no se activa, obteniendose un resultado correcto y en otro caso, uno erróneo.

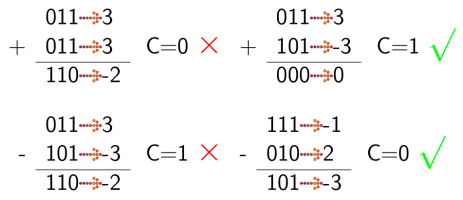


Figura 9.3: Significado del flag Carry en un contexto CA2

Flag V (Overflow)

Se activa (toma el valor 1) cuando el resultado no tiene el signo esperado, es decir en los siguientes casos:

(a) Cuando al sumar dos números positivos, se obtiene un resultado negativo

	positivo			010 ⇒ 2	
+	positivo	✗	+	010 ⇒ 2	, V=1
	negativo			100 ⇒ -4	

(b) Cuando al sumar dos valores negativos, se obtiene un resultado positivo

	negativo			100⇒-4	
+	negativo	✗	+	100⇒-4	, V=1
	positivo			000⇒0	

(c) Cuando al restar un valor positivo a uno negativo, se obtiene un resultado positivo

	negativo			110⇒-2	
-	positivo	✗	-	011⇒3	, V=1
	positivo			011⇒3	

(d) Cuando al restar un valor negativo a un valor positivo, se obtiene un resultado negativo

	positivo			001⇒1	
-	negativo	✗	-	100⇒-4	, V=1
	negativo			101⇒-3	

A partir de los **flags** descriptos, es posible evaluar las condiciones correspondiente a cada salto condicional, tal como se muestra en el cuadro 9.3.

Por último, es necesario aclarar que estos flags se calculan luego de ejecutarse la instrucción `CMP`, la cual, tal como ya se ha mencionado, se resuelve mediante una resta, con este único fin de calcular los flags, en lugar de obtener un resultado.

Volvamos al ejemplo de la rutina `asignarValor10` (que copiamos a continuación), suponiendo que `R0=0001`

```
1  asignarValor10: CMP R0, 0x0001
2                  JE asignarValor
3                  JMP fin
4    asignarValor: MOV R6, 0x000A
5                  fin: RET
```

Durante la ejecución de la instrucción `CMP`, la UC realiza la resta que resuelve la comparación, y luego calcula los 4 flags correspondientes en base al resultado obtenido. En este caso, la resta es:

	0000000000000001
-	0000000000000001
	0000000000000000

Y el cálculo de los flags resulta como: `Z=1`, `N=0`, `C=0`, `V=0`.

Op.	Descripción	Condición
JE	Igual / Cero	Z
JNE	No igual	$\overline{Z}$
JLE	Menor o igual	$Z + (N \oplus V)$
JG	Mayor	$\overline{Z + (N \oplus V)}$
JL	Menor	$N \oplus V)$
JGE	Mayor o igual	$\overline{(N \oplus V)}$
JLEU	Menor o igual sin signo	$C + Z$
JGU	Mayor sin signo	$\overline{(C + Z)}$
JCS	Carry / Menor sin signo	C
JNEG	Negativo	N
JVS	Overflow	V

Tabla 9.3: Detalle de la condición de cada salto condicional

Luego, se debe ejecutar el salto condicional `JE`, donde la UC evalúa, en base a los flags obtenidos, la expresión booleana correspondiente a su condición. En este caso como se trata de una comparación por igual, se evalúa el flag `Z` (ver tabla 9.3). Como dicho flag está activo `(Z=1)`, la expresión valúa **verdadero** y por ende, se desvía el flujo ("salta") a la dirección de memoria correspondiente según su desplazamiento. De esta manera se ejecuta la instrucción conjunta a la etiqueta `asignarValor`, la cual es `MOV R6, 0x000A`. Por último, la rutina finaliza con la ejecución de la instrucción `RET`.

Nota: tener en cuenta que en la explicación de la ejecución de CMP el foco está puesto en el bajo nivel (UC), mientras que con el resto de las instrucciones se puso énfasis en una explicación de alto nivel, sólo para completar, de manera resumida, la ejecución de la rutina.

9.3. Prueba de rutinas

Ya habiendo hablado de los conceptos de rutinas, subrutinas y reuso, se presenta un inconveniente. Y es que a veces pensamos rutinas que resuelven el problema para un conjunto reducido de datos, sin contemplar otros casos, los cuales pueden retornar un resultado erróneo.

Para corroborar que las rutinas cumplen con su propósito y retornan el resultado esperado, se necesita definir otras rutinas que prueben (testeen) el resutlado de aquellas que resuelven el problema original, brindando una herramienta de control de calidad de nuestras rutinas. A este tipo de rutinas se las denomina **rutinas de test**.

Es importante destacar que la rutina que resuelve el problema dado, se programa de manera general, mientras que la rutina de test se programa para un caso / situación específica; donde los valores o datos brindados serán definidos según el criterio de quien programe.

Con este objetivo, se utiliza una variable que funciona como indicador del resultado del test. Si luego de ejecutar la rutina de test la variable contiene una determinada cadena, por ejemplo `F000`, entonces el **test fue exitoso**, es decir, que la rutina funciona correctamente. En cambio, si la variable tiene la cadena, por ejemplo `FFFF`, entonces el resultado obtenido no es el que se esperaba, indicando que el **test falló**. Notar que las cadenas que describen error o éxito son elección arbitraria de la persona que escribe la rutina de test.

Por ejemplo, deseamos testear la rutina `promedio`. Para ello, vamos a definir una rutina de test que llamaremos `testPromedio`. En su código fuente, se debe invocar a la rutina a probar, que en este caso es `promedio`, realizando previamente el pasaje de parámetros correspondiente (lo mencinado en el campo **requiere** de la documentación de `promedio`). Luego, se deberá comparar el resultado obtenido con la cadena que representa el valor esperado (aquel mencionado en el campo **retorna** de la rutina `promedio`). Si los datos coinciden, se actualiza la variable del indicador del test, con la cadena `F000`, y en caso contrario con la cadena `FFFF`.

A continuación se describe la *estructura general de una rutina de test* :

```
Pasaje de parámetros ⇒   testRutina:   MOV var1, 0x...
Invocación ⇒                           CALL rutinaAValidar
Comparar con dato esperado ⇒           CMP R.Obtenido, 0x...
salto al caso de exito ⇒               JE funcionaBien
registrar fallo ⇒                      MOV R0, 0xFFFF
                                       JMP fin
registrar exito ⇒        funcionaBien: MOV R0, 0xF000
                                 fin:  RET
```

Aclaraciones: `var1` representa cualquier variable que la rutina a probar requiera, es decir, todas las celdas y registros del requiere.

A su vez, `F000` y `FFFF` dependen del valor elegido como indicador de un test exitoso o de falla, pudiendo ser distintos siempre y cuando se informe en la documentación qué valor (cadena) es el indicador de cada caso.

Como toda rutina, la misma deberá contar con su documentación, la cual deberá indicar los siguiente:

`testRutina`	
Requiere	—
Retorna	en R0 la cadena `F000` *(o el valor arbitrario elegido como éxito)* si la rutina `rutinaAValidar` retornó el valor que se esperaba, o `FFFF` *(o el valor arbitrario elegido como fallo)* en caso contrario
Modifica	**¡Completar según el test y la rutina a validar!**

Tener en cuenta que las rutinas de test no tienen parámetros, puesto que no requieren datos previos para poder ejecutarse.

9.3.1. Diseño de las pruebas

Suponer la documentación de la rutina `esMenorQueCeroSM` que se muestra a continuación.

esMenorQueCeroSM	
Requiere	en R5 un valor en SM(16)
Retorna	en R6 la cadena `0001` si es menor que cero, o `0000` si es mayor o igual que cero
Modifica	R4

A partir de la documentación de la rutina a testear -y sin conocer su código fuente- se puede definir el siguiente conjunto de rutinas de prueba, para analizar distintos casos.

Veamos la documentación del **Test 1** a modo de ejemplo, dejando de lado el resto de las documentaciones, las cuales son muy similares:

testIgualACero	
Requiere	—
Retorna	en R0 la cadena `0xF000` si el valor de la rutina `esMenorQueCeroSM` es 0, o la cadena `0xFFFF` en caso contrario.
Modifica	R4, R5, R6

Test1 Este caso de prueba (test) contempla cuando el dato es **igual** a cero. Por lo tanto, el parámetro pasado es la cadena `0x0000`.

```
1    testIgualACero: MOV R5, 0x0000
2                    CALL esMenorQueCeroSM
3                    CMP R6, 0x0000
4                    JE funcionabien
5                    MOV R0, 0xFFFF
6                    JMP fin
7    funcionabien: MOV R0, 0xF000
8        fin:    RET
```

Test2 Este caso de prueba (test) contempla cuando el dato nuevamente es **igual** a cero, dado que se trata del sistema SM que cuenta con doble representación del cero. Así, en este caso el parámetro pasado es la cadena `0x8000`.

```
1    testIgualACero2: MOV R5, 0x8000
2                     CALL esMenorQueCeroSM
3                     CMP R6, 0x0000
4                     JE funcionabien
5                     MOV R0, 0xFFFF
6                     JMP fin
7    funcionabien: MOV R0, 0xF000
8        fin:    RET
```

Test3 Este caso de prueba (test) contempla cuando el dato es **mayor** a cero. Por lo tanto, el parámetro pasado es la cadena `0x0008` (una a elección).

```
1    testMayorACero: MOV R5, 0x0008
2                    CALL esMenorQueCeroSM
```

```
                CMP R6, 0x0000
                JE funcionabien
                MOV R0, 0xFFFF
                JMP fin
funcionabien: MOV R0, 0xF000
    fin:    RET
```

Test4 Este caso de prueba (test) contempla cuando el dato es **menor** a cero. Por lo tanto, el parámetro pasado es la cadena `0x8008` (una a elección).

```
testMenorACero: MOV R5, 0x8008
                CALL esMenorQueCeroSM
                CMP R6, 0x0001
                JE funcionabien
                MOV R0, 0xFFFF
                JMP fin
funcionabien: MOV R0, 0xF000
    fin:    RET
```

9.3.2. Ejecución de las pruebas

En esta sección se ejecutan las rutinas de test que se definieron en la sección anterior para analizar los resultados y así poder elaborar conclusiones.

Considerando la documentación de la rutina `esMenorQueCeroSM` y su siguiente definición:

```
esMenorQueCeroSM:   MOV R4, R5
                    DIV R4, 0x8000
                    MOV R6, R4
                    RET
```

El resultado de la ejecución de las rutinas de test definidas es:

Test1 El resultado obtenido fue `R0=F000`

Test2 El resultado obtenido fue `R0=FFFF`

Test3 El resultado obtenido fue `R0=F000`

Test4 El resultado obtenido fue `R0=F000`

9.3.3. Conclusiones sobre la ejecución de las pruebas

Al momento de analizar los resultados del conjunto de pruebas es posible ver que en todas ellos se esperaba obtener en `R0` (que es el registro que se definió como indicador del resultado de la prueba) la cadena `0xF000` (la cadena elegida para representar que la prueba fue exitosa). Esto es lo que ocurrió en 3 de los 4 casos, indicando que dichas pruebas fueron exitosas.

Sin embargo, en el caso de la segunda prueba, se ve que se obtuvo un `FFFF`, indicando que ésta falló. ¿Cuál fue la causa de dicho resultado? Al analizar el caso, se ve que la cadena asignada a `R5` en el test, representa al 0 (o mas precisamente -0). A pesar de esto, la rutina `esMenorQueCeroSM` retorna un `0001`, indicando que el valor representado por `0x8000` es menor a cero, lo cual es falso.

¿Qué implicaciones tiene esta situación con respecto a la rutina `esMenorQueCeroSM`? Esto implica que se detectó un error en el funcionamiento de la misma, por lo que la rutina debe ser modificada. Sin embargo, la prueba definida **NO** debe modificarse, pues es útil para detectar un caso muy especial (llamado "borde") donde la rutina no funciona como se esperaba, es decir no cumple con su documentación.

Dicho de otro modo, la segunda prueba falla a causa de un error en la rutina que se estaba probando, por lo que sería incorrecto pensar que la prueba está mal planteada, sino todo lo contrario, pues fue precisamente gracias a la prueba que se encontró tal error.

Capítulo 10

Estructura de control: Repetición

En el capítulo 9.1 se vió que los saltos permiten desviar el flujo de ejecución de las instrucciones. Esta capacidad permite construir estructuras de programación para resolver problemas que comparten un mismo bloque de código.

Supongamos que no contamos con la operación `MUL` dentro de nuestro set de instrucciones, pero necesitamos resolver una multiplicación, por ejemplo `3x2`. Este problema se puede pensar como: `3*2 = 2+2+2`. De esta forma, podremos resolver la multiplicación como una sucesión de sumas.
En términos de algoritmia, este problema comparte el mismo bloque de código (la suma), el cual deberá ejecutarse 3 veces. A este tipo de estructuras se las conoce normalmente como **repeticiones**, también denomiadas iteraciones, ciclos o bucles (loop en inglés). Las repeticiones son un tipo de estructura de control, ya que se utilizan para "controlar" el flujo de la información y cómo procesarla.

De esta manera, mediante el uso de los saltos es posible desviar el flujo hacia la instrucción que deseamos repetir. Lo cual implica, en términos de arquitectura, actualizar el registro `PC` en dirección a una instrucción que ya se ha ejecutado, logrando así que un bloque de código se ejecute más de una vez. Para resolver este efecto de repetición se utilizará el **salto incondicional**, cuya instrucción es `JMP`. A su vez, es importante contar con una forma de terminar o "cortarçon dicha repetición, ya que de lo contrario, el programa entraría en un ciclo infinito, y cuya ejecución nunca finalizaría. En general, el corte de la repetición se resuelve con **saltos condicionales**, considerando alguna condición que permita validarse antes de continuar con la ejecución del bloque de código correspondiente.

Volvamos al ejemplo de la multiplicación, pero en esta ocación planteando el problema de manera generalizada. Así, la multiplicación entre 2 valores `A` y `B` queda expresada como: `A*B = B+B+...+B`, sumando B veces el valor de A (o viceversa). De esta manera, podremos definir una rutina que resuelva dicho problema general. Para ello, contamos con la siguiente documentación de la rutina que llamaremos `mulSinMul`

mulSinMul	
Requiere	en R7 un valor A y en R6 un valor B
Retorna	en R5 el resultado de realizar A*B
Modifica	R6

El algoritmo, expresado en pseudo-código, que permite cumplir con esta documentación es el siguiente:

```
1. R<-0  // variable R para el resultado, inicialmente en 0).
2. Validar si (B=0)
3. Si (B=0) es Verdadero, entonces Finalizar.
4. De ser Falso: R<-(R+A) //suma parcial
5. B<-(B-1) //decrementar B
6. Repetir, volviendo al paso 2.
```

Analicemos los pasos propuestos:

- **Paso 1:** a la variable `R`, utilizada para almacenar el resultado, se le asigna un valor inicial, que en este caso es el valor 0, a partir del cual se irán **acumulando los resultados parciales**. Así que, a las variables que se utilicen con este fin, se las denomina **acumulador**, y al hecho de asignarle un valor inicial, se lo conoce como **inicializar la variable**.
- **Paso 2:** la condición del paso 2, se utiliza para saber si es necesario continuar iterando o si se cumplió con la **cantidad de iteraciones correspondientes**, que en nuestro caso, es "agotar"las `B` veces. Para este tipo de decisiones se utilizan variables que denominamos **contadores**, las cuales nos permiten "contar"la cantidad de iteraciones. En este caso `B` cumple la función de contador.
- **Paso 3 y 4:** en estos pasos, se toman las acciones correspondientes a la evaluación de la condición por `Verdadero` y por `Falso`, donde para el primer caso, se "corta"la repetición, tras haber finalizado con la cantidad de veces necesarias y así haber obtenido el resultado final. En cambio, para el segundo caso (paso 4), se debe continuar con el cálculo parcial correspondiente, **acumulando en R** la suma en cuestión.
- **Paso 5:** para poder cortar con la repetición, es decir que en algún momento la condición de `B=0` sea verdadera, será necesario ir decrementando la variable `B` a medida que vamos resolviendo cada suma parcial.
- **Paso 6:** en este paso se realiza efectivamente la repetición que permite volver a ejecutar el bloque de código previamente mencionado, comenzando con la evaluación de la condición.

Una vez analizado el problema y habiendo definido una versión en pseudo-código que nos permita acercarnos a su desarrollo en el lenguaje Q, procedemos a definir la rutina correspondiente:

```
mulSinMul: MOV R5, 0x0000 //R5 es el acumulador (y donde retornar)
  repetir: CMP R6, 0x0000 //R6 contiene el valor B (es el contador)
           JE fin         //Si B=0 entonces fin
           ADD R5, R7     //R7 contiene el valor A (R<-R+A)
           SUB R6, 0x0001 //Se decrementa B
           JMP repetir    //Iterar, volver al paso 2
      fin: RET
```

Nota: las acotaciones indicadas al lado de cada instrucción con la sintaxis de doble barra ‘‘//" son comentarios internos del código que permite enriquecer la comunicación de la rutina para uso de quienes programan. Y por tanto no se ejecutarán como parte del código fuente.

10.1. Estructura general

En el ejemplo de repetición expuesto en la sección anterior, es posible identificar el esquema general que respetan las estructuras de control “repetición“ en las rutinas en lenguaje Q. Dicho esquema se conforma con las siguientes secciones:

- **Inicialización:** donde se inicializan las variables en las que se almacenan, y construyen los resultados, usualmente los contadores y/o acumuladores.
- **Condición de corte:** donde se evalúa el corte de la iteración. Se compone de una comparación y un salto condicional que permite salir / cortar con la repetición.
- **Cuerpo del ciclo:** donde se definen las instrucciones que deben repetirse para procesar y resolver el problema.
- **Retorno:** para implementar la iteración. Se realiza medainte un salto incondicional `JMP` hacia la condición de corte.
- **Fin:** finalmente, por fuera del ciclo, normalmente se ubica la etiqueta que identifica la primera instrucción fuera de la repetición.

Veamos esta estructura generalizada de la siguiente manera:

```
        INICIALIZACION  ==> Inicializar contadores y/o acumuladores
repetir:CMP X,Y        ==> X e Y son variables (registros o celdas)
        JE salir        ==> El salto condicional necesario
        CUERPO DEL CICLO ==> Deberian modificar a X o Y
        JMP repetir     ==> Retorno
  salir: ...            ==> Finaliza repeticion, continua la rutina
```

Entonces, al aplicar la estructura anterior a la rutina `mulSinMul` de la sección anterior, se puede identificar dicha **estructura**:

```
mulSinMul: MOV R5, 0x0000 //Inicializacion del acumulador
  repetir: CMP R6, 0x0000 //Condicion de corte
           JE fin
           ADD R5, R7     //Cuerpo del ciclo
           SUB R6, 0x0001
           JMP repetir    //Retorno
      fin: RET           //Fin
```

Capítulo 11

Q5: Máscaras

En muchas situaciones, dada una cadena de bits, es necesario trabajar solo con determinadas posiciones y que todas las restantes mantengan su valor original o bien tengan un valor que se desee.
Por ejemplo podría ser necesario lograr los siguientes efectos:

- Invertir ciertas posiciones dejando el resto intacto
- Conocer el valor de determinado bit de una cadena y que el resto de la cadena sea cero
- Separar un campo de la cadena (*slice*/rebanada).

Conceptualmente, se trata de mantener sólo aquellos bits que son de interés, y ocultar el resto detrás de valores determinados o controlados, como si se tratase de una máscara. A este concepto de **"enmascarar"** una cadena para obtener sólo una determinada porción de ella (subcadena) se la denomina **Máscara.**

Suponer el siguiente problema: se cuenta con una cadena de 8 bits que codifica dos datos independientes (dos campos), donde los 4 bits más significativos representan un desplazamiento en el eje X, y los 4 menos significativos un desplazamiento en el eje Y. Pero se desea obtener sólo aquellos bits que representen la coordenada sobre el eje Y, descartando el resto. Para resolver este problema, debemos **aplicar una máscara** que permita obtener sólo la porción de la cadena deseada.

En la figura 11.1 se ilustra la cadena de 8 bits, y la máscara de 4 bits que resulta en una especie de **ventana** sobre los bits menos significativos (correspondientes a la coordenada Y), y una especie de **bloqueo** en los bits más significativos correspondientes a la coordenada X). En la figura se muestra el resultado de "tapar" una parte de la cadena, para dejar visible sólo los bits que resultan de nuestro interés.

La tarea de aplicar una máscara es en realidad **una operación lógica entre dos cadenas**: la cadena con la **información a analizar**, y la cadena de la **máscara** que tiene el mismo tamaño y una estructura determinada para los bloqueos y las ventanas. Las operaciones lógicas entre cadenas, son operaciones que se llevan a cabo **bit a bit**.

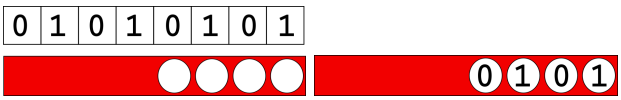


Figura 11.1: Ejemplo de aplicación de máscara (izq) y resultado (der)

	0101			0101
AND	0011	⇒	AND	0011
	????			0001
	0101			0101
XOR	0011	⇒	XOR	0011
	????			0110

Tabla 11.1: Ejemplos de operaciones lógicas bit a bit

De manera general, suponiendo que hay una operación lógica binaria $\triangle$, una cadena de 4 bits y una máscara **m** también de 4 bits, la operación se denota como sigue:

$$\begin{array}{rc} & b_3b_2b_1b_0 \\ \triangle & m_3m_2m_1m_0 \\ \hline & r_3r_2r_1r_0 \end{array}$$

Donde, cada bit r_i es el resultado de aplicar $b_i \triangle m_i$, por ejemplo $b_2 \triangle m_2 = r_2$. Para más ejemplos de operaciones lógicas ver los ejemplos en la tabla 11.

En los siguientes apartados se analizará cada operación lógica en relación a los bloqueos y ventanas que necesita.

11.1. Conjunción

De la tabla de verdad de la conjunción se deduce que:

- Al utilizar en la máscara, el bit 1 en determinadas posiciones, el resultado de la operación en esas mismas posiciones coincidirá con el bit de la cadena de la cual se desea extraer información. Es decir que, en la conjunción, un **bit en 1** en la máscara determina una **Ventana**. Quedando:

$$X \wedge 1 = X$$

- En cambio, donde la máscara tenga el bit 0, el bit de resultado será 0. Es decir que, en la conjunción, un **bit en 0** en la máscara determina un **Bloqueo**. Quedando:

$$X \wedge 0 = 0$$

De esta manera, siguiendo el ejemplo anterior de las cadenas con coordenadas X e Y, pero ampliando a 16 bits la cadena a resolver, se necesita como máscara la cadena: `0000000011111111`, ya que la aplicación bit a bit de la conjunción resulta en:

$$\begin{array}{rc} & 0101010101010101 \\ \wedge & 0000000011111111 \\ \hline & 0000000001010101 \end{array}$$

Notar que en color rojo se marcan los bloqueos de la máscara y de aquellos bits resultantes de aplicar dicha máscara a la cadena original.

Esta cadena de resultado representa el desplazamiento en el eje Y, que ahora puede utilizarse de manera independiente.

Para poder resolver esta situación en el lenguaje Q, se amplió el repertorio de instrucciones que obtenemos en Q5. Así, para la **conjunción** se utilizará la **instrucción** `AND`.

Veamos entonces como sería el uso de máscaras en una rutina, mediante la instrucción `AND`, que calcula la conjunción bit a bit. Para ello, se cuenta con la siguiente documentación:

`obtenerCoords`	
Requiere	en R3 una cadena de 16 bits, donde los 8 bits más significativos representan la coordenada X y los 8 bits menos significativos la coordenada Y
Retorna	en R3 la coordenadaY, y en R4 la coordenada de X
Modifica	—

```
1 obtenerCoords:  MOV R4, R3     // Copiar la cadena original
2                 AND R3, 0x00FF // mascara para Y
3                 AND R4, 0xFF00 // mascara para X
4                 RET
```

Para comprender cómo funciona la rutina, vamos a definir una rutina principal que contemple una cadena de ejemplo:

`main`	
Requiere	——
Retorna	en R3 la coordenadaY, y en R4 la coordenada de X de una cadena de 16 bits, donde los 8 bits más significativos representan la coordenada X y los 8 bits menos significativos la coordenada Y
Modifica	—-

```
1 main:  MOV R3, 0x55AA     // parametro
2        CALL obtenerCoords
3        RET
```

Analicemos el resultado de ejecutar la rutina `main`. Pero previamente vamos a detallar los **datos iniciales**, que es la información (los datos) a partir de los cuales vamos a procesar / resolver el problema. Y en el **resultado esperado**, vamos a incluir el/los resultado/s que esperamos obtener luego de ejecutar la rutina.

- **Datos iniciales:** el parámetro de la subrutina `obtenerCoords` que se corresponde con la cadena `0x55AA` que en binario es: `0101010110101010`.

- **Resultado esperado:** se espera que en `R3` quede almacenada la cadena con que representa la coordenada Y, la cual resulta en: `0000000010101010`. Mientras que en `R4`, la cadena que represneta la coordenada X, que resulta en: `0101010100000000`. Notar que en cada caso se **bloquearon** los bits que no se corresponden con lo buscado.

Ahora analicemos el código de la rutina `obtenerCoords`:

- Primero, se copia en `R4` la cadena original antes de operar sobre la coordenada Y, para luego poder calcular la coordenada X.
- A la cadena original que se encuentra en `R3`, se le aplica con el operador lógico `AND`, la máscara `0x00FF`, cuya cadena binaria es: `0000000011111111`. Generando así una ventana en los bits menos significativos que se corresponden con la coordenada Y.
- Asimismo, se aplica a `R4` (que tenía la cadena original), otro operador `AND` pero con la máscara `0xFF00`, cuya cadena binaria es `1111111100000000`, generandó así una ventana en los bits correspondientes a la coordenada X, bloqueando el resto.

11.2. Disyunción

De la tabla de verdad de la disyunción se deduce que:

- Al utilizar en la máscara el bit 1 en determinadas posiciones, el resultado de la operación en esas mismas posiciones será 1, independientemente del valor de la cadena de la cual se quiere extraer información. Es decir que, en la **disyunción**, un bit en 1 en la máscara determina un **bloqueo**.

$$X \vee 1 = 1$$

- En cambio, donde la máscara tenga el bit 0, el resultado coincidirá con el bit que tenga la cadena en dicha posición. Es decir que, en la **disyunción**, un bit en 0 de la máscara determina una **ventana**.

$$X \vee 0 = X$$

Con esta operación también es posible lograr la separación de los campos, como se necesitaba en el ejemplo de las coordenadas, pero utilizando una máscara diferente: `11110000`, acorde a su naturaleza:

$$\begin{array}{rl} & 0101010101010101 \\ \vee & 1111111100000000 \\ \hline & 1111111101010101 \end{array}$$

Esta cadena de resultado representa la coordenada de Y que ahora puede usarse de manera independiente.

De la misma manera que con la conjunción, Q5 incorpora una instrucción para operar con la **disyunción**, y se trata de la instrucción `OR`.

Veamos entonces como sería utilizar una máscara en una rutina, mediante la instrucción `OR`, que calcula la disynción bit a bit. Dado que el procedimiento es muy similar a la conjunción, en esta oportunidad se desea calcular solamente la coordenada X directamente desde la rutina principal.

```
1 main: MOV R3, 0x5555 // cadena a operar
2       OR R3, 0xFF00 // mascara para X
3       RET
```

En esta oportunidad la cadena original a operar es `0x5555`, que en binario resulta en: `0101010101010101`. Luego, se aplica el operador `OR` para la disyunción con la máscara `0xFF00`, cuya cadena binaria es: `1111111100000000`. Y donde el resultado final queda almacenado en el registro `R3`.

Negación

Al igual que el resto de los operadores, Q5 también incluyó la instrucción `NOT` que permite invertir los bits de una cadena, respetando así la lógica correspondiente del operador lógico de la negación. La implementación de este operador resulta de manera muy similar al resto de los operadores mencionados.

Disyunción exlusiva

De la tabla de verdad de la disyunción exclusiva (or exclusivo) se deduce que:

- Al utilizar en la máscara el bit 1 en determinadas posiciones, el bit del resultado de la operación en esas mismas posiciones será el opuesto al original.

$$X \oplus 1 = \overline{X}$$

- En cambio, donde la máscara tenga el bit 0, el resultado coincidirá con el bit que tenga la cadena en dicha posición, lo que denominamos una **ventana** para este operador.

$$X \oplus 0 = X$$

Ejemplo de aplicación:

$$\begin{array}{rr} & 0110 \\ \text{XOR} & 0101 \\ \hline & 0011 \end{array}$$

Una aclaración importante es que Q5, **no incluye una instrucción** para operar con una disyunción exclusiva. Por lo cual, de necesitar aplicar este operador, se deberá desarrollar una rutina que lo calculo utilizando los operadores correspondientes.

11.3. Rutinas con máscaras

Como se dijo previamente, una rutina aplica máscaras para destacar determinados bits dentro de una cadena y a partir de eso llevar a cabo ciertas tareas, que pueden estar condicionadas a los valores de dichos bits. Por lo que, a modo de resumen podemos decir que a Q_5 se incorporan las siguientes instrucciones:

Instrucción	Tipo	Efecto	Ejemplos
AND	2 operandos	Dest ← Dest ∧ Origen	`AND R1,0x00FF` / `AND R1, R5`
OR	2 operandos	Dest ← Dest ∨ Origen	`OR R4, 0xFFF7`
NOT	1 operando	Dest ← NOT Dest	`NOT [0xAABB]`

Suponer, por ejemplo, que se debe escribir una rutina `esPar` considerando la siguiente documentación:

EsPar	
Requiere	en R6 un valor en BSS(16)
Retorna	en R5 el código 0x000F si R6 es par, o el código 0x000A en caso contrario
Modifica	R4

Una posible solución para esta rutina es:

```
EsPar: MOV R4, R6
       AND R4, 0x0001 // Se obtiene el bit menos significativo
       CMP R4, 0x0001 // Validar si es par o no
       JNE siEsPar   //No termina en 1, es par
       MOV R5, 0x000A // Valor para indicar que es impar
       JMP fin
siEsPar: MOV R5, 0x000F // Valor para indicar que es par
 fin:  RET
```

Como se introdujo en la sección 9.3, para validar que las rutinas cumplen con el objetivo esperado, se deben definir otras rutinas, que se denominan *rutinas de test*. En cada *test* se deben proveer valores que cumplan lo requerido por la rutina (campo **Requiere**) y luego de invocarla se debe comparar el resultado con el valor esperado, poniendo en R0 la cadena `0xF000` en caso de éxito, o la cadena `0xFFFF` en caso de fallo.

Retomando el caso de la rutina `esPar`, el siguiente test prueba que, al usar como parámetro la cadena 0x0008 (valor 8, que es par) la rutina `esPar` retorna el código esperado (`000F`).

```
testEsPar: MOV R6, 0x0008
           CALL esPar
           CMP R5, 0x000F
           JE funcionaBien
           MOV R0, 0xFFFF
           JMP fin
funcionaBien: MOV R0, 0xF000
           fin: RET
```

Por otro lado, el siguiente test prueba el caso impar, pasando como parámetro a la rutina, la cadena `0xAAA1` (que es impar pues termina con 1), y esperando encontrar el código `0x000A`.

```
testEsImpar: MOV R6, 0xAAA1
             CALL esPar
             CMP R5, 0x000A
```

```
4               JE funcionaBien
5               MOV R0, 0xFFFF
6               JMP fin
7   funcionaBien: MOV R0, 0xF000
8               fin: RET
```

Capítulo 12

Q6: Arreglos

Mediante el lenguaje de Q podemos resolver muchos de los problemas que se abordan con los lenguajes de más alto nivel, pero hasta ahora carece de una estructura de datos que permita representar una colección de datos.

12.1. Estructura de datos

Desde el punto de vista de la información, un arreglo es una estructura de datos, es decir, una de las maneras de organizar la información. Puntualmente un arreglo es un conjunto finito homogéneo de datos. Esto quiere decir que todos los datos tienen la misma naturaleza, o dicho de otra manera, tienen el mismo tamaño y tipo. El tipo de los datos es el significado que se les da, es decir lo que representan dentro de un contexto y por lo tanto, en ciertos casos será necesario indicar el formato con el que debemos interpretarlos. A los datos del arreglo en ocasiones también se los denomina elementos.

Por ejemplo, el siguiente es un ejemplo de un arreglo de 4 elementos:

0	`0x0050`
1	`0x008A`
2	`0x00C5`
3	`0x000F`

Donde cada elemento/dato representa el monto total vendido, en el sistema $CA2(16)$, en una panadería para cada día de la semana, comenzando por el lunes. Así, el **tamaño** del dato es **16 bits** y el **tipo** es **complemento a 2**, pues así se los debe interpretar para conocer su significado. La lectura que se debe realizar sobre el arreglo es: en la posición 0, que representa el día lunes, se tiene la cadena `0x0050`, que representa el valor \$80, correspondiente a las ganancias del día lunes. En la posición 1 se tiene la cadena `0x008A` que representa lo ganado el día martes (\$138). Y así se sucesivamente, se pueden interpretar las ganancias del resto de la semana.

En términos de arquitectura, los arreglos se implementan a través de la **memoria principal**, donde se utiliza un bloque de celdas consecutivas y cada uno de sus elementos puede ocupar una o más celdas, dependiendo de su tamaaño. El **tamaño** del arreglo se determina por la cantidad de elementos que tiene, y

no por la cantidad de celdas que ocupa. Es decir que no siempre cada elemento ocupa una sola celda. Hay arreglos con elementos de 16 bits y por lo tanto cada elemento ocupa una sola celda, pero puede haber arreglos con elementos de 32 bits y por lo tanto cada elemento ocupa 2 celdas. Asimismo, si se cuenta con un arreglo cuyos elementos son de 8 bits, cada celda almacenará 2 elementos.

Veamos otro ejemplo. Se cuenta con un arreglo con 3 elementos, donde cada elemento ocupa 32 bits (2 celdas) y representa un valor en el formato IEEE 754 de simple precisión (para más información repasar la sección 15.2.4).

	...
2000	C26B
2001	8000
2002	0020
2003	B800
2004	C26B
2005	000F
	...

El primer elemento es la cadena `C26B8000`, almacenado en las celdas `2000` y `2001`. El hecho de que se organice así y no en el orden inverso (`8000C26B`), se relaciona con su formato, es decir con el modo de interpretar dichos datos, y el cual se puede consultar en la documentación correspondiente.

Como tercer ejemplo, considerar el siguiente arreglo con 3 elementos, donde cada elemento representa los permisos de un archivo en un sistema operativo Linux, sabiendo que cada archivo tiene un usuario y un grupo al que pertenece.

	...
2000	01C0
2001	01FF
2002	0124
	...

Los permisos se organizan de manera tal de poder indicar si el *usuario* tiene o no permisos para leer (Read), escribir (Write) o ejecutar (eXecute), si el *grupo* puede leer, escribir o ejecutar, y lo mismo para *otros* usuarios que no son del grupo. Para expresarlo en 16 bits se utiliza el siguiente formato:

`0000000(7b)`	$\mathrm{R}_u\mathrm{W}_u\mathrm{X}_u$(3b)	$\mathrm{R}_g\mathrm{W}_g\mathrm{X}_g$(3b)	$\mathrm{R}_o\mathrm{W}_o\mathrm{X}_o$(3b)

Donde los 7 bits más significativos son de relleno, cada $\mathtt{R}_i$ indica si el archivo tiene permiso de lectura para el modo i (el modo es usuario (u), grupo (g) u otros (o)). Respectivamente $\mathtt{W}_i$ indica el permiso de escritura y $\mathtt{X}_i$ el permiso de ejecución.

En el ejemplo, el primer elemento es la cadena `01C0`, cuyos 16 bits son `0000 0001 1100 0000`, y en base a su formato tenemos `0000000 111 000 000`, por lo que podemos concluir que la cadena representa los permisos `rwx --- ---`, es decir que el archivo tiene como permisos de usuario: lectura, escritura y ejecución, pero ni los miembros del grupo ni otras personas pueden realizar estas operaciones sobre el archivo. El segundo archivo (dato) es `01FF`, cuya cadena en 16 bits es `0000 0001 1111 1111`, en base a su formato `0000000 111 111 111`, es decir que el archivo tiene los permisos `rwx rwx rwx`, y entonces puede ser leido, escrito y

ejecutado por cualquier persona (usuario, grupo y otros). Finalmente el último archivo es `0124`, cuya cadena en 16 bits es `0000 0001 0010 0100`, con el formato tenemos que `0000000 100 100 100`, es decir que el archivo tiene los permisos `r-- r-- r--` y por lo tanto puede ser leído por cualquier persona, pero no puede ser escrito ni ejecutado por nadie.

Límites del arreglo

Hasta ahora hemos visto la estructura de un arreglo con diferentes ejemplos, pero para operar con sus datos será necesario conocer donde inicia el arreglo y dónde finaliza, es decir cuál es su **condición de fin** o también denominada **condición de corte**. Para el caso del inicio, basta con indicar la dirección de memoria donde se encuentra el primer elemento, pero para conocer cuál es su condició de fin es posible contar con varias opciones:

- Conocer su tamaño, es decir la cantidad de elementos que posee el arreglo. Así su fin está dado en término relativo (a su cantidad).
- Conocer la posición de su último elemento, es decir su dirección de memoria. Su fin está dado en término absoluto.
- Conocer específicamente su último elemento (dato válido). Su fin está dado en términos absolutos.
- Conocer un dato específico que es inválido, es decir que no es un elemento del arreglo, y por tanto, no hay que tenerlo en cuenta para su operación. Siendo el elemento anterior a éste, el último elemento del arreglo. Su fin está dado en término relativo (al elemento inválido).

Para comprender mejor, analizar la figura 12.1 con ejemplos de estas opciones.

Procesamiento de los datos: recorrido de arreglos

Ya conocemos la estructura de un arreglo y cómo identificar sus límites, así como el tipo de información que puede contener.

Pero ¿cómo utilizar esta estructura de datos dentro de nuestras rutinas?

La ventaja de contar con esta estructura de datos, es la posiblidad de **procesar** los datos que contiene, y así lograr una transformación sobre ellos, o simplemente conocer una parte de ellos. Por ejemplo, realizar cálculos o aplicar una función, como ser un descuento, invertir el signo de un valor, u ordenar sus datos. Otra opción sería la de acumular un resultado, como ser una sumatoria, una totalización, o un promedio. Y en caso de tratarse de un dato que contiene mucha información, por ejemplo se puede obtener un porción específica de ellos. Algunos ejemplos de procesamientos de arreglos pueden verse en la figura 12.2.

Para llevar a cabo esta tarea, será necesario **recorrer** el arreglo, de manera de "pasar"por cada elemento para poder procesarlo. Por lo tanto, resulta natural pensar en una repetición, siendo que se necesita iterar por cada elemento del arreglo para aplicar el mismo procesamiento.

Opciones	**A** **Tamaño**	**B** **Dir. de último elemento**	**C** **Último elemento**	**D** **Dato inválido**
Inicio **Fin**	Dirección 2000 4 elementos	Dirección A000 Dirección A003	Dirección 0090 Dato 00CA	Dirección 2C01 Dato FFFF

	...
2000	0002
2001	0008
2002	0001
2003	000F
	...

	...
A000	0050
A001	008A
A002	00C5
A003	000F
	...

	...
0090	0050
0091	008A
0092	00C5
0093	00CA
	...

	...
2C01	0050
2C02	008A
2C03	00C5
2C04	FFFF
	...

Figura 12.1: Ejemplos de arreglos con datos de 16 bits

Tomemos como ejemplo el arreglo A de la figura 12.1. Donde sus datos representan edades de personas, y se necesita **calcular la sumatoria** de sus edades.

Con este escenario, los datos iniciales se presentan en la figura mencionada. Y la sumatoria de todas las edades resulta en 26, cuya cadena es 0x001A.

Aprovechando la repetición, un posible algoritmo con las herramientas que tenemos hasta ahora en Q5, es:

sumarEdades	
Requiere	un arreglo con 4 edades, el cual inicia en la celda con dirección 2000
Retorna	en R1 la suma total de todas las edades del arreglo
Modifica	R0

```
1 sumarEdades: MOV R0, 0x0000 //se inicializa el acumulador
2              MOV R1, 0x0000  //se inicializa el contador
3       ciclo: CMP R1, 0x0004 // condicion de corte: 4 elem.
4              JE fin
5              ADD R0, [0x2000] //Se suma una edad, pero: PROBLEMA!
6              ADD R1, 0x0001  //se incrementa el contador
7              JMP ciclo
8         fin: RET
```

Problema: se puede observar que con la instrucción `ADD R0, [0x2000]` siempre se suma la misma edad, es decir, funciona bien para la primera edad, pero no así para el resto. Por más que iteremos vamos acumulando siempre la cadena 0002. De esta manera es notorio que necesitamos un mecanismo para cambiar dinámicamente la dirección del elemento (en este caso 2000). Para solucionar esta problemática, será necesario incorporar a la arquitectura Q, **nuevos modos de direccionamientos** que funcionen como una variable, y permitan acceder a cada elemento del arreglo para su procesamiento.

12.2. Modos de direccionamiento indirectos

La motivación para incorporar un nuevo modo de direccionamiento a la arquitectura Q, es la necesidad de un mecanismo de acceso que permita *indicar*

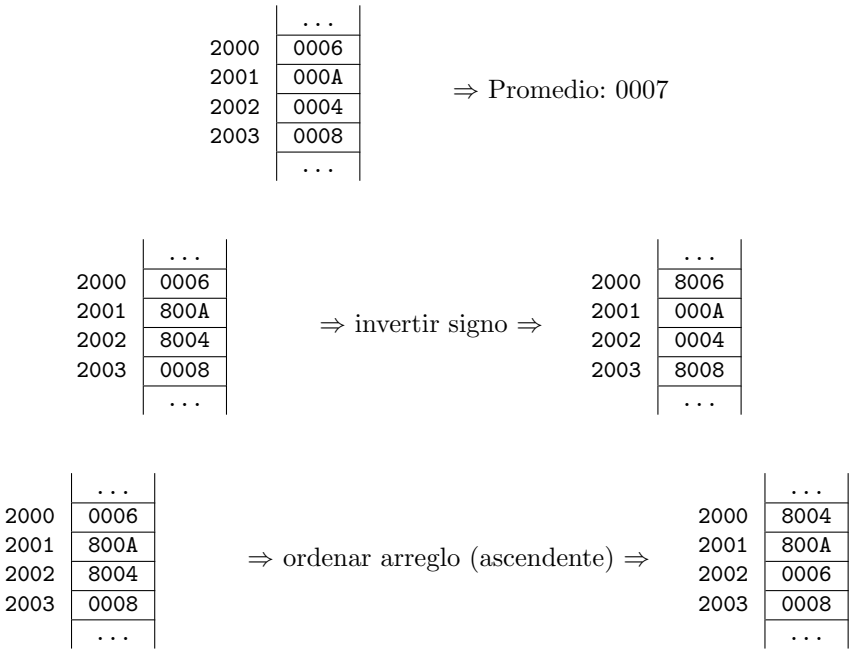


Figura 12.2: Ejemplos de procesamientos de un arreglo

el operando mediante una expresión y no de manera constante como se vió en el problema de la sección anterior. Una situación similar se vió antes en la gestión de la pila a través del registro SP o *Stack Pointer*.

A este nuevo modo de direccionamiento se lo denomina **Modos de direccionamiento indirecto**, y los mismos se encuentran en la versión actualizada de Q, que es `Q6`. En este tipo de modos de direccionamiento, el operando indicado en la instrucción es en realidad una dirección de un operando, y tiene dos posibles implementaciones: **Indirecto por Registro e Indirecto por Memoria**. De esta manera el operando, en lugar de representar un dato, direcciona a una celda donde sí se encuentra el dato. Veamos esta situación de indirección en un ejemplo de la la vida cotidiana. Supongamos que se necesita encontrar una oficina de la universidad y al preguntarle a una docente de la carrera, ésta nos **redirecciona** a la oficina de portería donde hay personas que tienen dicha información.

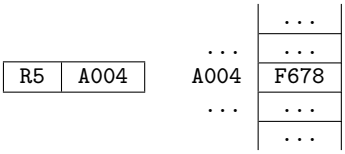
Indirecto por registro:

En este modo de direccionamiento se indica un registro que, en lugar de contener al operando, contiene la dirección de dicho operando. Esto es aplicable tanto, al operando origen, como al operando destino, y la sintaxis es `[Rx]`. Siendo `x` el número del registro.

Veamos el siguiente mapa de memoria para la instrucción: `MOV R0, [R5]`

Modo	Codificación
Inmediato	000000
Directo	001000
Indirecto por memoria	011000
Registro	100rrr
Indirecto por registro	110rrr

Tabla 12.1: Modos de direccionamiento en Q6



Donde el efecto de la ejecución de esta instrucción es: `R0 <- F678`

Analicemos la instrucción para comprender el funcionamiento de este nuevo modo de direccionamiento.
Se desea copiar a `R0`, **el contenido de la dirección que se encuentra en** `R5`. Es decir que los corchetes en el operando `[R5]` hacen referencia a una dirección de memoria. Por lo que el contenido del registro es en realidad una dirección (y no un dato), y por tanto, el dato se encuentra en la celda mencionada por éste. Así, la cadena `A004` almacenada en `R5`, es la dirección de memoria donde se encuentra finalmente el dato a copiar en `R0`, que en este caso es `F678`. De esta manera, se deduce que al buscar el operando, ocurre un acceso a memoria.

Pero para analizar mejor los accesos a memoria que provoca la ejecución de esta instrucción, será necesario primero ensamblarla. Para lo cual será necesario agregar un nuevo código que identifique este modo de direccionamiento (ver cuadro 12.1)

Ensamblado:
Así, el código máquina de la instrucción es `0001 100000 110101` y comprimido en hexadecimal es `0x1835`. Como podemos concluir, esta instrucción ocupa sólo una celda de memoria.

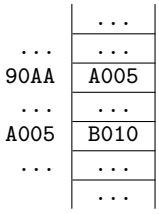
Volviendo al análisis de los accesos a memoria en las distintas etapas del ciclo de ejecución, se debe definir donde está ensamblada la instrucción. Asumamos que se encuentra en la celda con dirección `1E00`, y por lo tanto, los accesos a memoria por cada etapa son:

Instrucción	BI	BO	AR
`MOV R0, [R5]`	1E00	A004	–

Es notorio como el modo indirecto por registro del operando origen, provoca un acceso a memoria en la dirección `A004` durante la búsqueda de operandos. Notar, por otra parte, que en caso de que el operando destino fuese el que tenga el modo indirecto, provocaría un acceso en la etapa de AR, para su escritura, en lugar de la BO.

Indirecto por memoria:

De manera similar al caso de la Indirección por Registro, la Indirección por Memoria agrega un paso más para alcanzar el operando, utilizando como sintaxis el doble corchete, quedando un ejemplo: `[[0x90AA]]`. Nuevamente, este modo es aplicable tanto al operando origen como al destino.
Veamos la siguiente instrucción de ejemplo: `MOV R0, [[0x90AA]]`, y el siguiente mapa de memoria:



Donde el efecto de la ejecución de esta instrucción es: `R0 <- B010`

Analicemos el efecto de esta instrucción. Nuevamente el operando tiene un modo de direccionamiento indirecto (tiene corchetes), y por ende, su contenido hace referencia a una dirección de memoria. Pero éste, también referencia a una dirección de memoria, pues encontramos otros corchetes. De esta manera, lo que se busca es copiar a `R0`, **el dato que se encuentra en la celda cuya dirección es el contenido de otra celda**.
Volviendo al operando origen de la instrucción `[[0x90AA]]`, si analizamos el contenido del operando (corchete de más afuera), obtenemos esta expresión: `[0x90AA]`, y como nuevamente hay un direccionamiento (otros corchetes), se debe acceder a la dirección allí contenida, `0x90AA`, la cual almacena la dirección donde efectivamente está el operando.
Como conclusión, para buscar el operando, se accede a memoria tantas veces como direccionamientos tenga el operando (corchetes). En este caso, hay 2 corchetes, se accede 2 veces para llegar al operando en cuestión.

Ensamblado:
Para llevar adelante el análisis del ciclo de ejecución es necesario analizar cómo es el código máquina de una instrucción que utilice el modo de direccionamiento indirecto por memoria. Teniendo en cuenta nuevamente el cuadro 12.1, vemos que la instrucción `MOV R0, [[0x90AA]]` se ensambla: `0001 100000 011000 1001 0000 1010 1010` y compactado en hexadecimal esto es `1818 90AA`, es decir que el código máquina de la instrucción ocupa 2 celdas. Asumiendo que se la ubica a partir de la celda con dirección `1955`, los accesos a memoria principal para cada etapa son:

Instrucción	BI	BO	AR
`MOV R0, [[0x90AA]`	`1955, 1956`	`90AA,A005`	–

Nuevamente no hay almacenamiento de resultados (AR), dado que el operando destino es un registro, pero de invertirse los operandos, el acceso a las celdas `90AA` y `A005` se realizarían en la etapa de AR en lugar de la BO. Y por último, de tratarse de una operación aritmética o lógica, se accederá a memoria en ambas etapas, en BO para leer y en AR para escribir.

12.3. Recorrido de arreglos

Volviendo al problema de la sección inicial 12.1, y contando con los nuevos modos de direccionamientos, podremos reescribir el algoritmo para resolver el problema generado en la primera versión de la rutina.

Recordemos su documentación:

sumarEdades	
Requiere	un arreglo con 4 edades, el cual inicia en la celda con dirección `2000`
Retorna	en `R0` la suma total de todas las edades del arreglo
Modifica	`R1` y `R2`

Donde el resultado esperado es la cadena `0x001A`, que al interpretarla se obtiene el valor `26`.

	...
2000	0002
2001	0008
2002	0001
2003	000F
	...

En el primer abordaje notamos que los modos de direcciónamientos conocidos hasta entonces (registro, inmediato y directo) no eran suficientes para realizar el recorrido. A partir de contar con los modos indirectos, utilizaremos una variable (registro o celda de memoria) para llevar cuenta de la posición del elemento a procesar en cada iteración del recorrido. A esta variable se la denomina **índice del Arreglo**. El índice será necesario en cualquier recorrido que se realice sobre un arreglo para procesar sus elementos.

Índice del arreglo

Con este objetivo, vamos a utilizar, por ejemplo, el registro `R2` como índice del arreglo. Por lo cual, será necesario almacenar en el registro la dirección `2000`, ya que es el inicio del arreglo. Y así poder comenzar a recorrerlo. De esta manera la rutina se reescribe como:

```
1  sumarEdades: MOV R0, 0x0000 //se inicializa el acumulador
2               MOV R1, 0x0000  //se inicializa el contador
3               MOV R2, 0x2000 // se inicializa el indice
4         ciclo: CMP R1, 0x0004 // condicion de corte: 4 elem.
5               JE fin
6               ADD R0, [??]    //PROBLEMA
7               ADD R1, 0x0001  //se incrementa el contador
8               ADD R2, 0x0001 //se incrementa el indice (sig. dato)
9               JMP ciclo
10          fin: RET
```

Como último desafío debemos resolver el acceso a cada elemento del arreglo dentro del ciclo (`problema`), y para esto utilizaremos un **modo de direccionamiento indirecto por registro**, cuya instrucción resulta en `ADD R0, [R2]`.

Así, el código final de la rutina queda:

```
1  sumarEdades: MOV R0, 0x0000 //se inicializa el acumulador
2               MOV R1, 0x0000  //se inicializa el contador
3               MOV R2, 0x2000 // se inicializa el indice
4        ciclo: CMP R1, 0x0004 // condicion de corte: 4 elem.
5               JE fin
6               ADD R0, [R2]    //se acumula edad del elem. actual
7               ADD R1, 0x0001  //se incrementa el contador
8               ADD R2, 0x0001 //se incrementa el indice (sig. dato)
9               JMP ciclo
10         fin: RET
```

Procesar un arreglo de tamaño relativo

Pensemo el mismo problema que en el ejemplo anterior, pero considerando que el tamaño del arreglo no es absoluto (fijo), donde su condición de corte (fin) está dada a partir del dato inválido `0xFFFF`, cuyo valor 65.535 no tiene sentido dentro del contexto de las edades que contiene el arreglo.
Con esta nueva condición de corte, se debe recorrer el arreglo para calcular la sumatoria entre los elementos almacenados a partir de la celda con dirección `2000` pero hasta el primer dato `FFFF`, el cual no debe sumarse.

La documentación para esta versión de rutina es:

`sumarEdadRel`	
Requiere	un arreglo de edades que inicia en la celda con dirección `2000`, y finaliza con el dato inválido `0xFFFF`
Retorna	en `R0` la suma total de todas las edades del arreglo
Modifica	`R1`, `R2`

```
1  sumarEdadRel: MOV R0, 0x0000 //se inicializa el acumulador
2                MOV R2, 0x2000 // se inicializa el indice
3                MOV R1, [R2]      //se guarda el primer elem. (edad)
4         ciclo: CMP R1, 0xFFFF  //condicion de corte: dato invalido
5                JE fin
6                ADD R0, R1    //se acumula edad del elem. actual
7                ADD R2, 0x0001 //se inc. el indice para sig. elem
8                ADD R1, [R2]    //se guarda nuevo elem. para iterar
9                JMP ciclo
10          fin: RET
```

Capítulo 13

Subsistema de memoria

El sistema de cómputos necesita tener varios tipos de memorias para ser utilizadas en diferentes situaciones cumpliendo distintos objetivos. Antes de avanzar con este punto, veamos cómo clasificar las memorias según diferentes aspectos.

Los tipos de memorias pueden ser:

Volatilidad

1. **Interna o externa al sistema** según si es fija o desmontable (portable a otro sistema)
2. **Volátil o Persistente**. Se dice que es volátil si su contenido se pierde al cortarse la alimentación eléctrica (RAM). Y persistente, si no necesita electricidad para mantener la información.
3. **De lectura y escritura o sólo lectura**. De lectura y escritura, como se requiere en una memoria principal, o bien de sólo lectura para almacenar información estática que no necesita modificarse.

Métodos de Acceso

4. **Métodos de acceso: secuencial, directo, aleatorio o asociativo**. El **método de acceso secuencial** requiere que la dirección (o identificación) de cada dato se encuentre almacenada junto con él, y por lo tanto el dispositivo debe recorrerse secuencialmente hasta encontrar la identificación buscada. El **método de acceso directo** es el utilizado por los discos magnéticos, donde la dirección del dato se basa en su ubicación física, en particular la búsqueda se da en dos etapas: se accede primero a la zona que incluye al dato y luego se busca secuencialmente dentro de dicha zona. En el **método aleatorio** cada dato tiene un mecanismo de acceso único y cableado físicamente (cada acceso es independiente de los accesos anteriores). Finalmente, el **método asociativo** organiza cada unidad de información con una etiqueta que la describe en función de su contenido. Entonces, para recuperar un dato se debe analizar un determinado conjunto de bits dentro de la celda, buscando la coincidencia con la clave o patrón buscado. Esta comprobación del contenido de las celdas se lleva a cabo de manera simultánea en todas las celdas.

13.1. Memorias de sólo lectura: ROM

Para algunas aplicaciones, el programa no necesita ser modificado y entonces puede ser almacenado de manera permanente en una memoria de sólo lectura ó ROM (*Read Only Memory*). Ejemplos de memorias ROM pueden encontrarse en videojuegos, calculadoras, hornos de microondas, computadoras en automóviles, etc. Las computadoras en general necesitan , por ejemplo una memoria ROM donde almacenar a un disco rígido primario el primer programa que da arranque al sistema operativo.

13.2. Jerarquía de memorias

En la arquitectura de Von Neumann 6.1 la **memoria principal** es volátil y de acceso aleatorio. Por lo tanto al ser volátil se necesita otra posibilidad de almacenamiento donde los programas puedan almacenarse de manera persistente y desde donde se recuperen bajo demanda del usuario (cuando dispara la ejecución de un programa). A esta memoria la llamaremos Memoria Secundaria.

La **memoria secundaria** puede ser resuelta con un disco rígido o un disco de estado sólido (SSD) donde se mantengan persistentes (instalados) los programas. Cuando se necesita de la ejecución de un programa, ya sea a partir del requerimiento de un usuario o por invocación de otro programa, debe ser cargado en memoria y permanecer allí hasta que la memoria se sobreescriba o se apague el sistema.

La pregunta que puede surgir es ¿por qué no montar la memoria principal en una tecnología persistente, como ser un disco de estado sólido? Para responderla es importante destacar que existe una relación de compromiso entre el tiempo de acceso, el costo por bit y la capacidad de almacenamiento (ver figura 13.1). Un disco tiene mayor capacidad (y por lo tanto menor costo por bit) pero un tiempo de acceso mucho mayor. Esto impacta directamente en el desempeño de la CPU, pues el tiempo de ejecución de una instrucción está condicionado por el tiempo de acceso a memoria principal. Además, en este punto es importante notar que algunos programas pueden no ser ejecutados nunca y algunos datos pueden no ser accedidos, aunque se los tenga disponibles en el sistema.

Figura 13.1: Relación de compromiso entre capacidad, velocidad y costo

Por este motivo es que se incorpora una memoria más pequeña y rápida (de acceso aleatorio) donde se cargan los programas al momento de ser ejecutados, y

los datos a medida que se van necesitando. Si lo que se necesita es asegurar una velocidad aceptable y no se necesita gran capacidad, ¿Por qué no implementar la memoria principal con registros de la CPU? Para responder esto se debe tener en mente que el costo sea razonable. En general las arquitecturas cuentan con pocas decenas de registros debido al costo que eso implica. Por otro lado, si se implementa la memoria principal con una RAM, se otorga flexibilidad para extender su tamaño pues se trata de un componente externo a la CPU y en cambio los registros suelen estar definidos en el diseño electrónico de la misma. Este esquema se describe en la figura 13.2.

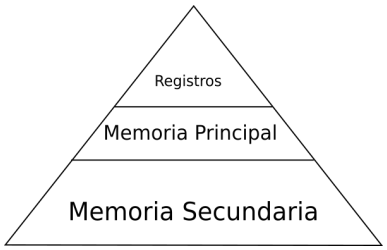


Figura 13.2: Jerarquía de memoria

13.3. Memoria Secundaria

13.3.1. Discos Magnéticos

Para la lectura o la escritura del disco, el cabezal debe estar posicionado al comienzo del sector deseado. En el caso de los dispositivos de cabezal móvil, el posicionamiento implica mover el cabezal hasta la pista deseada (*seek*) y luego esperar a que el sector deseado pase por debajo del cabezal (*latency*). De esta manera, el **tiempo de acceso al disco** es la suma del seek time, el latency time y el tiempo de transmisión.

Tiempo de Búsqueda (seek time): Es el tiempo que le toma a las cabezas de Lectura/Escritura moverse desde su posición actual hasta la pista donde esta localizada la información deseada. Como la pista deseada puede estar localizada en el otro lado del disco o en una pista adyacente, el tiempo de búsqueda variara en cada búsqueda. En la actualidad, el tiempo promedio de búsqueda para cualquier búsqueda arbitraria es igual al tiempo requerido para mirar a través de la tercera parte de las pistas. Los HD de la actualidad tienen tiempos de búsqueda pista a pista tan cortos como 2 milisegundos y tiempos promedios de búsqueda menores a 10 milisegundos y tiempo máximo de búsqueda (viaje completo entre la pista más interna y la más externa) cercano a 15 milisegundos .

Latencia (latency): Cada pista en un HD contiene múltiples sectores una vez que la cabeza de Lectura/Escritura encuentra la pista correcta, las cabezas permanecen en el lugar e inactivas hasta que el sector pasa por debajo de ellas. Este tiempo de espera se llama latencia. La latencia promedio es igual al tiempo que le toma al disco hacer media revolución y es igual en

aquellos drivers que giran a la misma velocidad. Algunos de los modelos más rápidos de la actualidad tienen discos que giran a 10000 RPM o más reduciendo la latencia.

Command Overhead: Tiempo que le toma a la controladora procesar un requerimiento de datos. Este incluye determinar la localización física del dato en el disco correcto, direccionar al .ªctuador"para mover el rotor a la pista correcta, leer el dato, redireccionarlo al computador.

Transferencia: Los HD también son evaluados por su transferencia, la cual generalmente se refiere al tiempo en la cual los datos pueden ser leídos o escritos en el drive, el cual es afectado por la velocidad de los discos, la densidad de los bits de datos y el tiempo de acceso. La mayoría de los HD actuales incluyen una cantidad pequeña de RAM que es usada como cache o almacenamiento temporal. Dado que los computadores y los HD se comunican por un bus de Entrada/Salida, el tiempo de transferencia actual entre ellos esta limitado por el máximo tiempo de transferencia del bus, el cual en la mayoría de los casos es mucho más lento que el tiempo de transferencia del drive.

13.3.2. Discos de estado sólido

Una unidad de estado sólido o SSD (acrónimo en inglés de *solid-state drive*) es un dispositivo de almacenamiento de datos que usa una memoria no volátil, como la memoria flash, para almacenar datos, en lugar de los platos giratorios magnéticos encontrados en los discos mencionados antes.

En comparación con los anteriores, las unidades de estado sólido son menos sensibles a los golpes, son prácticamente inaudibles y tienen un menor tiempo de acceso y de latencia.

La mayoría de los SSD utilizan memoria flash basada en compuertas NAND, que retiene los datos sin alimentación. Para aplicaciones que requieren acceso rápido, pero no necesariamente la persistencia de datos después de la pérdida de potencia, los SSD pueden ser construidos a partir de memoria de acceso aleatorio (RAM). Estos dispositivos pueden emplear fuentes de alimentación independientes, tales como baterías, para mantener los datos después de la desconexión de la corriente eléctrica.

13.3.3. Redundant Array of Inexpensive Drives

RAID es una tecnología que emplea el uso simultáneo de dos o mas discos duros para alcanzar mejor performance, mayor fiabilidad y mayor capacidad de almacenamiento. Los diseños de RAID involucran dos objetivos de diseño: mejor fiabilidad y mejor performance de lectura y escritura. Cuando un conjunto de discos físicos se utilizan en un RAID, se los denomina conjuntamente **vector RAID**. Este vector distribuye la información a través de varios discos, pero esto es transparente para el sistema operativo, pues lo maneja como si se tratara de un solo disco.

Algunos vectores RAID son redundantes en el sentido que se almacena información extra, derivada de la información original, de manera que el fallo de un disco en el vector no provocará pérdida de información. Una vez reemplazado el disco, su información se reconstruye a partir de los otros discos y la

información extra. Claramente, un vector redundante tiene menos capacidad de almacenamiento.

Existen varias combinaciones de estos enfoques dando diferentes relaciones de compromiso entre protección contra la perdida de datos, capacidad y velocidad. Estas combinaciones se denominan niveles, y los niveles 0,1 y 5 son los mas comúnmente encontrados pues cubren la mayoría de los requerimientos.

RAID 0 (*striped disks*): Distribuye la información a través de distintos discos de manera que se mejore la velocidad y la capacidad total, pero toda la información se pierde si alguno de los discos falla.

RAID 1 (*mirrored disks*): Utiliza dos (en algunos casos mas) discos que almacenan exactamente la misma información. Esto permite que la información no se pierda mientras al menos un disco funcione. La capacidad total de este esquema es la misma que la de un disco, pero el fallo de un dispositivo no compromete el funcionamiento del arreglo de discos.

RAID 5 (*striped disks with parity*): Este esquema combina tres o mas discos de una manera que se protege la información contra la perdida de un disco y la capacidad de almacenamiento del arreglo se reduce en un disco.

La tecnología RAID involucra importantes niveles de cálculo durante lecturas y escrituras. Si el RAID está implementado por hardware, esta tarea es llevada a cabo por el controlador. En otros casos, el sistema operativo requiere que el procesador lleve a cabo el mantenimiento del RAID, lo que impacta en la performance del sistema.

Los RAID redundantes pueden continuar trabajando sin interrupción cuando ocurre una falla, pero son vulnerables a fallas futuras. Algunos sistemas de alta disponibilidad permiten que el disco con fallas sea reemplazado sin necesidad de reiniciar el sistema.

Nivel de redundancia

$$red = capTotal/capU$$

Capítulo 14

Memoria Caché

14.1. Motivación

Desde hace un par de décadas el progreso tecnológico de las computadoras está marcando la tendencia de duplicar la velocidad de los procesadores y el tamaño de la memoria principal, pero no así la velocidad de las memorias, las cuales crecen apenas un 10 %. Esto ocasiona un crecimiento en la brecha entre la velocidad del procesador y la velocidad de la memoria, causando así un mayor impacto en el tiempo de ejecución de los programas, pues la velocidad con la que la CPU puede ejecutar instrucciones está limitada por el tiempo de acceso a memoria. Recordar que se accede al menos una vez dentro del ciclo de ejecución de instrucción (ver figura 7.6).

Una forma de abordar la solución a esta brecha, es incorporar una memoria más pequeña que la memoria principal pero mas rápida, teniendo en mente que no es viable construir la memoria principal a partir de registros internos de la CPU.

Esta idea se basa en que los accesos que se solicitan a la memoria principal no son azarosos, pero tampoco absolutamente predecibles, sino que respetan cierta lógica que tiene relación con la naturaleza de la ejecución de los programas. Durante la ejecución de un programa, un alto porcentaje de las instrucciones ejecutadas son consecutivas (es decir que no están desordenadas o distribuidas al azar en memoria), pero además de ser accedidas para recuperar instrucciones, las celdas pueden contener datos que muchas veces forman parte de una estructura de datos, como puede ser un arreglo (otro ejemplo de celdas de memoria consecutivas relacionadas semánticamente). Por otro lado hay situaciones en las que una celda de memoria se accede más de una vez, por ejemplo las instrucciones dentro de un ciclo (repetición) o una rutina que se reusa.

En consecuencia, el acceso a las instrucciones como a los datos, puede caracterizarse por dos principios, el **principio de localidad espacial** (ejemplo del arreglo) y el **principio de localidad temporal** (ejemplo de un ciclo o de reuso)

Formalmente, estos patrones de acceso se definen como:

1. El principio de **localidad espacial** enuncia que las direcciones de memoria cercanas a alguna celda accedida recientemente, son más probables de ser requeridas que aquellas más distantes.

2. El principio de **localidad temporal** menciona que cuando un programa hace referencia a una dirección de memoria, se espera que vuelva a hacerlo a la brevedad.

Como se mencionó anteriormente, el objetivo es subsanar la brecha entre los tiempos de respuesta de la memoria principal y de la CPU. Así es que a partir de estos principios se busca tener sólo las celdas más utilizadas (y no todas) en una memoria más inmediata e intermedia entre la CPU y la memoria principal (ver figura 14.1). Dicha memoria se denomina **Memoria Caché** (del francés: 'escondida'). Así es que en lugar de contar con una sola memoria, pasamos a contar con un **subsistema de memoria**, integrado por ambas memorias. Este subsistema será parte de la arquitectura de Von Neumann (sistema general).

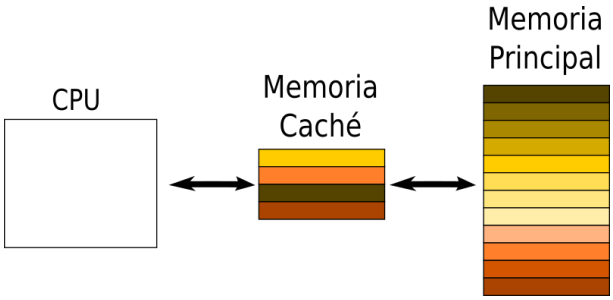


Figura 14.1: Memoria Caché como intermediaria

De esta manera podemos reformular la pirámide de la figura 13.2 incorporando una memoria menos costosa que los registros, pero con menos capacidad que la memoria principal. Ver figura 14.2.

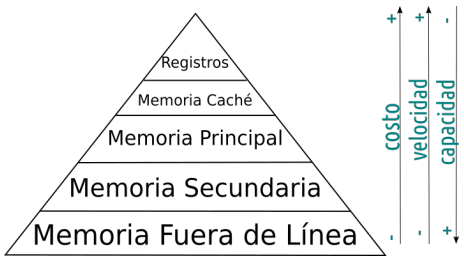


Figura 14.2: Jerarquía con Memoria Caché

14.2. Estructura de la memoria Caché

La memoria caché tiene como objetivo proveer una velocidad de acceso cercana a la de los registros pero con una mayor capacidad. Es por este motivo que la caché contiene copias de porciones de la memoria principal denominadas **bloques**. De este modo, la memoria principal se segmentará ("divide") en bloques, conceptualmente hablando, los cuales se copiarán a la memoria caché cada vez que se requiera. Un bloque básicamente consta de un conjunto de celdas. Por lo que la memoria caché almacena los datos contenidos en dichas celdas pero asociados, de alguna manera, a sus direcciones. Recordemos que una celda tiene asociada una dirección a través de la cual se accede a su contenido, es decir a su dato (instrucción, operando, etc).

Así como la memoria principal tiene celdas con direcciones, la memoria caché dispone de **líneas** [1], identificadas por **número de línea**, para acceder a su contenido; tanto a los datos como a la información asociada a ellos. Esta información dependerá de cómo se vinculan ambas memorias. Existen 2 maneras que se llevan a cabo mediante las **funciones de correspondencia**, que veremos en la próxima sección.

Siguiendo con la estructura mencionada, cabe destacar que cada bloque se identifica mediante un número de bloque. Análogamente, las líneas se identifican mediante un número de línea; siempre hablando en término de cadenas binarias.

Otra característica que comparten ambas memorias son las operaciones. La memoria caché, al igual que la memoria principal, también admite las **operaciones de lectura y escritura**. Pero siendo que en proporción son más numerosas las primeras que las segundas, arrancaremos con el **algoritmo de lectura** en caché, para finalmente concluir con el **algoritmo de escritura**, porque además ambos comparten algunos pasos (acceso a la caché). A su vez, el algoritmo de escritura dispone de 2 estrategias diferentes, denominadas **políticas de escrituras**, las cuales se clasifican en: **write-through** y **write-back**

14.2.1. Algoritmo de lectura en caché

Recordemos que la motivación de contar con una memoria más rápida radica en la ejecución de los programas. Supongamos entonces que se está implementando la etapa de búsqueda de instrucción del ciclo de ejecución, por lo que la unidad de control solicita la lectura de una celda. Así, la señal enviada por el bus de control es R/$\overline{W}$=1 y la dirección de la instrucción viaja por el bus de direcciones hasta el **subsistema de memoria** que ahora se compone de la memoria principal y la memoria caché.
Entonces, *¿cómo resuelve este subsistema de memoria la solicitud de la UC?*. Para responder a esta pregunta, veremos a continuación el algoritmo de lectura en caché.

El algoritmo se muestra gráficamente en la figura 14.3. Analicemos dicho algoritmo teniendo en cuenta la estructura mencionada en la sección anterior.

[1] También denominada como ranura o slot en inglés

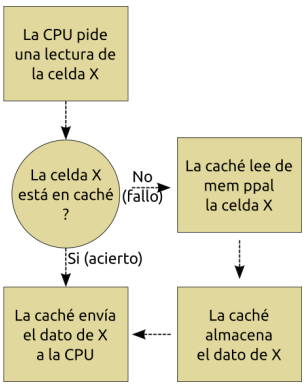


Figura 14.3: Algoritmo de lectura de caché

1. La CPU, puntualmente la UC, solicita la lectura de una celda, enviando su dirección hacia el subsistema de memoria.
2. Primero se debe resolver si la celda en cuestión (el dato) se encuentra almacenada en la memoria Caché, es decir si *la celda está cacheada*.Y siendo que se trata de una condición, vamos a contar con 2 alternativas.
3. En caso afirmativo, es decir la celda está cacheada, se dice que ocurrió un **acierto**, para lo cual la caché envía el dato de la celda a la UC a través del bus de datos.
4. En caso contrario, la celda no está cacheada, se dice que ocurrió un **fallo**, para lo cual la caché deberá:
   a) **Acceder a la memoria principal**: solicita, mediante su dirección, la celda a la memoria principal. Por lo que realiza una lectura sobre ésta.
   b) **Almacenar el bloque de la memoria principal**: almacena en una línea, la copia de la celda provista por la memoria principal junto con el resto de las celdas que conforman el bloque. Pues recordemos que la caché almacena bloques de celdas, no una única celda. También es importante notar que se trata de una copia, por lo que la información persiste en ambas memorias.
   c) **Retornar el dato:** finalmente la caché está en condiciones de enviar a la UC el dato de la celda para la dirección solicitada.

Si observamos, la tarea final de responder a la UC con el dato de la dirección solicitada es común en ambas alternativas, tanto ante un acierto como ante un fallo. Para este último caso, la tarea es más sencilla, puesto que se accedió al dato durante la lectura en la memoria principal (paso 4.a) Pero ante un acierto, esta tarea requiere de un paso más, puesto que dicho dato es parte de un bloque, por lo que será necesario identificarlo dentro del mismo. Así es que introducimos el concepto de **índice** o mejor dicho **bits de índice**. Como su nombre lo indica, estos bits están destinados a proyectar una porción del bloque [2]. Es decir que

[2]Esto puede resolverse a través de un multiplexor cuyas líneas de control se alimentan de estos bits de índice

se utilizan para identificar, y leer, un dato dentro de un bloque. Básicamente el índice referencia la posición del dato dentro del bloque al que pertenece.

Conclusión
A modo de conclusión es importante relacionar la motivación de contar con una caché y su manera de llevarse a la práctica (su algoritmo). De esta forma, podemos decir que el manejo de bloques respeta el **principio de localidad espacial**, dado que al almacenar en la caché uno de los bloques de datos, será muy probable que próximamente se necesiten otras celdas del mismo bloque.

A nivel general, el funcionamiento con una memoria caché implica que ésta resuelva cuáles celdas son las más accedidas, y lo haga de manera transparente a la CPU. En otras palabras, la CPU se debe abstraer de la lógica de funcionamiento de la memoria caché. De ahí que se denomina **"escondida"**.

14.3. Funciones de correspondencia

Como ya hemos adelantado, las funciones de correspondencia proponen distintos criterios para corresponder los bloques de memoria principal con las líneas de memoria caché, es decir, que indican de alguna manera con qué flexibilidad se almacenan los bloques en las líneas.
Estas funciones de correspondencia intervienen en la forma que utiliza la caché para almacenar la información (copiada de la memoria principal) y, como consecuencia, en la manera de resolver si una celda está cacheada o no. Así es que la caché contará con diferentes mecanismos de operar en base a la función de correspondencia que disponga la arquitectura.

Las funciones de correspondencias son:

- Correspondencia Asociativa
- Correspondencia Directa
- Correspondencia asociativa por conjuntos

Antes de avanzar recordemos que si bien hemos mencionado muchos conceptos, en términos prácticos la información almacenada no son más ni menos que cadenas de bits. Por lo que las distinciones en la manera de operar con cada función de correspondencia, hace referencia a cómo utilizará la caché dichos bits. Por este motivo es que cada función de correspondencia, debido a sus características, distribuye los bits de la dirección que recibe de una manera diferente.

14.3.1. Correspondencia asociativa

La correspondencia asociativa se caracteriza por asociar cualquier bloque de la memoria principal con cualquier línea de caché. Además será necesario identificar dicho bloque de alguna manera, para lo cual se utiliza su número. Así es que, con el fin de poder identificarlo entre sus líneas, el nro de bloque también será almacenado en una línea (cual etiqueta) junto con los datos del bloque, para posteriormente resolver si una celda está cacheada o no. A esta etiqueta,

se la conoce por su término en inglés como **tag**, donde, en la función asociativa el tag identifica al nro de bloque, y como está relacionado con las direcciones de las celdas que componen su bloque, podemos decir que deriva de ellas.

Por otro lado, debido a que en esta correspondencia un bloque se puede almacenar en cualquier línea, para saber si una celda está cacheada o no, **se deberá buscar su nro de bloque (tag) en todas las líneas simultáneamente, es decir a la vez.**

Distribución de los bits de una dirección

Así como se utiliza el tag para identificar el nro de bloque, recordemos que se utiliza el índice para identificar el dato dentro del bloque. Con esta información podemos decir que ciertos bits de la dirección se destinarán para el tag y otros para el índice. Quedando un esquema como:

N bits de dirección	
n bits para tag (nro bloque)	m bits para índice

Para implementar dicho esquema será necesario realizar algunos cálculos para distribuir los bits de la dirección. El cálculo de manera generalizada es:

$$\frac{\text{cantidad de celdas}}{\text{cantidad de celdas por bloque}} = \text{cantidad de bloques}$$

Siendo que el objetivo no es conocer dichos valores, sino la cantidad de bits que se destinarán para cada uno (tag e índice), conviene expresar estas cantidades en términos de potencias, ya que el exponente indica la cantidad de bits necesarios para representar tal valor.

Entonces, podemos generalizar la fórmula anterior como:

$$\frac{2^n \text{ celdas}}{2^m \text{ celdas x bloque}} = 2^{(n-m)} \text{ bloques}$$

Al cancelarse la unidad "celdas" del numerador con la unidad "celdas" del denominador, obtenemos como resultado la cantidad de bloques finales. Pero tal como hemos mencionado, no son estos valores los que nos interesan, sino su cantidad de bits. Por lo tanto, del denominador (celdas por bloque) podemos obtener que m es la cantidad de bits para el índice, y $(n - m)$ la cantidad de bits para el tag (revisar la definición de cada concepto para profundizar).

Veamos cómo se aplica este cálculo en un ejemplo, para lo cual debemos conocer las características del subsistema de memoria a emplear. Para comenzar con un ejemplo sencillo, en lugar de utilizar la arquitectura Q, vamos a considerar un sistema más pequeño.
Se cuenta con una memoria principal con **direcciones de 6 bits** y capacidad de almacenamiento de 8 bits por celda. Y una memoria caché con 4 líneas, donde

cada línea almacena un **bloque de 4 celdas**.
Así los cálculos de distribución de bits quedan:

$$\frac{2^6 \text{ celdas}}{2^2 \text{ celdas x bloque}} = 2^{(6-2)} \text{ bloques} = 2^4 \text{ bloques}$$

De aquí podemos deducir varias cuestiones que nos permitan conocer la cantidad de bits tanto para el tag como para el índice.
Las 64 celdas (2^6) divididas en bloques de 4 celdas, resultando 16 bloques en total (2^4). Por lo que se van a necesitar 4 bits para el identificar a cada bloque, es decir al tag. Por otro lado, serán necesarios 2 bits para identificar a cada uno de los 4 datos del bloque (2^2), es decir para el índice.

Quedando la distribución de bits de la siguiente manera:

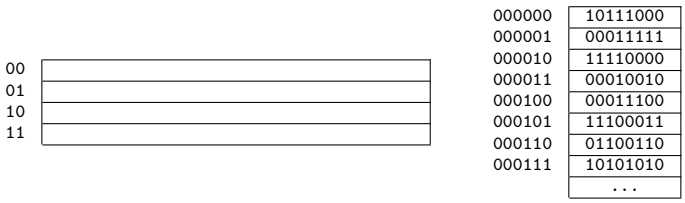
6 bits: Dirección	
4 bits: Tag (Nro bloque)	2 bits: Índice

Los 16 bloques se enumerarán desde el `B0` (`0000`) al `B15` (`1111`).

Ejemplos de accesos a memoria cache en correspondencia asociativa

Veamos ahora varios ejemplos de aplicación para la arquitectura mencionada en la sección anterior y la distribución de los bits de dirección.

Ejemplo 1
A continuación un mapa de memoria principal y de memoria caché, la cual inicia vacía. Se solicita a este subsistema de memoria la lectura de una celda:



Notar que: la caché tiene 4 líneas, por lo que se necesitan 2 bits para enumerarlas ($4 = 2^2$).

Para responder a esta petición, el subsistema deberá aplicar el algoritmo de lectura en caché.

1. Se recibe la dirección `000010` con el objetivo de retornar el dato de dicha celda. De esta dirección sabemos, por la distribución de bits previa, que los primeros 4 bits hacen referencia al tag mientras que los 2 bits restantes al índice:
$$\frac{0000}{Tag}\ \frac{10}{Indice}$$
2. Se debe resolver si la celda está cacheada. Por lo que la caché deberá buscar el tag `0000` en todas sus líneas simultáneamente. Conceptualmente el nro de bloque es el `B0`.

3. Al no encontrar dicho tag, ocurre un **fallo**. Así, la caché debe solicitar a la memoria principal dicha celda y almacenarla junto con su bloque correspondiente. Las celdas que comparten el bloque `B0` son:

000000	10111000
000001	00011111
000010	11110000
000011	00010010
	...

Este bloque se almacenará, en este caso, en la línea `00`. Quedando la caché:

00	0000 10111000 00011111 11110000 00010010
01	
10	
11	

El orden de almacenamiento de las celdas inicia con el tag (resaltado en verde) y luego el bloque con los 4 datos.
Nota: tener presente que no existen espacios entre los datos. En el mapa los mismos se encuentran separados a modo de mejorar la legibilidad y comprensión de la información.

Finalmente, el subsistema de memoria puede retornar el dato solicitado, que en este caso, al tratarse de un fallo lo obtuvo a partir de acceder a la memoria principal antes de copiar el bloque. Así, el dato enviado es `11110000`.

Ejemplo 2

En este ejemplo continuamos con el mismo subsistema de memoria, y por ende, la misma distribución de bits. El escenario en este caso prosigue a partir de la solicitud anterior, de manera que la memoria caché ya no se encuentra vacía.
En esta oportunidad se requiere la lectura de la celda con dirección `000011`. Así que a partir del mapa de memoria dado, aplicaremos nuevamente el algoritmo de lectura en caché.

00	0000 10111000 00011111 11110000 00010010
01	
10	
11	

000000	10111000
000001	00011111
000010	11110000
000011	00010010
000100	00011100
000101	11100011
000110	01100110
000111	10101010
	...

1. Dada la lectura de la dirección `000011`, entonces tenemos que:

$$\underset{Tag}{\overline{0000}}\ \underset{Indice}{\overline{11}}$$

2. ¿La celda está cacheada? Para ello se busca el tag `0000` en todas las líneas de manera simultánea.

3. La caché encuentra el tag `0000` en la línea `00` por lo que ocurre un **acierto**. Así, la caché está en condiciones de retornar el dato, para lo cual deberá identificarlo dentro del bloque y así poder entregarlo. Recordemos que

para ello se utiliza el **índice**, que en este caso corresponde con los últimos 2 bits de la dirección. Veamos cómo está compuesto el bloque encontrado:

10111000	00011111	11110000	00010010
00	01	10	11

Así entonces, a partir del índice `11` proyecta el dato y lo envía como respuesta: `00010010`.

A partir de este momento, con cada fallo, la caché irá llenando sus líneas disponibles, una por una, en orden consecutivo.

Ejemplo 3

Ya hemos analizado un fallo y un acierto en casos donde la caché aún tenía líneas disponibles. En este ejemplo nos centraremos en el caso opuesto. Veamos qué sucede cuando la caché se encuentra llena. Aquí el mapa de memoria:

0	00	0000	10111000	00011111	11110000	00010010
0	01	1010	10011101	00100111	11101010	10010110
1	10	0000	01010101	11100111	01010101	10011001
0	11	0110	10001111	10101010	00001111	10101010

000000	10111000
000001	00011111
000010	11110000
000011	00010010
000100	00011100
000101	01100001
000110	01100110
000111	10101010
	...

Antes de continuar con el algoritmo, es importante notar 2 cuestiones:
Por un lado, en la memoria caché se observa 1 bit previo al nro de línea, el cual se utilizará para la resolución de este escenario particular, y a detallar más adelante.
Por otro lado, notar en la memoria principal el patrón resaltado con colores. Los colores verde y azul hacen referencia a los distintos bloques, mientras que el rojo referencia al índice. En verde el bloque `B0` (`0000`), y en azúl el bloque siguiente `B1` (`0001`). Notar que las direcciones de las 4 celdas que componen cada bloque inician con la misma cadena de 4 bits. Es notoria la cadena binaria que se utiliza como (`tag`) y cuándo cambia cada bloque.
Pero *¿por qué los bits de índice están resaltados con el mismo color?*
Pues porque repiten el mismo patrón con cada bloque que inicia. Dado que todos los bloques se conforman de 4 celdas, la posición (`índice`) que ocupa cada dato dentro del mismo está dada siempre por las mismas combinaciones: `00`,`01`, `10`, `11`.
Aclaración: la cantidad de bits mencionados refieren al subsistema de memoria puesto como ejemplo. En la arquitectura Q este esquema se respeta pero con su respectiva distribución de bits.

Ahora sí retomamos el ejemplo en cuestión:

1. Se solicita la lectura de la celda con dirección `000101`. Por lo que:

0001	01
Tag	*Indice*

2. ¿La celda está cacheada? Para ello la caché busca el tag `0001` en todas las líneas de manera simultánea.

3. El tag no se encuentra en ninguna línea, por lo que ocurre un **fallo**. Así es que la caché solicita a la memoria principal el bloque con la celda correspondiente. Pero, a diferencia del ejemplo 1, en este escenario la caché no cuenta con líneas disponibles, entonces *¿cómo resuelve esta situación?* Lo que se debe realizar es un **desalojo**, es decir, desalojar un bloque de alguna línea para reemplazarlo por el que se necesita. A partir de aquí, surge una nueva pregunta *¿qué bloque se desaloja?*
Para resolver esta situación existen los **algoritmos de reemplazo**. Hay 4 algoritmos que veremos en la próxima sección.

   Para este escenario vamos a elegir el **algoritmo de reemplazo FIFO**, que básicamente consiste en reemplazar el bloque que fue almacenado en primer lugar. Este bloque es el marcado con el bit previo al tag, al inicio de la línea. En este caso dicho bloque es el correspondiente a la línea `01`. Así es que se elimina dicho bloque y se almacena el nuevo bloque `B1` cuyo tag es `0001`.
   Veamos 2 mapas de caché, el primero con el estado previo al desalojo y el segundo con el estado actualizado:

   Estado previo:
   En color naranja el bloque a reemplazar según el algoritmo FIFO:

0	00	0000 10111000 00011111 11110000 00010010
0	01	1010 10011101 00100111 11101010 10010110
1	10	0000 01010101 11100111 01010101 10011001
0	11	0110 10001111 10101010 00001111 10101010

   Estado actualizado:

0	00	0000 10111000 00011111 11110000 00010010
0	01	0000 01010101 11100111 01010101 10011001
0	10	0001 00011100 01100001 01100110 10101010
1	11	0110 10001111 10101010 00001111 10101010

   Notar que: además de haberse reemplazado el bloque en la línea `10`, también se actualizó el bit de reemplazo al bloque de la siguiente línea.

   Por último, el subsistema de memoria puede retornar el dato solicitado, que en este caso se obtuvo directamente de la memoria principal antes de realizar el desalojo y su posterior almacenamiento con el nuevo bloque. Así, el dato enviado es el `01100001`.

   Importante: cuando ocurre un fallo, no se vuelve a leer en la caché, y por tanto no se utiliza el índice para retornar el dato solicitado, dado que se aprovecha el dato leído durante el acceso a memoria principal.

14.3.2. Correspondencia Directa

La implementación de una correspondencia asociativa tiene un costo elevado debido a que la búsqueda del tag implica acceder a todas las líneas en forma simultánea, lo cual requiere una tecnología de acceso aleatorio (similar a la de memoria principal). Para reducir este costo es posible presentar otro mecanismo de correspondencia donde cada bloque de memoria principal esté asignado a una línea determinada, y así reducir el costo en su búsqueda.

Dado que la cantidad de líneas es mucho menor a la cantidad de bloques (porque en otro caso la memoria principal y la caché tendrían la misma capacidad), a cada línea le corresponde un conjunto de bloques **candidatos** que "compiten" entre si. A esta función se la denomina **correspondencia directa**, donde cada línea de caché es considerada como un recurso compartido entre un conjunto de bloques de memoria. Es importante destacar que, con el objetivo de aprovechar los principios de localidad y maximizar la tasa de aciertos, este conjunto de bloques asociado a una misma línea no es consecutivo.
El criterio que se utiliza para corresponder bloques con líneas es **circular**, tal como se muestra a continuación en la figura 14.4:

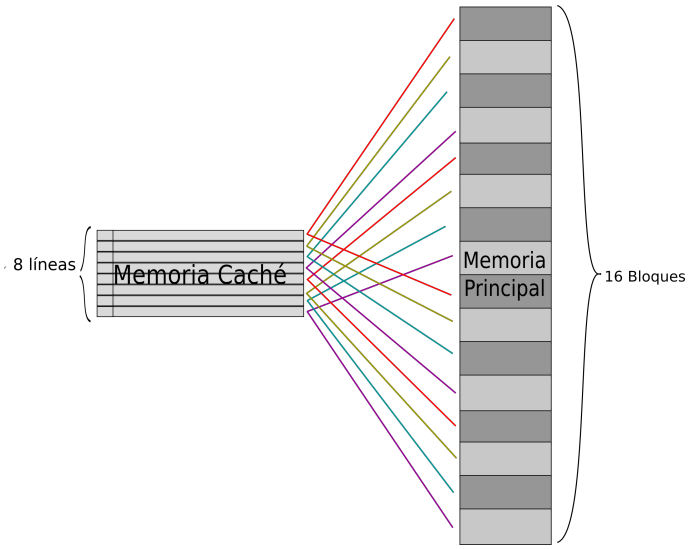


Figura 14.4: Función de correspondencia directa

En el gráfico se observa en colores los bloques que compiten para almacenarse en cada línea. Notar que cada 4 celdas se vuelven a repetir los mismos colores, y ocurre un patrón matemático debido al esquema circular.

Debido a las características mencionadas previamente, en esta función de correspondencia cada bloque se asocia **directamente** a una línea específica (de allí su nombre) a diferencia de la correspondencia asociativa, donde los bloques se pueden almacenar en cualquier línea. Este rasgo requiere una nueva forma de buscar (leer) en caché y por ende de almacenar la información.
La caché debe buscar el bloque **únicamente en la línea que le corresponde**. De esta manera, al buscarse en una sola línea se simplifica la tecnología para implementar la caché y por ende se reduce su costo.
Pero, a partir de este mecanismo ya no alcanza con almacenar el nro de bloque, dado que no hay uno solo, sino varios candidatos por línea, y por tanto será necesario indicar cuál de ellos es. Para esto se utiliza el **tag**. Éste actúa como una etiqueta que indica cuál es el bloque, dentro del subconjunto de candidatos, que se almacenó. De esta manera, el tag ya no coincide con el nro de bloque como en la correspondencia asociativa, no son equivalentes.

Por ejemplo, supongamos que hay 16 bloques (`B0, B1, ...B15`) que se reparten en 8 líneas, entonces hay 2 candidatos por línea (ver figura 14.4). En la línea `0` los candidatos son: el nro de bloque 0 (`B0`), y el nro 8 (`B8`). Como se puede observar, dicha línea puede contener cualquiera de los dos. Entonces, dada una dirección, *¿cómo puede identificar la caché cuál de los dos se almacenó y resolver si la celda está cacheada?* Para poder distinguir uno del otro se utiliza el **tag**. En este caso, al contar con sólo 2 candidatos, el tag indicará si se trata del primero, el `0`, o del segundo, el `1`. Este es un ejemplo muy sencillo, pero de haber más candidatos resultaría más notorio el subconjunto de bloques formados.

En conclusión, **la lectura en caché se resuelve buscando el tag en la línea correspondiente a su bloque**. Por lo que, al copiarse el bloque de memoria principal, la caché almacena el tag (que no es nro de bloque) junto con los datos correspondientes.

Distribución de los bits de una dirección

A partir de los conceptos mencionados, la manera en que se distribuyen los bits en la correspondencia directa difiere de la correspondencia asociativa.
La dirección además de proveer el nro de bloque y el índice, también informa el tag y la línea que le corresponde a dicho bloque, y para esto toma una porción de bits del nro del bloque. De esta forma, la distribución de los bits queda organizada de la siguiente manera:

<table>
<tr><td colspan="3">N bits: Dirección</td></tr>
<tr><td colspan="2">n bits: Nro bloque</td><td>n bits: Índice</td></tr>
<tr><td>n bits: Tag</td><td>n bits: Línea</td><td></td></tr>
</table>

En esta ocasión los cálculos requieren un paso más para obtener la cantidad de bits destinados a la línea.

De manera generalizada:

Paso 1: mismo paso que en la asociativa para los bloques

$$\frac{\text{cantidad de celdas}}{\text{cantidad de celdas por bloque}} = \text{cantidad de bloques}$$

Paso 2: nuevo paso para los bloques candidatos (subconjunto)

$$\frac{\text{cantidad de bloques}}{\text{cantidad de líneas}} = \text{cantidad de bloques por línea}$$

Versión en términos de potencia para obtener la cantidad de bits:

Paso 1: se calculan los bloques totales

$$\frac{2^n \text{ celdas}}{2^m \text{ celdas x bloque}} = 2^{(n-m)} \text{ bloques}$$

Paso 2: se calculan los bloques candidatos (el tag)

$$\frac{2^n \text{ bloques}}{2^m \text{ líneas}} = 2^{(n-m)} \text{ bloques x línea}$$

Retomando el ejemplo

Considerar una memoria principal con direcciones de 6 bits, bloques de 4 celdas con capacidad de 8 bits por celda, y con una memoria caché con 8 líneas. Así, la distribución de bits es:

Paso 1:

$$\frac{2^6 \text{ celdas}}{2^2 \text{ celdas x bloque}} = 2^{(6-2)} \text{ bloques} = 2^4 \text{ bloques}$$

Paso 2:

$$\frac{2^4 \text{ bloques}}{2^3 \text{ líneas}} = 2^{(4-3)} \text{ bloques x línea} = 2^1 \text{ bloques por línea}$$

De este calculo se deduce que:

6 bits: Dirección		
4 bits: Nro bloque		2 bits: Índice
1 bit: Tag	3 bits: Línea	

La memoria principal tiene `64` celdas `(`2^6`)` divididas en bloques de 4 celdas (2^2), se obtienen `16` bloques en total `(`2^4`)`, los cuales a su vez se distribuyen entre las `8` líneas `(`2^3`)`, quedando 2 bloques candidatos por línea (2^1).

Entendiendo esta situación se puede analizar la distribución de bits de la dirección como: de los 6 bits, los primeros 4 hacen referencia al nro de bloque, pero de estos, 1 bit se utiliza para el tag [3], puesto que en cada línea se almacenarán 2 posibles bloques, por lo que alcanza con un sólo bit para distinguir entre ellos. Del nro de bloque, además, se utilizan 3 bits para identificar a qué línea corresponde el bloque. Finalmente, se utilizan los 2 últimos bits de la dirección para el índice (bloques de 4 celdas).

Notar que: en ambas funciones de correspondencia se almacena el tag y los datos correspondientes al bloque. La diferencia radica en, que en la correspondencia asociativa el nro de bloque coincide con el tag, mientras que en la directa, éstos representan conceptos diferentes, donde el tag es un subconjunto de los bloques.

[3] Recordar que en esta función el nro de bloque no coincide con el tag

Ejemplos de accesos a memoria caché (correspondencia directa)

En esta sección veremos los mismos 3 ejemplos que analizamos para la función asociativa, pero aplicados a la función de correspondencia directa.

Ejemplo 1:
A continuación un mapa de memoria principal y de memoria caché, la cual inicia vacía. Se solicita a este subsistema de memoria la lectura de una celda:

000	
001	
010	
011	
100	
101	
110	
111	

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000000
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

Para responder a esta petición, el subsistema deberá aplicar el algoritmo de lectura en caché.

1. Se recibe la dirección `011010`. De esta dirección sabemos, por la distribución de bits previa, que los primeros 4 bits hacen referencia al nro de bloque, del cual se extra 1 bit para el tag y 3 para la línea. Los últimos 2 bits de la dirección para el índice:

$$\underset{NroBloque}{0110}\ \underset{Indice}{10}$$

   Donde:

$$\underset{Tag}{0}\ \underset{Linea}{110}$$

2. Se debe resolver si la celda está cacheada. Por lo que la caché **busca el tag `0` en la línea `110`.**

3. Al no encontrar el tag en dicha línea, ocurre un **fallo**. Así, la caché debe solicitar a la memoria principal dicha celda y almacenarla junto con su bloque correspondiente. Las celdas que comparten el bloque nro `0110` son:

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000000
	...

   El bloque se almacenará en la línea `110`. Quedando la caché:

000	
001	
010	
011	
100	
101	
110	0 10111000 00011111 01010101 00000000
111	

   Finalmente, el subsistema de memoria puede retornar el dato solicitado, que en este caso, al tratarse de un fallo lo obtuvo a partir de acceder a la memoria principal antes de copiar el bloque. Así, el dato enviado es `01010101`.

Ejemplo 2

El escenario en este caso prosigue a partir de la solicitud anterior, de manera que la memoria caché ya no se encuentra vacía.
En esta oportunidad se requiere la lectura de la celda con dirección `011011`. Así que a partir del mapa de memoria dado, aplicaremos nuevamente el algoritmo de lectura en caché.

000	
001	
010	
011	
100	
101	
110	0 10111000 00011111 01010101 00000000
111	

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000000
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

1. Dada la lectura de la dirección `011011`, entonces tenemos que:

$$\underset{NroBloque}{0110}\ \underset{Indice}{11}$$

   Donde:

$$\underset{Tag}{0}\ \underset{Linea}{110}$$

2. ¿La celda está cacheada? Para ello busca el tag `0` en la línea `110`.

3. Como vemos en el mapa de caché, en este caso ocurre un **acierto**. Por lo que la caché retorna el dato a través del índice. Veamos cómo está compuesto el bloque encontrado:

$$\underset{00}{00011111}\ \underset{01}{00011111}\ \underset{10}{01010101}\ \underset{11}{00000000}$$

   Así entonces, a partir del índice `11` proyecta el dato y lo envía como respuesta: `00000000`.

Ejemplo 3

En este ejemplo nos vamos a centrar en el caso donde la caché está completa, y así poder comparar esta situación con la correspondencia asociativa. Aquí el mapa de memoria:

000	0 11010010 00001001 00101110 10001001
001	1 11010001 01000000 00001011 11111111
010	1 01110101 00011010 00001110 01010001
011	0 10000111 00011111 01111000 10011000
100	0 00000001 10000000 11001100 00110011
101	1 00011100 11100011 10101010 00011001
110	0 10111000 00011111 01010101 00000000
111	1 00010101 00000010 01111100 00011000

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000000
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

1. Se solicita la lectura de la celda con dirección `011100`. Por lo que:

$$\underset{NroBloque}{0111}\ \underset{Indice}{00}$$

$$\underset{Tag}{0}\ \underset{Linea}{111}$$

2. ¿La celda está cacheada? Para ello la caché busca el tag `0` en la línea `111`.

3. En dicha línea no se encuentra ese tag, por lo que ocurre un **fallo**. Así es que la caché solicita a la memoria principal el bloque con la celda correspondiente. Pero a diferencia del ejemplo 1, la línea está ocupada, por lo que se debe realizar un **desalojo**. En la correspondencia asociativa, un desalojo deriva en la necesidad de contar con un criterio para decidir qué línea reemplazar. Pero en esta correspondencia, ésto no es necesario, dado que a cada bloque le corresponde una línea, y por tanto, dicha línea es la que será reemplazada. Así es que tampoco requiere de un bit extra para esta operación. En este ejemplo, se deberá reemplazar el bloque de la línea `111`.

Estado previo:

000	0	11010010	00001001	00101110	10001001
001	1	11010001	01000000	00001011	11111111
010	1	01110101	00011010	00001110	01010001
011	0	10000111	00011111	01111000	10011000
100	0	00000001	10000000	11001100	00110011
101	1	00011100	11100011	10101010	00011001
110	0	10111000	00011111	01010101	00000000
111	1	00010101	00000010	01111100	00011000

Estado nuevo:

000	0	11010010	00001001	00101110	10001001
001	1	11010001	01000000	00001011	11111111
010	1	01110101	00011010	00001110	01010001
011	0	10000111	00011111	01111000	10011000
100	0	00000001	10000000	11001100	00110011
101	1	00011100	11100011	10101010	00011001
110	0	10111000	00011111	01010101	00000000
111	0	00011100	11100011	01100110	10101010

Por último, el subsistema de memoria puede retornar el dato solicitado, que en este caso se obtuvo directamente de la memoria principal antes de realizar el desalojo y su posterior almacenamiento con el nuevo bloque. Así, el dato enviado es el `00011100`.

14.3.3. Correspondencia asociativa por conjuntos

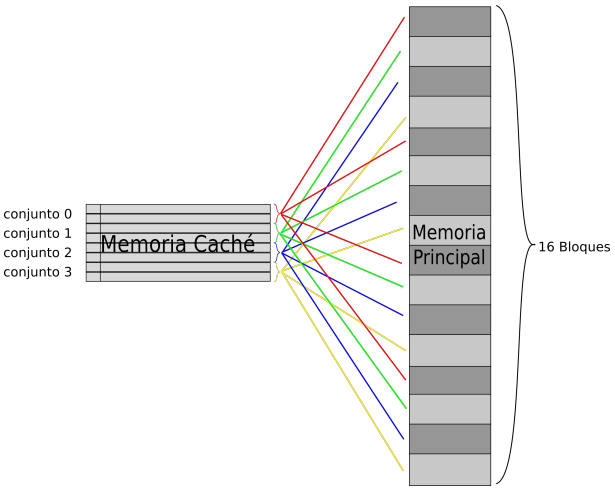


Figura 14.5: Correspondencia asociativa por conjuntos

A la hora de comparar los enfoques anteriores, se ve que la correspondencia directa es mas económica en su construcción pero la correspondencia asociativa es mas flexible y maximiza el porcentaje de aciertos. Para hacerlo evidente, suponer una secuencia de accesos que requiera repetidamente cachear bloques que corresponden a una misma línea, causando repetidos fallos. En una correspondencia asociativa, esos bloques no compiten y no se darían mas fallos que los que producen bloques nuevos.

En una búsqueda de balancear los dos aspectos mencionados (eficiencia vs. costo), se propone una **correspondencia asociativa por conjuntos**, donde la memoria caché se divide en conjuntos de N líneas y a cada bloque le corresponde uno de ellos. Es decir que dentro del conjunto de líneas asignado, un bloque de memoria principal puede ser alojado en cualquiera de las N líneas que lo forman, es decir que dentro de cada conjunto la caché es totalmente asociativa.

Esta situación es la más equilibrada, puesto que se trata de un compromiso entre las técnicas anteriores: si N es igual a 1, se tiene una caché de mapeo directo, y si N es igual al número de líneas de la caché, se tiene una caché completamente asociativa. Si se escoge un valor de N apropiado, se alcanza la solución óptima.

En la figura 14.5 se ejemplifica esta nueva forma de correspondencia. Con este mecanismo, las lineas se agrupan en conjuntos, y cada conjunto se corresponde con determinados bloques de la memoria principal. Además, dentro de cada conjunto los bloques se almacenan con un criterio asociativo. Cuando la caché recibe una dirección, debe determinar a que conjunto corresponde el bloque de la celda buscada y luego buscar asociativamente dentro del conjunto el tag que corresponde.

Suponer una computadora como la de la figura 14.5, con 16 bloques de 4 celdas en la memoria principal, y una memoria caché con 4 conjuntos de 2 linea cada uno. Por ejemplo, el conjunto `00` podría almacenar los bloques `0000`,`0100`,`1000` y `1100`. Si se pide la lectura de la celda `110011`, que corresponde al bloque `1100`, entonces se deben analizar las lineas del conjunto `00`, buscando el tag `11`.

14.4. Políticas de escritura

Hasta ahora hemos visto cómo funciona la memoria caché ante una solicitud de lectura. Pero como ya sabemos, durante la ejecución de los programas se necesita realizar tanto lecturas como escrituras. Por ejemplo durante la etapa de búsqueda de instrucción se realiza una lectura, pero durante la etapa de almacenamiento de resultado se realiza una escritura, así como también durante la etapa de ejecución para apilar un dato al ejecutar la instrucción `CALL`. Un ejemplo de almacenamiento de resultado es la instrucción `MOV [0xACCA], 0x0001`.

Siendo que el subsistema de memoria cuenta con 2 memorias, será necesario disponer de alguna estrategia que permita mantener los datos actualizados en ambas memorias, según se necesite. Es decir para manejar las diferentes versiones entre ambas copias del dato modificado (versión original en memoria principal y el dato alterado en memoria caché). Para ello se cuenta con 2 políticas de escritura: *write-through* y *write-back*.

Política write-through

También conocida como escritura simultánea, inmediata o en directo. Esta consiste en que cada vez que un dato se modifica en memoria caché, se debe actualizar inmediatamente en memoria principal. Esto garantiza que en todo momento los datos de ambas memorias son iguales, aspecto importante cuando la memoria principal es compartida por varios procesadores.

Política write-back

También conocida como escritura ascincrónica, o en diferido. En este caso, la actualización se difiere (se retrasa) hasta el momento en que el bloque que contiene el dato alterado por la escritura sea el elegido para el desalojo (ya sea por correspondencia directa o asociativa). Esta estrategia requiere que se lleve un registro del dato alterado en la memoria caché (o del bloque que lo contiene) para actualizar la memoria principal sólo en caso de ser necesario. Para ello se asocia a cada línea de caché un bit denominadoo *dirty bit* (del inglés: indicador de bloque sucio), el cual indica en 1 que esa línea contiene una versión diferente del dato, y en 0 el caso contrario.

Es importante destacar que para escribir en una línea de caché, previamente se deberá identificar la línea donde se encuentra el dato, y por tanto será necesario resolver si la celda se encuentra cacheada o no. Por lo que se aplica el mismo algoritmo de lectura en caché para poder acceder al dato en cuestión, y luego proceder con la política de escritura correspondiente.

Veamos un ejemplo sencillo de cada política, retomando la arquitectura mencionada en las secciones anteriores, con el ejemplo de la correspondencia directa.

Ejemplos de write-through

Ejemplo 1:

Se solicita la escritura del dato `00000001` en la celda con dirección `011011`. El mapa de memoria con el que se cuenta hasta el momento contiene:

000	
001	
010	
011	
100	
101	
110	0 10111000 00011111 01010101 00000000
111	

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000000
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

Se busca si la celda está cacheada. En este caso ocurre un **acierto** en la línea `110`, por lo que, primero se deberá modificar (escribir) en la memoria caché el dato `00000001` en la posición indicada por el índice `11`, y luego actualizarlo en la memoria principal. Quedando:

Paso 1:

000	
001	
010	
011	
100	
101	
110	0 10111000 00011111 01010101 00000001
111	

Paso 2:

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000001
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

Ejemplo 2:

En caso de solicitar la escritura en una celda que no se encuentra cacheada, es decir que ocurra **Fallo**, el orden es el inverso. Primero se modifica el dato en la memoria principal y luego se cachea el bloque de memoria principal con el dato ya actualizado.

Por ejemplo, continuando con el mapa anterior, ahora se solicita escribir el dato `00000010` en la dirección `011101`. Ocurre un fallo, dado que el tag `0` no se encuentra en la línea `111`. Por lo que:

Paso 1: se actualiza el dato en la memoria principal

Antes:

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000001
011100	00011100
011101	11100011
011110	01100110
011111	10101010
	...

Después:

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000001
011100	00011100
011101	00000010
011110	01100110
011111	10101010
	...

Paso 2: se almacena el bloque con el dato actualizado

000	
001	
010	
011	
100	
101	
110	0 10111000 00011111 01010101 00000001
111	0 00011100 00000010 01100110 10101010

Ejemplos de write-back

Ejemplo 1:

A partir del mapa anterior, veremos por ejemplo que ahora se solicita la escritura del dato `10000000` en la celda con dirección `011000`. Ocurre un **Acierto**, dado que tag `0` se encuentra en la línea `110`. Recordar que para esta política, la caché incorpora el *dirty-bit* como indicador del bloque con el dato actualizado.

Se actualiza el dato en caché y su correspondiente dirty-bit. La memoria principal por el momento no se modifica.

000	0	
001	0	
010	0	
011	0	
100	0	
101	0	
110	1	0 10000000 00011111 01010101 00000001
111	0	0 00011100 00000010 01100110 10101010

	...
011000	10111000
011001	00011111
011010	01010101
011011	00000001
011100	00011100
011101	00000010
011110	01100110
011111	10101010
	...

Ejemplo 2:

Ahora se solicita la escritura del dato `11110000` en la celda con dirección `100010`. Ocurre un **Fallo**, dado que tag `1` no se encuentra en la línea `000`, por lo que se trae el bloque completo de la memoria principal y se lo almacena en caché, para luego modificar el dato en cuestión y actualizar su dirty-bit.
Finalmente queda como:

000	1	1 11001010 01100010 11110000 01010101
001	0	
010	0	
011	0	
100	0	
101	0	
110	1	0 10000000 00011111 01010101 00000001
111	0	0 00011100 00000010 01100110 10101010

	...
011100	00011100
011101	00000010
011110	01100110
011111	10101010
100000	11001010
100001	01100010
100010	00011101
100011	01010101
	...

Nuevamente no se modifica la memoria principal. Esto no genera un problema dado que en caso de solicitarse una lectura de dicha celda, primero se busca en caché, y por tanto se retornará el dato correcto (actualizado).
Veamos a continuación una situación, de manera general, que sí podría generar un problema. Para ello nos vamos a abstraer por un momento de la arquitectura propuesta.

1) Se solicita escribir el dato `B` en una dirección `x`. Se actualizó dicho dato en la caché, en la línea `0`, y se marcó el dirty-bit en `1`. En la memoria principal, la dirección x aún tiene el dato anterior `A`.

2) Luego, se solicita una lectura que ocasiona un fallo y se debe desalojar la línea `0`, almacenando un nuevo bloque, por lo que se perdió el dato modificado `B`.

¿Qué sucede si se solicita una lectura de la celda con dirección `x` para obtener el dato `B`?

Al buscar en caché habrá un fallo, y al traer el bloque correspondiente de memoria principal, el dato a retornarse será el `A`. Pero este es incorrecto, puesto que el programa que se está ejecutando ya lo había modificado.
Para resolver este problema es que primero se necesita actualizar el dato en memoria principal, escribiendo todo el bloque completo, antes de desalojar el bloque de memoria caché y reemplazarlo con uno nuevo. De esta manera, la próxima vez que se requiera la celda con dicho dato se podrá cachear el bloque con el dato correcto para su posterior retorno.

Ejemplo 3:

A partir del mapa de memoria dado a continuación, se solicita la lectura de la celda con dirección `000001`. El tag `0` no se encuentra en la línea `000`, es decir que ocurre un fallo. Pero dicha línea se encuentra ocupada, por lo que habrá que desalojar el bloque. Para esto, primero se debe tener el cuenta el estado del dirty-bit. En este caso el mismo se encuentra activo, por lo que antes de desalojar se debe actualizar el dato (su bloque) en memoria principal, para luego reemplazar el bloque solicitado, además de desactivar el dirty-bit.
Notar que el dato en caché, marcado en naranja, difiere del dato naranja de memoria principal.

000	1	1 11001010 01100010 11110000 01010101
001	0	
010	0	
011	0	
100	0	
101	0	
110	1	0 10000000 00011111 01010101 00000001
111	0	0 00011100 00000010 01100110 10101010

000000	11111111
000001	11001100
000010	00001100
000011	00110011
...	...
011100	00011100
011101	00000010
011110	01100110
011111	10101010
100000	11001010
100001	01100010
100010	00011101
100011	01010101
	...

Mapa de memoria actualizado paso a paso:

Paso 1:

000000	11111111
000001	11001100
000010	00001100
000011	00110011
...	...
011100	00011100
011101	00000010
011110	01100110
011111	10101010
100000	11001010
100001	01100010
100010	11110000
100011	01010101
	...

Paso 2:

000	0	0 11111111 11001100 00001100 00110011
001	0	
010	0	
011	0	
100	0	
101	0	
110	1	0 10000000 00011111 01010101 00000001
111	0	0 00011100 00000010 01100110 10101010

Tener en cuenta que si el dirty-bit está desactivado, no es necesario realizar el paso 1, simplemente se reemplaza un bloque por otro (paso 2).

14.5. Algoritmos de reemplazo

En la presente sección se detallará cada uno de los algoritmos de reemplazo relacionados a la función de correspondencia asociativa.

Cuando se produce un fallo y un nuevo bloque debe ser almacenado en la caché, debe elegirse una línea que puede o no contener otro bloque para ocupar. En el caso de correspondencia directa, no se requiere hacer tal elección, pues existe solo una posible línea para un determinado bloque.

Sin embargo, en los casos de las correspondencia asociativa y correspondencia asociativa por conjuntos, dado que ambos mecanismos aplican mayor o menor nivel de asociatividad, se necesita un criterio para elegir el bloque que será reemplazado. Con este objetivo se diseñan los **algoritmos de reemplazo** descriptos a continuación.

14.5.1. Algoritmo LRU (Least Recently Used)

El algoritmo más usado es el algoritmo *LRU*. Tiene la característica de que luego de cada referencia se actualiza una lista que indica cuan reciente fue la última referencia a un bloque determinado. Si se produce un fallo, **se reemplaza aquel bloque cuya última referencia se ha producido en el pasado más lejano**.

Para poder implementarlo, es necesario actualizar, luego de cada referencia, una lista que ordena los bloques por último acceso, indicando cuan reciente fue la última referencia a cada bloque. Si se produce un fallo, se reemplaza aquel bloque que se encuentra último en la lista, pero si ocurre un acierto, el bloque accedido debe convertirse en el primero de la lista.

Por ejemplo, considerar una memoria principal de 64 celdas (direcciones de 6 bits), y una memoria caché con 4 líneas y 8 celdas por bloque. La política LRU para una lista de accesos de ejemplo se muestra en la tabla 14.1

Dirección	N. Bloque	A/F	Lista de bloques
111000	111	Fallo	111
011001	011	Fallo	011,111
001111	001	Fallo	001,011,111
110000	110	Fallo	110,001,011,111
011111	011	Acierto	011,110,001,111
111111	111	Acierto	111,011,110,001
000111	000	Fallo	000,111,011,110
001111	001	Fallo	001,000,111,011

Tabla 14.1: Ejemplo LRU

En una caché completamente asociativa la lista es una sola, pero en una caché asociativa por conjuntos, cada conjunto debe mantener su propia lista ordenada. En ambos casos se necesita almacenar información extra para simular cada lista encadenada. Diversas simulaciones indican que las mejores tasas de acierto se producen aplicando este algoritmo.

14.5.2. Algoritmo FIFO (First In-First Out)

La política **FIFO** (el primero en entrar es el primero en salir) hace que, frente a un fallo, se siga una lista circular para elegir la línea donde se alojará el nuevo bloque. De esta manera, se elimina de la memoria cache aquel que fue almacenado en primer lugar y los bloques siguientes son removidos en el mismo orden, análogamente a lo que ocurre en una cola de espera en un banco.

Similarmente para el caso LRU, en una memoria totalmente asociativa se debe mantener el registro de una sola lista, y en una asociativa por conjuntos, una por cada conjunto.

Es posible observar que esta política no tiene en cuenta el principio de localidad temporal. Para hacerlo, considerar el caso donde un bloque es requerido repetidamente en una memoria caché como la del caso LRU. Cada 4 fallos, dicho bloque es reemplazado por otro, sin importar si es muy usado.

La implementación de la lista puede resolverse a través de un bit que se almacena en cada línea (además del tag y de los datos del bloque) que indica cuál es el bloque candidato a ser reemplazado. De esta manera, solo uno de los bloques tiene un 1, y los restantes deben tener 0. Cuando ocurre un fallo, el bloque que tenía un 1 se reemplaza, se resetea el bit y se actualiza el bit de 0 a 1 de la siguiente línea.

14.5.3. Algoritmo LFU (Least Frequently Used)

En el método **LFU** (*Least Frequently Used*) el bloque a sustituir será aquel que se acceda **con menos frecuencia**. Será necesario entonces registrar la frecuencia de uso de los diferentes bloques de caché. Esta frecuencia se debe mantener a través de un campo contador almacenado en cada línea, pero esto presenta una limitación en el rango de representación. Para mejorar esto, una posible solución es ir recalculando la frecuencia cada vez que se realice una operación en caché, dividiendo el número de veces que se ha usado un bloque por el tiempo que hace que entró en caché. La contraparte de este método es que el cálculo mencionado presenta un costo adicional elevado.

14.5.4. Algoritmo Aleatorio

Con este algoritmo, el bloque es elegido al azar. En una memoria asociativa por conjuntos, el azar se calcula entre los bloques que forman el conjunto en el cual se ha producido la falla. Este algoritmo es contrario al principio de localidad, pues lo desconoce; sin embargo, algunos resultados de simulaciones indican que al utilizar el algoritmo aleatorio se obtienen tasas de acierto superiores a los algoritmos FIFO y LRU.
Posee las ventajas de que por un lado su implementación requiere un mínimo de hardware y por otro lado no es necesario almacenar información extra en cada bloque.

14.6. Desempeño (performance) de la caché

El desempeño de una memoria Caché está dado por la cantidad de aciertos y fallos en un conjunto representativo de programas. Al diseñar una caché se deben proponer dimensiones (cantidad de líneas, capacidad de las líneas), así como funciones de correspondencia y algoritmos de reemplazo que permitan maximizar los aciertos para un conjunto de programas de prueba. Lo que se busca es que al ejecutar esos programas en una arquitectura con la caché diseñada, el tiempo de acceso sea menor que al ejecutarlos sin una memoria caché.

Suponer una máquina con arquitectura **Q6** con una memoria caché de correspondencia directa de 64 líneas y 4 celdas por bloque. Se tiene el siguiente programa ensamblado a partir de la celda con dirección `B110` y se sabe que `R1 = AC00`, `R3 = A702`, `SP=FFEE` y que la celda con dirección `A702` tiene la cadena 0x0001.

	...
B110	CMP [R3],
B111	0x0000
B112	JE fin
B113	ADD R0, [R1]
B114	ADD R2, [R3]
B115	fin: RET
	...

Se necesita analizar la performance de la caché en cuanto a la cantidad de fallos que se producen durante la ejecución del programa. Para eso se lleva registro de las direcciones que se solicitan al subsistema de memoria , y para cada una si se produjo un acierto o fallo. Ver la siguiente tabla:

dir(hexa)	dir (binario)	tag	linea	F/A
B110	1011000100010000	10110001	000100	F
B111	1011000100010001	10110001	000100	A
B112	1011000100010010	10110001	000100	A
B113	1011000100010011	10110001	000100	A
AC00	1010110000000000	10101100	000000	F
B114	1011000100010100	10110001	000101	F
A702	1010011100000010	10100111	000000	F
B115	1011000100010101	10110001	000101	A

En la ejecución se produjeron 8 accesos, de los cuales 4 fueron fallos, podemos decir que se tiene una **tasa de fallos** de $\frac{4}{8} = 0,5$, y una **tasa de aciertos** de $\frac{4}{8} = 0,5$. Por otro lado, si el tiempo de acceso a la memoria principal es de $0{,}5ms$ y el de la caché es de $0{,}05ms$. La ejecución de ese programa llevó en total:

$$4 * (0{,}5ms + 0{,}05ms) + 4 * 0{,}05ms = 2{,}4ms$$

Ahora si en el programa se logran reducir los fallos a 3 (de los 8 accesos). El tiempo de ejecución total se reduce a:

$$3 * (0{,}5ms + 0{,}05ms) + 5 * 0{,}05ms = 1{,}9ms$$

Con una **tasa de fallos** de $\frac{3}{8} = 0,375$, y una **tasa de aciertos** de $\frac{5}{8} = 0,625$.

De aquí, se puede generalizar el cálculo del tiempo de ejecución mediante la siguiente fórmula:
Sea:

- cf = cantidad de fallos
- ca = cantidad de aciertos
- tm = tiempo de acceso a la memoria principal
- tc = tiempo de acceso a la memoria caché
- tte = tiempo total de ejecución

tte = cf * (tm + tc) + ca * tc.

Conclusión

Con cada fallo se necesita acceder a ambas memorias, y por tanto se acumula la suma de ambos tiempos, mientras que en los aciertos se evitan los accesos a la memoria principal, cumpliendo así el objetivo de contar con una memoria que reduzca el tiempo de ejecución. Y por tal motivo, el tiempo de acceso a la caché es mucho menor que a la memoria principal, agilizando así el proceso de ejecución al contar con una memoria más pequeña pero más rápida.
En resumen, la caché funciona gracias al principio de localidad temporal y espacial, buscando un alto porcentaje de aciertos que agilicen la ejecución del procesador, haciendo que se actúe casi a la velocidad de la caché la mayor parte del tiempo, y no a la velocidad de la memoria principal.

Capítulo 15

Sistemas de numeración fraccionarios

Para representar números reales en una computadora es necesario tener una manera de codificar la çoma raíz". En este capítulo vamos a ver dos tipos de sistemas que nos permiten representar números fraccionarios.

15.1. Sistemas de Punto Fijo

El sistema numérico de punto fijo es una generalización de los sistemas enteros que permite representar números fraccionarios, esto se logra conviniendo la posición de la coma en un lugar fijo, separando de esta manera la cadena en dos partes: parte entera y parte fraccionaria. En un sistema de punto fijo, se destina una cierta cantidad de bits a la parte entera y el resto a la parte fraccionaria, considerando que existe un punto (o coma) que los separa, tal como en el sistema decimal. De esta manera, para poder representar la parte fraccionaria, es necesario sacrificar el rango de alcance de la representación. Es decir, cuantos más bits fraccionarios posea un sistema, mayor precisión y menor rango poseerá (para una misma cantidad total de bits). En los sistemas binarios de punto fijo, el punto no está representado explícitamente (es decir, que no se almacena como un bit), sino que se asume una posición determinada. Por ejemplo, se puede construir un sistema binario sin signo con punto fijo de 8 bits, considerando 5 bits para la parte entera y 3 bits para la parte fraccionaria.

En cuanto a la notación, además del valor **n** que describe la cantidad total de bits, se agrega un valor adicional **m** que hace referencia a la cantidad de bits fraccionarios, quedando cada sistema de la siguiente manera:

- BSS(n,m): sistema de punto fijo en BSS con n bits en total, de los cuales m son fraccionarios.
- SM(n,m): sistema de punto fijo en SM con n bits en total, de los cuales 1 es de signo, y m de parte fraccionaria. Esto implica que $n - m - 1$ corresponden a la parte entera.

Los bits fraccionarios, al igual que los enteros, tienen un peso, pero están asociados a una potencia de base 2 negativa, es decir que el exponente será negativo. Así, el primer bit fraccionario (de izquierda a derecha) tiene un peso de 2^{-1}, el segundo de 2^{-2}, y así sucesivamente hasta el último bit fraccionario.

15.1.1. Interpretación

Para **Interpretar** una cadena en punto fijo se tiene dos mecanismos. El mecanismo mecanismo por partes y el mecanismo escalado.

El *mecanismo por partes* requiere que se trabaje por un lado la parte entera en sistema BSS y por otro lado la parte fraccionaria, aplicando las potencias negativas mencionadas previamente.

Por ejemplo, queremos interpretar la cadena `10011101` en $BSS(8,3)$. Esto quiere decir que se toman los primeros 5 bits para la parte entera, y se interpretan en el sistema correspondiente, en este caso BSS(5), quedando las potencias para la parte entera como $2^4 + 2^1 + 2^0$, continuando con las potencias fraccionarias $2^{-1} + 2^{-3}$.
Así, la interpretación queda:

$$I_{BSS(8,3)}(10011101) = 2^4 + 2^1 + 2^0 + 2^{-1} + 2^{-3} =$$
$$= 16 + 2 + 1 + 0,5 + 0,125 = 19,625$$

Por otra parte el *Mecanismo Escalado* aprovecha la interpretación BSS adaptando el valor que se obtiene a la posición de la **coma desplazada**, es decir, m lugares. Este desplazamiento de coma tiene en cuenta la escala entre el sistema $BSS(n)$ y el punto fijo $BSS(n,m)$. Por lo que la relación termina resultando 2^{-m}.

$$\textbf{cadena} \longrightarrow^{I_bss} \textbf{valor entero} \longrightarrow^{escala} \textbf{valor fraccionario}$$

Aplicando este mecanismo al ejemplo anterior, la cadena `10011101` se interpreta como:

Paso 1: interpretar en `BSS`

$$I_{BSS(8)}(10011101) = 2^7 + 2^4 + 2^3 + 2^2 + 2^0 =$$
$$= 128 + 16 + 8 + 4 + 1 = 157$$

Paso 2: escalar el valor entero

$$157 \cdot 2^{-3} = 157 \cdot \frac{1}{2^3} = \frac{157}{2^3} = \frac{157}{8} = 19,625$$

Finalmente:

$$I_{BSS(8,3)}(10011101) = 19,625$$

Notar que: 2^{-3} *es la escala del sistema con respecto a BSS, dado que se utilizan 3 bits como fraccionarios.*

Pasando en limpio:

$$\texttt{10011101} \longrightarrow^{I_{bss}} \textbf{157} \longrightarrow^{\times 2^{-3}} \textbf{valor 19,625}$$

15.1.2. Representación y error

Tal como ocurre con la interpretación, la representación también tiene estos dos posibles mecanismos.

El mecanismo por partes requiere partir el problema y representar por un lado la parte entera (usando la representación de BSS) y por otro lado la parte fraccionaria realizando sucesivas multiplicaciones.

Por ejemplo para representar el valor **3,8** en el sistema BSS(7,4) necesitamos:

1. Representar el **valor 3** sólo usando los bits enteros (3 bits): `011`
2. Construir la cadena fraccionaria que aproxime el valor **0,8** multiplicando por 2 tantas veces como bits fraccionarios se tiene, y en cada multiplicación separar la parte entera del resultado, que es lo que corresponde a cada nuevo bit.
   Volviendo al ejemplo:
   a) 0,8 * 2 = **1**,6 $\Rightarrow b_{-1} = 1$
   b) 0,6 * 2 = **1**,2 $\Rightarrow b_{-2} = 1$
   c) 0,2 * 2 = **0**,4 $\Rightarrow b_{-3} = 0$
   d) **0**,4 * 2 = **0**,8 $\Rightarrow b_{-4} = 0$

   En este caso multiplicamos 4 veces por 2 porque disponemos de 4 bits para la parte fraccionaria.

   Este mecanismo establece un criterio de corte basado en la cantidad de bits disponibles, pero como en el caso del sistema decimal, esto puede obtener una cadena que no aproxima de la mejor manera. Por ejemplo, en el sistema decimal si contamos con sólo 2 dígitos fraccionarios para aproximar el valor 0,009, existe una gran diferencia entre aproximarlo con 0,00 o aproximarlo con 0,01. En el ultimo caso se llevó a cabo un *redondeo* mientras que en el primer caso se truncó la cadena ocasionando más pérdida.

   redondeo

   Entonces, para evitar que la cadena de bits quede truncada se hace un paso más (m+1 pasos en total) a los fines del redondeo, y si el bit obtenido en este último paso es un `1`, se suma el valor 1 a la cadena de bits fraccionaria resultante.
   Siguiendo el ejemplo, el último paso sería:

   0,8 * 2 = **1**,6 $\Rightarrow b_{-5} = 1$

   Como en este último paso obtuvimos un 1, debemos sumar el valor 1 a la cadena fraccionaria original (sin este último bit), quedando la cadena fraccionaria resultante de la siguiente manera:

   $1100 + 1 = 1101$
3. Finalmente ambas subcadenas se componen en la cadena resultante: `011 1101`

El *Mecanismo escalado* requiere llevar el valor a representar a un entero para luego aplicar la representación de BSS (R_{bss}). Dicho entero se obtiene al escalar y redondear el valor original, teniendo en cuenta, como en la interpretación, que

Mecanismo escalado

el valor la escala entre el sistema $BSS(n)$ y $BSS(n,m)$ es 2^{-m}. Es decir que se debe multiplicar por 2^m.

$$\textbf{valor fraccionario} \longrightarrow^{escala} \textbf{valor entero} \longrightarrow^{R_{bss}} \textbf{cadena}$$

Por ejemplo, representar **el valor 3,8** en BSS(7,4)

1. Escalar: $3,8 * 2^4 = 60,8$
2. Redondeo: $60,8 \approx 61$
3. $R_{bss(7)}(61) = 0111101$

$$\textbf{valor 3,8} \longrightarrow^{\times 2^4} \textbf{60,8} \longrightarrow^{redondeo} \textbf{61} \longrightarrow^{R_{bss}} \texttt{0111101}$$

Con ambos mecanismos obtuvimos el mismo resultado, la cadena `0111101`, pero una vez finalizada la representación es oportuno controlar el trabajo realizado aplicando la interpretación (con cualquiera de los mecanismos) sobre la cadena obtenida. En el ejemplo:

$$\begin{aligned} I_{bss(7,4)}(0111101) &= 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} \\ &= 2 + 1 + 0,5 + 0,25 + 0,0625 = 3,8125 \end{aligned}$$

Sin embargo, se quería representar el valor **3,8**. Esto ocurre porque no todos los números del rango son representables en el sistema, por lo tanto puede haber un error de representación llamado *error absoluto* , que es el valor absoluto de la diferencia entre el número que se deseaba representar y el número que efectivamente se logró representar. En este ejemplo, la diferencia es:

$$|3,8 - 3,8125| = 0,0125$$

El error absoluto indica la distancia entre el número x (que se quiere representar) y el número x' (valor representable con mejor aproximación de x). Por lo que se calcula mediante la diferencia entre éstos:

$$EA(x) = |x - x'|$$

Al haber una cantidad limitada de bits en la parte fraccionaria, también se limita la precisión, y por lo tanto, existe un error máximo en la representación de un número real denominado *error absoluto máximo* . Este error puede ser nulo para los números racionales [1] pero siempre existe en los irracionales.

Notar que el error absoluto debe ser menor a la mitad de la resolución, es decir $EA(x) <= R/2$.

Por otro lado, el **error relativo** es una idea o descripción de la importancia del error absoluto en relación al valor "x", considerando que no es igual de *importante* el error si "x" es un valor de gran magnitud que si es de magnitud pequeña (cerca del 0). Se calcula mediante la siguiente proporción:

$$ER(x) = \frac{EA(x)}{x}$$

Por ejemplo, llegar 20 minutos tarde a una clase de 3 horas es más importante que 20 minutos de atraso en la llegada de un colectivo de larga distancia que llega desde Bariloche. Notar que el error relativo es un valor en el rango [0.,1].

[1] Un número x es racional si existen $a \in \mathbb{Z}$ y $b \in \mathbb{Z}$ tales que $x = \frac{a}{b}$. Los racionales tienen una cantidad finita de cifras o una periodicidad en su parte fraccionaria.

15.1.3. Rango y Resolución

Al igual que en los sistemas enteros (BSS, SM y CA2), el rango en Punto Fijo está determinado por el intervalo de números representables. Por ejemplo, el rango del sistema BSS(2,1) está dado por:

Mínimo : $I_{bss(2,1)}(00) = 0$

Máximo : $I_{bss(2,1)}(11) = 2^0 + 2^{-1} = 1 + 0,5 = 1,5$

Y el del sistema sm (4,2) por:

Mínimo :$I_{sm(4,2)}(1111) = -(2^0 + 2^{-1} + 2^{-2}) = -(1 + 0,5 + 0,5) = -1,75$

Máximo :$I_{sm(4,2)}(0111) = 2^0 + 2^{-1} + 2^{-2} = 1 + 0,5 + 0,5 = 1,75$

Sin embargo, el rango se refiere a todos los números representables en el intervalo. En los sistemas enteros esto es trivial (todos los **enteros** del intervalo), pero en punto fijo, para determinar exactamente qué números están representados dentro del intervalo es necesario el concepto de **resolución**: *la resolución es la distancia entre dos números representables consecutivos.*

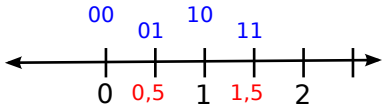


Figura 15.1: BSS(2,1)

Por ejemplo en el sistema $BSS(2,1)$ los valores representables son $\{0; 0{,}5; 1; 1{,}5\}$ (ver figura 15.1) y es posible observar que la distancia entre cada par de valores consecutivos representables es la misma a lo largo de todo el rango y es $0,5$.

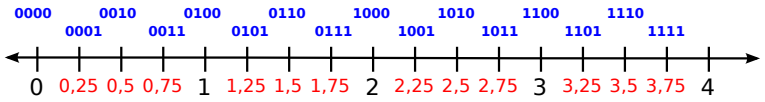


Figura 15.2: BSS(4,2)

Por otra parte, en el sistema $BSS(4,2)$ las cadenas se distribuyen como se muestra en la Figura 15.2. Es posible observar que la distancia entre ellos es $0,25$ y por lo tanto esa es la resolución del sistema.

Por último, es importante notar los sistemas de punto fijo incluyen a los sistemas enteros, es decir que los sistemas de numeración $BSS(n)$, $SM(n)$ y $Ex(n, \Delta)$ *son sistemas de punto fijo con una parte fraccionaria de 0 bits.* Y todos cumplen con tener sus valores a distancia constante, es decir *resolución constante* .

resolución constante

Otros ejemplos

BSS(3,1) Sistema de fijo con 3 bits en total de los cuales 2 son enteros y 1 es fraccionario

Formato

Parte entera	Parte fraccionara
2b	1b

Rango son los valores representables entre:

- $min = I_{bss(3)}(000) * 2^{-1} = 0/2 = 0$
- $max = I_{bss(3)}(111) * 2^{-1} = 7/2 = 3,5$

Resolución $0,5$

SM(3,1) Sistema de punto fijo con 3 bits en total de los cuales 1 es de signo y 2 de magnitud. Los bits de magnitud se reparten entre parte entera (1b) y parte fraccionaria (1b).

Formato

Signo	Magnitud	
	Parte entera	Parte fraccionaria
1b	1b	1b

Rango son los valores representables entre:

- $min = I_{sm(3)}(111) * 2^{-1} = -3/2$
- $max = I_{sm(3)}(011) * 2^{-1} = 3/2$

Resolución $0,5$

15.2. Sistemas de Punto Flotante

Como se mencionó en la sección 15.1.2, los sistemas de punto fijo tienen un error de representación relacionado con la resolución constante del sistema. Entonces el error producido al representar un número de magnitud pequeña es más significativo que el mismo error al representar un valor de magnitud mayor (*Error Relativo*). Por eso es importante relativizar el error, pensándolo como una proporción (o un porcentaje).

Esto indica una limitación en los sistemas de punto fijo, donde al representar valores muy pequeños (cercanos al cero) o valores muy grandes se comete el mismo **error absoluto máximo** , y este no tiene la misma importancia en los extremos del rango como entorno al cero. Como un segundo ejemplo, considerar un sistema de punto fijo cuya resolución sea de 0,5

1. Si se intenta representar el valor *100.000,3* se lo aproxima con el valor *100.000,5* con un error de 0,2.

2. Si se intenta representar el valor *0,26* se lo aproxima con el valor *0,5* con un error de 0,24.

En el primer caso el error obtenido es despreciable, mientras que en el segundo es muy importante.

¿Cómo es posible mejorar el sistema de numeración para que permita ser más precisos en las magnitudes pequeñas y no tan precisos

en las magnitudes grandes, donde el error relativo es "aceptable"?

Con la intención de diseñar un sistema más flexible en cuanto al error de representación, se presentan los sistemas de **punto flotante**. Estos permiten tener una resolución variable, teniendo una resolución pequeña en la zona cercana al cero y una resolución grande en las zonas alejadas del cero. Esto se profundizará más adelante.

punto fijo

Suponer un sistema limitado de bits, en *punto fijo* si queremos tener más precisión en cuanto a los fraccionarios, se debe "sacrificar.[el] rango representable de la parte entera.

mantisa

exponente

La *notación científica* permite escribir de modo abreviado números muy grandes (con muchos dígitos enteros) y números muy cercanos a cero (con muchos dígitos fraccionarios). Las expresiones que utilizan esta notación cuentan con dos partes. Por un lado, la *mantisa* , que representa la parte significativa del número, y el *exponente* , que permite "recordar" dónde estaba la coma antes de abreviar. Ejemplos:

$$62000000000000000000000 = 62 * 10^{20}$$

$$0,00000000000000000000062 = 62 * 10^{-20}$$

Si se aprovecha esto en la representación de números dentro de una computadora, se utiliza la base 2 y los valores se representan mediante dos campos: una **mantisa** y un **exponente**:

$$m \times 2^e$$

Por ejemplo, suponer la cadena `11000000`. Su interpretación en punto fijo BSS(8,0) es:

$$I(11000000) = 2^7 + 2^6 = 64 + 128 = 192$$

Si se busca "comprimirla" debemos correr la coma 6 lugares, es decir remover los 0s de la derecha y escalar una cadena más chica (`11`).

$$I(11000000) = I(11) * 2^6 = (2^2 + 2^1) * 2^6 = 3 * 64 = 192$$

Entonces al almacenar esta información se tiene una mantisa `11` y un exponente `110`, que representa el 6:

$$I_{bss}(11) \times 2^{I_{bss}(110)}$$

La base del exponente no hace falta registrarla porque es siempre 2. Entonces a la hora de almacenarlas, se concatenan ambas subcadenas: `11110`, en este caso primero la mantisa y luego el exponente aunque podría elegirse concatenar en otro orden (`11011`).

Así, con pocos bits de exponente se pueden guardar valores muy grandes. Considerar un sistema **con 2 bits de exponente**, que permite representar (en BSS) hasta el número 3, y por lo tanto se pueden "comprimir" 3 posiciones (o ceros). Con 4 bits se puede representar (en BSS) hasta el número 15, y por lo tanto se pueden "comprimir" 15 posiciones (o ceros).

$$I(1100000000000000) = I(11) * 2^{15}$$

Concluyendo, la mantisa (m) y el exponente (e) **son cadenas binarias que codifican valores en los sistemas de numeración conocidos**. El exponente puede estar representado en un sistema entero como binario sin signo, complemento a dos, exceso o signo y magnitud. La mantisa, además, admite ser representada sistema de punto fijo. Si todos sus bits son fraccionarios, entonces se la considera *mantisa fraccionaria* .

mantisa fraccionaria

Los sistemas de punto flotante no solo nos dejan representar números fraccionarios, sino también números muy grandes utilizando una menor cantidad de bits.

15.2.1. Interpretación

Para interpretar una cadena en un sistema de punto flotante es necesario conocer la especificación del sistema. Suponer por ejemplo un sistema cuya mantisa está codificada en $BSS(4)$ y el exponente en $BSS(3)$, organizado de la siguiente manera.

mantisa BSS(4)	exponente BSS(3)

Es decir que las cadenas del sistema en cuestión tienen 7 bits de longitud, y para interpretar cualquiera de ellas se la **debe segmentar para interpretar por separado mantisa y exponente aplicando las reglas de interpretación de los sistemas correspondientes**. Por ejemplo, interpretar la cadena `1101101` requiere interpretar la cadena `1101` en $BSS(4)$ y la cadena `101` en el sistema $BSS(3)$, para finalmente calcular

$$m \times 2^e$$

Interpretación de la mantisa:

$$m = I_{bss(4)}(1101) = 2^3 + 2^2 + 2^0 = 13$$

Interpretación del exponente:

$$e = I_{bss(3)}(101) = 2^2 + 2^0 = 5$$

Por último se reemplazan los valores obtenidos en la formula

$$m \times 2^e$$

obteniéndose:

$$13 \times 2^5 = 13 \times 32 = 416$$

15.2.2. Rango y resolución

Para analizar el rango de un sistema de manera general, se construye la cadena que representa al número más chico y la que representa al número más grande. Notar que en particular en los sistemas de punto flotante se debe tener en cuenta el exponente, el cual puede tener un subsistema distinto al de la mantisa. De esta manera el valor máximo se construye utilizando la mantisa máxima y el máximo exponente, mientras que el valor mínimo se compone con la mantisa mínima y el exponente máximo.

Para ejemplificar, considerar el siguiente sistema:

e:bss(2)	m:bss(2)

Como el sistema $BSS(2)$ no admite números negativos, la mantisa mínima es: `00`, cuya interpretación es:

$$M_{min} = I_{bss}(00) = 0$$

La mantisa máxima es: `11`, y su interpretación:

$$M_{max} = I_{bss}(11) = 2^1 + 2^0 = 3$$

El exponente máximo es: `11`, y representa:

$$E = I_{bss}(11) = 2^1 + 2^0 = 3$$

Por lo tanto el rango es el siguiente:

$$Rango = [M_{min} \times 2^E; M_{max} \times 2^E] = [0 \times 2^3; 3 \times 2^3] = [0; 24]$$

En la tabla 15.1 se detallan las interpretaciones de todas las cadenas del sistema anterior y al graficar las cadenas con respecto a los valores que representan se obtiene una distribución como la de la figura 15.3.

cadena	exponente	mantisa	$N = M \times 2^E$
`0000`	$I_{bss}(00) = 0$	$I_{bss}(00) = 0$	$N = 0 \times 2^0 = 0$
`0001`	$I_{bss}(00) = 0$	$I_{bss}(01) = 1$	$N = 1 \times 2^0 = 1$
`0010`	$I_{bss}(00) = 0$	$I_{bss}(10) = 2$	$N = 2 \times 2^0 = 2$
`0011`	$I_{bss}(00) = 0$	$I_{bss}(11) = 3$	$N = 3 \times 2^0 = 3$
`0100`	$I_{bss}(01) = 1$	$I_{bss}(00) = 0$	$N = 0 \times 2^1 = 0$
`0101`	$I_{bss}(01) = 1$	$I_{bss}(01) = 1$	$N = 1 \times 2^1 = 2$
`0110`	$I_{bss}(01) = 1$	$I_{bss}(10) = 2$	$N = 2 \times 2^1 = 4$
`0111`	$I_{bss}(01) = 1$	$I_{bss}(11) = 3$	$N = 3 \times 2^1 = 6$
`1000`	$I_{bss}(10) = 2$	$I_{bss}(00) = 0$	$N = 0 \times 2^2 = 0$
`1001`	$I_{bss}(10) = 2$	$I_{bss}(01) = 1$	$N = 1 \times 2^2 = 4$
`1010`	$I_{bss}(10) = 2$	$I_{bss}(10) = 2$	$N = 2 \times 2^2 = 8$
`1011`	$I_{bss}(10) = 2$	$I_{bss}(11) = 3$	$N = 3 \times 2^2 = 12$
`1100`	$I_{bss}(11) = 3$	$I_{bss}(00) = 0$	$N = 0 \times 2^3 = 0$
`1101`	$I_{bss}(11) = 3$	$I_{bss}(01) = 1$	$N = 1 \times 2^3 = 8$
`1110`	$I_{bss}(11) = 3$	$I_{bss}(10) = 2$	$N = 2 \times 2^3 = 16$
`1111`	$I_{bss}(11) = 3$	$I_{bss}(11) = 3$	$N = 3 \times 2^3 = 24$

Tabla 15.1: Interpretación de cadenas con exponente BSS(2) y mantisa BSS(2)

Como es posible apreciar en la figura anterior, muchas cadenas representan al mismo valor y, además, los valores representables no son equidistantes, como sí ocurre en los sistemas enteros y de punto fijo. Esta característica recibe el nombre de **resolución variable** . En el sistema del ejemplo, la resolución varía entre 1 (resolución mínima del sistema) y 8 (resolución máxima).

resolución variable

Como se puede comprobar, el rango y la resolución de cualquier sistema de punto flotante está determinado por los sistemas subyacentes. Por ejemplo,

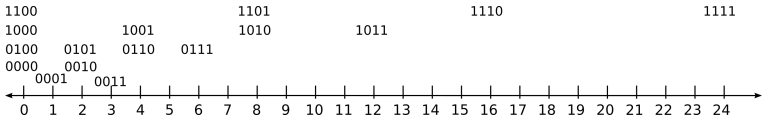


Figura 15.3: Exponente en BSS(2) y Mantisa en BSS(2)

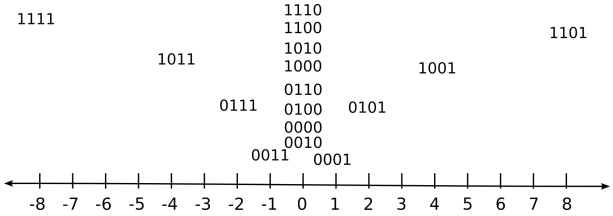


Figura 15.4: Exponente en BSS(2) y Mantisa en SM(2)

considerar el caso donde tanto mantisa como exponente permiten representar un rango sólo con valores positivo (ver figura 15.3) en contraposición a un sistema donde la mantisa permite representar números negativos (ver figura 15.4).

Podría concluirse que el rango del sistema está relacionado con el de la mantisa. En caso que ésta sea simétrica con respecto al cero el sistema de punto flotante también lo es. Por otro lado, es importante analizar cómo el sistema del exponente afecta al rango y la resolución del sistema. Considerar dos sistemas con mantisa BSS donde el primero tiene exponentes positivos (ver figura 15.3) y el otro tiene un sistema en el exponente que permite representar números negativos (ver figura 15.5).

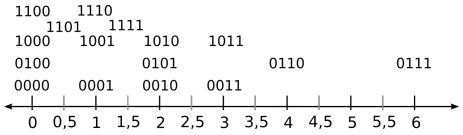


Figura 15.5: Exponente en SM(2) y Mantisa en BSS(2)

En la siguiente tabla se presenta una comparación entre ambos:

Mantisa	Exponente	Mínimo valor representable	Máximo valor representable	Resolución mínima	Resolución máxima
$BSS(2)$	$BSS(2)$	0	24	1	8
$BSS(2)$	$SM(2)$	0	6	0,5	2

Es posible entonces concluir que el exponente, cuando su sistema permite representar números negativos, permite resoluciones menores a los enteros. Por último ver el gráfico de la figura 15.6, que es similar al de la figura 15.4 pero más aglutinado en torno al cero.

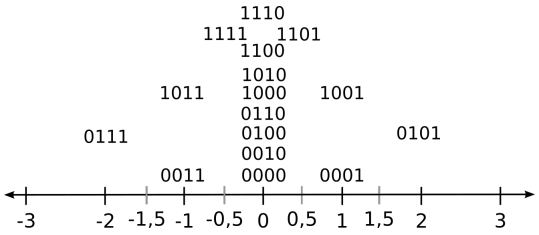


Figura 15.6: Exponente en SM(2) y Mantisa en SM(2)

Ejemplo de cálculo de resoluciones

Suponer un sistema de punto flotante como el siguiente para calcular sus resoluciones máxima y mínima: mantisa en sistema $SM(4)$ y exponente en sistema Exc(4,8). Con el formato:

`m:sm(4)`	`e:exc(4,8)`

a) Para la **resolución mínima** se necesita el mínimo exponente, es decir, `0000` en este sistema. El cual representa al número -8. Las cadenas a utilizar son `0000 0000` y `0001 0000`, las cuales son consecutivas, pues tienen igual exponente. La interpretación de estas cadenas es:

- $m_1 = I_{SM(4)}(0000) = 0$ entonces $n_1 = m_1 \times 2^{-8} = 0 \times 2^{-8} = 0$
- $m_2 = I_{SM(4)}(0001) = 2^0 = 1$ entonces $n_2 = m_2 \times 2^{-8} = 1 \times 2^{-8} = 2^{-8}$

Y al restar los valores se obtiene:

$$|n_1 - n_2| = |0 - 2^{-8}|$$

Por lo tanto, la **resolución mínima** es: 2^{-8}

b) Para la resolución máxima es necesario el máximo exponente, es decir, `1111`, que representa al número 7. Las cadenas a utilizar son `0000 1111` y `0001 1111`, las cuales son consecutivas. La interpretación de estas cadenas es como sigue:

- $M_1 = I_{SM(4)}(0000) = 0$ entonces $N_1 = M_1 \times 2^7 = 0 \times 2^7 = 0$
- $M_2 = I_{SM(4)}(0001) = 2^0 = 1$ entonces $N_2 = M_2 \times 2^7 = 1 \times 2^7 = 2^7$

Y al restar los valores se obtiene:

$$|N_2 - N_1| = |2^7 - 0|$$

Por lo tanto, la **resolución máxima** es: 2^7

Tres (3) opciones diferentes para calcular resoluciones

1. Se basa en la definición de resolución de manera específica, es decir, respetando su algoritmo. Así es que:
   Resolución mínima: la diferencia entre mantisa y exponente mínimo, con la mantisa siguiente consecutiva y exponente mínimo.
   Resolución máxima: la diferencia entre mantisa y exponente máximo, con la mantisa anterior consecutiva y exponente máximo.

2. Siendo que la distancia entre la cadena mínima y su siguiente, así como la máxima y su anterior, siempre será uno, se puede calcular la mantisa como la diferencia entre las cadenas más sencillas `00..00` y `00..01`, con sus respectivos exponentes para la resolución mínima y máxima:
   Resolución mínima: el exponente mínimo y las cadenas mencionadas para la mantisa.
   Resolución máxima: el exponente máximo y las cadenas mencionadas para la mantisa.

3. Generalizando la definición, podemos decir que la diferencia entre una cadena y su siguiente o anterior (`00..00` y `00..01`) dará como ressultado la cadena `00..01`, concluimos que:
   Resolución mínima: se toma la cadena `00..01` para la mantisa con el exponente mínimo.
   Resolución máxima: se toma la cadena `00..01` para la mantisa con el exponente máximo.
   Aclaración: esta opción no es viable para el sistema `Exceso` dado que al utilizar un desplazamiento esta generalización no aplica.

Veamos un ejemplo para cada opción:
Suponer un sistema con mantisa y exponente en el subsistema $SM(2)$, con el siguiente formato:

e:SM(2)	m:sm(2)

Para las opciones 1 y 2, denominamos $C1$ y $C2$ a las cadenas consecutivas. Y $N1$ y $N2$ a sus respectivos valores para el cáluclo de la distancia.

OPCIÓN 1:
Mínima: exponente mínimo con mantisa mínima y exponente mínimo con mantisa siguiente
$C_1 =$ `11 11`, donde $N_1 = -1 \times 2^{-1} = -0{,}5$
$C_2 =$ `11 10`, donde $N_2 = -0 \times 2^{-1} = 0$
Resolución mínima: $|N_2 - N_1| = |-0{,}5 - 0| = 0{,}5$

Máxima: exponente máximo con mantisa máxima y exponente máximo con mantisa anterior
$C_1 =$ `01 01`, donde $N_1 = 1 \times 2^1 = 2$
$C_2 =$ `01 10`, donde $N_2 = 2 \times 2^1 = 4$
Resolución máxima: $|N_2 - N_1| = |2 - 4| = 2$

OPCIÓN 2:
Mínima: exponente mínimo con mantisa `00` y exponente mínimo con mantisa consecutiva `01`
$C_1 =$ `11 00`, donde $N_1 = 0 \times 2^{-1} = 0$
$C_2 =$ `11 01`, donde $N_2 = 1 \times 2^{-1} = 0{,}5$
Resolución mínima: $|N_2 - N_1| = |0 - 0{,}5| = 0{,}5$

Máxima: exponente máximo con mantisa `00` y exponente máximo con mantisa consecutiva `01`
$C_1 =$ `01 00`, donde $N_1 = 0 \times 2^{1} = 0$
$C_2 =$ `01 01`, donde $N_2 = 1 \times 2^{1} = 2$
Resolución máxima: $|N_2 - N_1| = |0 - 2| = 2$

OPCIÓN 3: en este caso no contamos con 2 cadenas para la mantisa, dado que tanto la resolución mínima como la máxima se interpretará con la cadena constante `00..01`, que en este caso, al ser de 2 bits queda `01`.
Mínima: mantisa `01` con exponente mínimo
$C =$ `11 01`, donde:
Resolución mínima: $I_{SM(2)}(01) \times 2^{I_{SM(2)}(11)} = 1 \times 2^{-1} = 0{,}5$

Máxima: mantisa `01` con exponente máximo
$C =$ `01 01`, donde:
Resolución máxima: $I_{SM(2)}(01) \times 2^{I_{SM(2)}(01)} = 1 \times 2^{1} = 2$

Notar que para este sistema cualquiera de las 3 opciones son válidas, dado que se llega al mismo resultado, y en todos los casos se aplica la definición de resolución, ya sea de manera específica o genrealizada.

Ejemplo de cálculo de resoluciones con mantisa fraccionaria

Suponer ahora un sistema de punto flotante como el siguiente para calcular sus resoluciones máxima y mínima: mantisa en sistema $SM(4,3)$ y exponente en sistema Exc(4,8). Con el formato:

`m:sm(4,3)`	`e:exc(4,8)`

Se aplicará la opción 2:

a) Para la **resolución mínima** se necesita el mínimo exponente, es decir, `0000` en este sistema. El cual representa al número -8. Las cadenas a utilizar son `0000 0000` y `0001 0000`, las cuales son consecutivas, pues tienen igual exponente. La interpretación de estas cadenas es como sigue:

- $m_1 = I_{SM(4,3)}(0000) = 0$ entonces $n_1 = m_1 \times 2^{-8} = 0 \times 2^{-8} = 0$
- $m_2 = I_{SM(4,3)}(0001) = 2^{-3}$ entonces $n_2 = m_2 \times 2^{-8} = 2^{-3} \times 2^{-8} = 2^{-11}$

Y al restar los valores se obtiene:

$$|n_2 - n_1| = |2^{-11} - 0|$$

Por lo tanto, la **resolución mínima** es: 2^{-11}

b) Para la resolución máxima es necesario el máximo exponente, es decir, `1111`, que representa al número 7. Las cadenas a utilizar son `0000 1111` y `0001 1111`, las cuales son consecutivas. La interpretación de estas cadenas es como sigue:

- $M_1 = I_{SM(4,3)}(0000) = 0$ entonces $N_1 = M_1 \times 2^7 = 0 \times 2^7 = 0$
- $M_2 = I_{SM(4,3)}(0001) = 2^{-3} = 1$ entonces $N_2 = M_2 \times 2^7 = 2^{-3} \times 2^7 = 2^4$

Y al restar los valores se obtiene:

$$|N_2 - N_1| = |2^4 - 0|$$

Por lo tanto, la **resolución máxima** es: 2^4

Generalización del calculo de resolución para sistemas de punto flotante

Analicemos un poco el ejemplo anterior con el fin de generalizarlo para cualquier sistema.

En el ejemplo utilizamos dos cadenas (las cuales llamaremos C_1 y C_2 por simplicidad), en las cuales la mantisa de C_2 (llamémosle, m_2 = `0001`) es consecutiva de la mantisa de C_1 (llamémosle, m_1 = `0000`). Teniendo eso en cuenta, podemos decir que la mantisa m_2 puede reescribirse tal que: $m_2 = m_1$ + `0001` (utilizando la definición de que dado un número x, su consecutivo es $x + 1$).

Ahora bien, los pasos del ejemplo anterior era:

1. Tener dos cadenas consecutivas C_1 y C_2.
2. Restar las notaciones científicas de cada (es decir, la suma de potencias de 2 que correspondan a la mantisa por 2^e, siendo e el exponente máximo o mínimo)
3. Obtener el resultado final.

Si prestamos atención a los resultados que obtuvimos, veremos que en ambos, la única potencia de 2 que se multiplica finalmente por 2^e es 2^{-4}, la cual es la diferencia entre las mantisas m_2 y m_1, es decir, 1 pues siendo m_1 una cadena cualquiera y $m_2 = m_1$ + `0001`, su diferencia sera siempre 1, representado con la cadena 0..01 (todos ceros, exceptuando el ultimo bit).

Como el algoritmo para calcular la resolución siempre utiliza dos cadenas consecutivas (por definición de resolución), se puede deducir que al restarlas siempre se obtendrá como resultado de la resta de las mantisas esa cadena 0..01, por lo que se puede calcular la resolución mínima y máxima de un sistema con las cuentas:

Resolución mínima = $I_m(0..01) \times 2^e$

siendo $I_m(0..01)$ la interpretación de la cadena 0..01 en el sistema de la mantisa del sistema de punto flotante y e el mínimo exponente posible. Y:

Resolución máxima = $I_m(0..01) \times 2^E$

siendo $I_m(0..01)$ la interpretación de la cadena 0..01 en el sistema de la mantisa del sistema de punto flotante y E el máximo exponente posible.

Nota: a su vez, puede escribirse $I_m(0..01)$ como r, siendo r la resolución propia del sistema usado en la mantisa, pues la cadena (0..01), al interpretarla en sistemas *BSS, SM o CA2* (tanto enteros como de punto fijo), nos dará la resolución del sistema usado para interpretarla.

15.2.3. Normalización

Cualquier número en punto flotante puede expresarse de distintas formas. Sin embargo, no es deseable tener múltiples representaciones para el mismo valor, por dos motivos. En primer lugar, no se quiere subaprovechar las cadenas, y en segundo lugar, no es bueno tener redundancia pues esto implica que la lógica de comparación entre cadenas sea mas compleja.

Por ejemplo, si se considera un sistema con mantisa en SM(8) y exponente en CA2(5), con el formato (s)(e)(m), la interpretación de las siguientes cadenas muestra que representan al mismo valor:

- $I(0000000000100) = 4 \times 2^0 = 2^2$
- $I(0000010000010) = 2 \times 2^1 = 2^2$
- $I(0000100000001) = 1 \times 2^2 = 2^2$

La solución a este problema es la *normalización* , esto es: establecer una regla para seleccionar las cadenas válidas, descartando las demás. En general se establece como regla que las cadenas válidas deben tener una mantisa que comienza con el bit 1, es decir en su bit mas significativo. En el ejemplo anterior la única cadena normalizada para representar el 2^2 es `0 11100 1000000`.

normalización

$$N = I_{sm}(01000000) \times 2^{I_{ca2}(11100)} = 2^6 \times 2^{-4} = 2^2$$

Fácilmente pueden verse algunas desventajas de este criterio de normalización. Por un lado, la mitad de las cadenas se descartan y por otro lado, ese bit que debe asegurarse en valor 1 ocupa un espacio innecesario y podría no escribirse. Esto lleva a una optimización que deja implícito ese primer bit de la mantisa. Es decir que ese bit no se incluye en la cadena pero si se lo considera como un componente de peso a la hora de interpretar.

bit implícito

La optimización del bit implícito permite tener disponible en la mantisa un bit mas, consiguiendo cubrir un rango más amplio si la mantisa es entera, o más precisión, si la mantisa es fraccionaria.

Notación La mantisa fraccionaria se denota con el criterio del sistema de punto fijo. Por ejemplo: SM(10,10). Si además este sistema está normalizado y con un bit implícito esto se denota: SM(10+1, 10)

Ejemplo Suponer un sistema con mantisa SM(10+1, 10) y exponente SM(5) y cuyo formato es (magnitud mantisa)(signo mantisa)(signo exponente)(magnitud exponente). La interpretación de la cadena `010001011 1 0 1110` requiere llevar adelante los siguientes pasos:

1. $I_{sm(10+1,10)}(1010001011)$ (el bit de la izquierda es el signo)
2. Agregar el bit implícito: $I_{sm(11,10)}(11010001011)$ (notar que se trata de la interpretación en el sistema SM(11,10)
3. Interpretar el bit de signo: $-I_{bss(10,10)}(1010001011) = -(2^{-1}+2^{-3}+2^{-7}+2^{-9}+2^{-10})$
4. Intepretar el exponente: $I_{sm(5)}(01110) = 2^1 + 2^2 + 2^3 = 14$
5. Finalmente, el valor del número representado es

$$N = -(2^{-1} + 2^{-3} + 2^{-7} + 2^{-9} + 2^{-10}) \times 2^{14}$$

Ejemplo de cálculo de resoluciones en un sistema normalizado

Calculemos la resolución máxima y mínima de un sistema de punto flotante normalizado con bit implícito como el siguiente: mantisa en el sistema $SM(4+1,4)$ y exponente en el sistema $CA2(4)$, con el formato:

`m:sm(4+1,4)`	`e:ca2(4)`

a) Para la resolución mínima se necesita el mínimo exponente, es este sistema la cadena `1000`, que representa al número -8. Las cadenas a utilizar son `0000 1000` y `0001 1000`, las cuales son consecutivas y como las mantisas están normalizadas y tienen bit implícito, la interpretación es como sigue

   - $m_1 = I_{SM(4+1,4)}(0000) = 2^{-1}$ (pues el bit implícito vale 2^{-1}). Entonces $n_1 = m_1 \times 2^{-8} = 2^{-1} \times 2^{-8} = 0$
   - $m_2 = I_{SM(4+1,4)}(0001) = 2^{-1} + 2^{-4}$, entonces $n_2 = m_2 \times 2^{-8} = (2^{-1} + 2^{-4}) \times 2^{-8}$

   Y al restar los valores se obtiene:

$$|n_2 - n_1| = |(2^{-1} + 2^{-4}) \times 2^{-8} - (2^{-1} \times 2^{-8})| = |(2^{-1} + 2^{-4} - 2^{-1}) \times 2^{-8}| =$$

$$= |2^{-4} \times 2^{-8}|$$

   Por lo tanto, la **resolución mínima** es: 2^{-12}

b) Para la resolución máxima es necesario el máximo exponente, en este sistema la cadena `0111`, que representa al número 7. Las cadenas a utilizar son `0000 0111` y `0001 0111`, las cuales son consecutivas y como las mantisas están normalizadas y tienen bit implícito, la interpretación es como sigue

   - $M_1 = I_{SM(4+1,4)}(0000) = 2^{-1}$ (pues el bit implícito vale 2^{-1}). Entonces $N_1 = M_1 \times 2^7 = 2^{-1} \times 2^7 = 0$

- $M_2 = I_{SM(4+1,4)}(0001) = 2^{-1} + 2^{-4}$, entonces $N_2 = M_2 \times 2^7 = (2^{-1} + 2^{-4}) \times 2^7$

Y al restar los valores se obtiene:

$$|N_2 - N_1| = |(2^{-1}+2^{-4})\times 2^7 - (2^{-1}\times 2^7| = |(2^{-1}+2^{-4}-2^{-1})\times 2^7| = |2^{-4}\times 2^7|$$

Por lo tanto, la **resolución máxima** es:

$$2^{-4} \times 2^7 = 2^3$$

Comparación con un sistema no normalizado

Mantisa	exp	Resolución mínima	Resolución máxima
$BSS(2,2)$	CA2(2) minE =-2 , maxE=1	$I(0010) - I(0110)$ $= \|0 \times 2^{-2} - 0,25 \times 2^{-2}\|$ $= 0,25 \times 2^{-2}$	$I(1001) - I(1101)$ $= \|0,5 \times 2^1 - 0,75 \times 2^1\|$ $= 0,25 \times 2^1$
$BSS(2+1,3)$	CA2(2) minE =-2 , maxE=1	$I(0010) - I(0110)$ $= \|0,5\times 2^{-2} - (0,5+0,125)\times 2^{-2}\|$ $= 0,125 \times 2^{-2}$	$I(1001) - I(1101)$ $= \|(0,5 + 0,25) \times 2^1 - (0,5+0,25+0,125)\times 2^1\|$ $= 0,125 \times 2^1$

15.2.4. Estándar IEEE

El estándar IEEE 754 define representaciones para números de coma flotante con diferentes tipos de precisión: simple y doble, utilizando anchos de palabra de 32 y 64 bits respectivamente. Estas representaciones son las que utilizan los procesadores de la familia x86, entre otros.
Estos sistemas, a diferencia de los anteriores, permiten representar también valores especiales, los cuales serán tratados posteriormente.

Precisión simple

En la representación de 32 bits, el exponente se representa en exceso de 8 bits, con un desplazamiento de 127, y la mantisa está representada en un sistema SM(24+1,23), es decir que:

- Se tienen 24 bits explícitos y uno implícito
- 23 bits son fraccionarios

Como es un sistema signo-magnitud, se tiene 1 bit de signo y 24 bits de magnitud. De los bits de la magnitud, 1 está implícito y los otros 23 son los que se usan explícitamente. De aquí que estos 23 bits son fraccionarios y el bit implícito es entero.

Además, el total de 32 bits se escriben con el siguiente formato:

S	Exponente: 8b	Magnitud: 23b

Precisión doble

De manera similar, en la representación IEEE de doble precisión, el bit mas significativo es utilizado para almacenar el signo de la mantisa, los siguientes 11 bits representan el exponente y los restantes 52 bits representan la mantisa. El exponente se representa en exceso de 11 bits, con un desplazamiento de 1023.

S	Exponente: 11b	Magnitud: 52b

Como en el caso de precisión simple, también se tiene una mantisa normalizada con un bit entero y los restantes fraccionarios, es decir que tiene la forma "1,X", donde X es el valor de los bits fraccionarios. Además, como se tiene un bit implícito, el dígito 1 (entero) está oculto y por lo tanto no es almacenado en la representación, permitiendo así ganar precisión.

Sin embargo, los parámetros usados en las representaciones de simple y doble precisión son los que se describen en la siguiente tabla:

	P. simple	P. doble
Cant. total de bits	32	64
Cant. de bits de la mantisa (*)	24	53
Cant. de bits del exponente	8	11
Mínimo exponente (emin) (**)	-126	-1022
Máximo exponente (emax) (**)	127	1023

(* incluyendo el bit implícito)

(** emin es -126 en lugar de -127, que corresponde al mínimo valor del exceso(8,127), ver siguiente sección)

Nota:

Representación de valores especiales

Una cuestión de interés para los sistemas de numeración usados en las computadoras, es analizar qué sucede cuando una operación arroja como resultado un número indeterminado o un complejo. En estos casos el resultado constituye un valor especial para el sistema y se almacena como NaN (Not a Number) tal como ocurre al hacer, por ejemplo $\frac{\infty}{\infty}$ ó $\sqrt{-4}$.

A veces sucede que el resultado de una operación es muy pequeño y menor que el mínimo valor representable, en este caso se almacenará como +0 ó -0, dependiendo del signo del resultado. También se observa que al existir un 1 implícito en la mantisa no se puede representar el valor cero como un número normal, por lo que éste es considerado un valor especial.

Por otro lado, ante una operación que arroje un resultado excesivamente grande (en valor absoluto), este se almacenará como $+\infty$ ó $-\infty$.

De las situaciones mencionadas, surge la necesidad de una representación para los valores especiales.

El exponente lo dice todo

Es importante detenerse en la representación del exponente, que como se ha visto, utiliza el sistema Exceso con frontera no equilibrada (127 o 1023), lo que permite almacenar exponentes comprendidos en el rango [-127,128] en el sistema de precisión simple o [-1023,1024] en el sistema de precisión doble. Pues, puede verse en la tabla de la sección anterior que el rango entre emin y emax no cubre todo el rango disponible, y esto se debe a que se reservan las representaciones de emin-1 y emax+1 en ambas precisiones para representar valores especiales. Nótese que esta elección no es arbitraria: la cadena que representa emin-1 está compuesta de ceros y la cadena que representa el valor emax+1 está compuesta por unos, ambos fácilmente reconocibles.

Adicionalmente pueden representarse valores subnormales o desnormalizados, es decir números **no normalizados**, de la forma $\pm 0, X * 2^{\delta}$, que se extienden en el rango comprendido entre el mayor número normal negativo y el menor número normal positivo. Dicho exponente especial δ tiene el valor -126.

Nota: estos números desnormalizados no tienen bit implícito (ó es cero).

De esta manera, se definieron las siguientes clases de representaciones:

Números normalizados Las cadenas de esta clase se distinguen con un exponente que no sea nulo (cadena compuesta por 0s) ni saturado (cadena compuesta por 1s). Gráficamente, la clase de números normalizados se distribuye como sigue (LP=Límite positivo/ LN=Límite negativo):

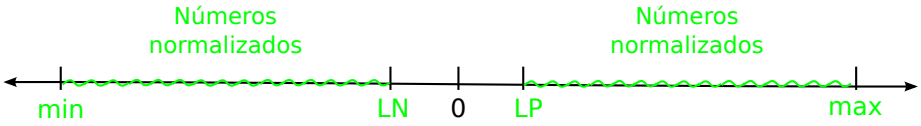


Figura 15.7: Normalizados del estándar IEEE 754

Números denormalizados Las cadenas de esta clase tienen exponente nulo pero mantisa no nula. Gráficamente, la clase de números denormalizados se distribuye como sigue:

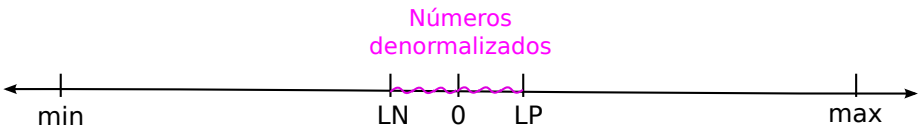


Figura 15.8: Denormalizados del estándar IEEE 754

Ceros Esta clase incluye sólo dos cadenas: aquella compuesta con exponente y mantisa nulos, con ambos signos posibles. Esto permite representar el valor 0 positivo o negativo. Es importante notar que ninguna de las clases anteriores (normalizados o desnormalizados) permite construir por si misma el valor 0.

Infinitos Esta clase se identifica con exponente saturado (`1..1`) y mantisa nula. Permite representar la situación en que el resultado está fuera del rango representable por los normalizados

Not a number (NaN) Esta clase se identifica con exponente saturado y es utilizada para representar los casos de error descriptos antes

La siguiente tabla resume cómo se distinguen las cadenas de las diferentes clases.

Exponente	Mantisa	Clase de número
0..0	0..0	± 0
0..0	$\neq$ 0.,0	Denormalizados: $\pm 0, X * 2^{emin}$
1..1	0..0	$\pm\infty$
1..1	$\neq$ 1.,1	NaN
[emin,emax]	cualquiera	Normalizados: $\pm 1, X * 2^{e}$

Ejemplos de interpretación

Cadena normalizada Por ejemplo, se quiere interpretar la cadena en formato de precisión simple:

`1100 0010 0110 1011 1000 0000 0000 0000`

Para esto, es necesario separar los diferentes campos de la cadena:

1	10000100	110 1011 1000 0000 0000 0000
S	exponente:8b	Mantisa: 23b t

Dado que el exponente no es la cadena `00000000` ni la cadena `11111111`, se entiende que se lo debe interpretar como **un número en la clase normalizada**, por lo que se debe interpretar separadamente:

- Exponente: interpretar en Exc(8,127)

$$e = I_{ex}(10000100) = I_{bss}(10000100) - 127 = 2^7 + 2^2 - 127 = 5$$

- Mantisa: Dado que hay un bit implícito cuyo peso es $2^0 = 1$.

$$m = -(1 + I_{bss(23,23)}(11010111000000000000000)) =$$

$$= -(1 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-7} + 2^{-8})$$

Cadena desnormalizada El siguiente es un ejemplo de una cadena (en formato de precisión simple) cuyo exponente indica que es un número desnormalizado:

`0000 0000 0010 0000 1011 1000 0000 0000`

Separando los campos se obtiene:

0	00000000	010 0000 1011 1000 0000 0000
S	Exp:Exc(8,127)	Mant: BSS(23,23)

- Exponente: En este caso no se interpreta el exponente, sino que se usa el exponente especial

$$e = -126$$

- Mantisa: Dado que **no hay bit implícito**

$$m = I_{bss(23,23)}(01000001011100000000000) =$$

$$= 2^{-2} + 2^{-8} + 2^{-10} + 2^{-11} + 2^{-12}$$

Apéndice A

Validación de programas

Prueba de escritorio

Como se presentó en el apartado **??**, la prueba de escritorio es una herramienta de validación del funcionamiento de los programas. Para hacerlo se considera el punto de vista de la unidad de control y se sigue la ejecución secuencial de las instrucciones una a una, tomando en cuenta todas las modificaciones que sufren los registros y la memoria.

En esta oportunidad, se considerará una única rutina para ser evaluada en diferentes escenarios. Considerar el siguiente ejemplo de especificación de la rutina: "Se necesita un programa que considere un valor en el sistema $BSS(16)$ en R5 para determinar si dicho valor es par o impar. Para esto se espera que en R5 tenga el valor `0001` si es impar y si es par que el valor sea `0000` en el mismo registro."

Se escribe el siguiente programa en la arquitectura Q para resolver lo indicado:

```
MOV R4, R5
MUL R4, 0x0002
DIV R4, 0x0002
SUB R5, R4
```

El programa anterior no cumple con la especificación pedida y para demostrarlo se realizarán varias pruebas de escritorio considerando los siguientes escenarios:

- Un caso con un número par, donde el resultado esperado es `R5=0000`
- Un caso con un número impar, donde el resultado esperado es `R5=0001`
- Un caso de borde: un numero par que provoque un desborde del sistema de numeración.

En el primer escenario se supone que en R5 esta el valor `0850`, y además que PC tiene la dirección de la primer celda de memoria de la primera instrucción.

1. El valor almacenado en R5 se copia al registro R4, ahora los dos registro tienen el valor `0850`

2. El valor almacenado en R4 se multiplica por `0002` y se almacena en R4, además el valor `0000` se almacena en R7. R4 ahora almacena `10A0`
3. El valor almacenado en R4 se divide por `0002` y se almacena en R4. R4 ahora almacena `0850`
4. Al valor almacenado en R5 (`0850`) se le resta el valor almacenado en R4 (`0850`). R5 ahora almacena `0000`
5. Finaliza la ejecución

Por lo tanto, dado que `R5=0000` es posible concluir que el programa esta funcionando de acuerdo a lo esperado para `0850`, es decir que `0000` es el resultado que se esperaba.

En el segundo escenario se probará el mismo programa con un valor impar, el valor `53BD`.

1. El valor almacenado en R5 se copia al registro R4, ahora los dos registro tienen el valor `53BD`
2. El valor almacenado en R4 se multiplica por `0002` y se almacena en R4, además el valor `0000` se almacena en R7. R4 ahora almacena `A77A`
3. El valor almacenado en R4 se divide por `0002` y se almacena en R4. R4 ahora almacena `53BD`
4. Al valor almacenado en R5 (`53BD`) se le resta el valor almacenado en R4 (`53BD`). R5 ahora almacena `0000`
5. Finaliza la ejecución

Al analizar el resultado obtenido se puede ver que el programa no funciona como se quería, pues se esperaba `R5=0001` y se obtuvo `R5=0000`, y se deberá analizar los posibles errores para corregirla.

Por último, en el tercer escenario se probará con el `8000`, que al ser multiplicado por 2 provoca un desborde del rango del sistema $BSS(16)$.

1. El valor almacenado en R5 se copia al registro R4, ahora los dos registro tienen la cadena `8000`
2. El contenido de R4 se multiplica por `0002` y el resultado se almacena en R4 (la parte menos significativa) y R7 (la parte mas significativa). Es decir que la cadena `0001` se almacena en R7 y la cadena `0000` se almacena en R4.
3. El contenido de R4 se divide por `0002` y se almacena en R4. R4 ahora tiene la cadena `0000`
4. Al valor almacenado en R5 (`8000`) se le resta el valor almacenado en R4 (`0000`). R5 ahora almacena `8000`
5. Finaliza la ejecución

Al analizar el resultado es posible ver que lo obtenido no corresponde con ninguno de los valores esperados en R5 ($\texttt{0000}_{16}$ o $\texttt{0001}_{16}$). Entonces es posible concluir que por cómo esta construido este programa no funciona con valores mayores a $\texttt{8000}_{16}$ (32768). Para abordar esta situación se podría aclarar esta restricción o se podría rediseñar el código para considerar este nuevo caso.

Como se puede ver en el ejemplo anterior la prueba de escritorio en este caso permitió encontrar dos tipos de errores. Por un lado una limitación del lenguaje (cuando se supera $\texttt{8000}_{16}$ se tiene un desborde), y por otro lado un error de programación, relacionado con el orden de las operaciones `MUL` y `DIV`.

Apéndice B

Especificación de la arquitectura Q

La arquitectura Q tiene las siguientes características generales:

- Memoria de celdas con direccionamiento de 16 bits, con un total de 65536 celdas.
- ALU que soporta los sistemas de numeración BSS y CA2.
- 8 registros de uso general de 16 bits: R0..R7.
- Program Counter (PC) de 16 bits.
- Stack Pointer (SP) de 16 bits. Comienza en la dirección $\texttt{FFEF}_{16}$.
- Flags: Z, N, C, V (Zero, Negative, Carry, oVerflow). Instrucciones que alteran Z y N: ADD, SUB, CMP, DIV, MUL, AND, OR, NOT. Las 3 primeras además calculan C y V

B.1. Instrucciones de 2 operandos

El conjunto de instrucciones de este tipo es el que se detalla en la siguiente tabla. Es importante notar que la instrucción `MOV` copia el valor al operando destino, a diferencia de lo que puede dar a entender su nombre *(mover)*.

Por otro lado, dado que el sistema de numeración que soporta Q de forma nativa es el $BSS(\,)$ entonces se toma la instrucción `DIV` como una división entera.

Operación	Cod Op	Efecto
MUL	0000	{R7,Dest} ← Dest * Origen
MOV	0001	Dest ← Origen
ADD	0010	Dest ← Dest + Origen
SUB	0011	Dest ← Dest - Origen
AND	0100	Dest ← Dest ∧ Origen
OR	0101	Dest ← Dest ∨ Origen
CMP	0110	Modifica los **Flags** según el resultado de Dest - Origen
DIV	0111	Dest ← Dest % Origen

El formato de instrucción de las instrucciones de 2 operandos es como sigue:

Cod_Op (4)	Modo Destino(6)	Modo Origen(6)	Destino(16)	Origen(16)

B.2. Instrucciones de 1 operando Origen

El conjunto de instrucciones de este tipo es el que se detalla en la siguiente tabla.

Operación	Cod Op	Efecto
JMP	1010	PC ← Origen
CALL	1011	[SP] ← PC; SP ← SP - 1; PC ← Origen

El efecto de la instrucción `CALL` se describe como una secuencia de microcódigo. A continuación algunos ejemplos de uso de estas instrucciones:

- `JMP unaEtiquetaInterna`
- `CALL unaEtiquetaOtraRutina`

El formato de instrucción de este tipo de instrucciones no hace referencia al destino, por lo que utiliza un relleno que se indica a continuación.

Cod_Op(4)	Relleno (000000)	Modo Origen(6)	Origen(16)

B.3. Instrucciones de 1 operando Destino

El conjunto de instrucciones de este tipo es el que se detalla en la siguiente tabla.

Operación	Cod Op	Efecto
NOT	1001	Dest ← NOT Dest (bit a bit)

El formato de instrucción de este tipo de instrucciones se indica a continuación.

Cod_Op (4)	Modo Destino (6)	Relleno (000000)	Destino(16)

B.4. Instrucciones sin operandos

El conjunto de instrucciones de este tipo es el que se detalla en la siguiente tabla.

Operación	CodOp	Bits no utilizados	Efecto
RET	1100	0000 0000 0000	SP ← SP + 1; PC ← [SP];

El formato de instrucción de este tipo de instrucciones se indica a continuación.

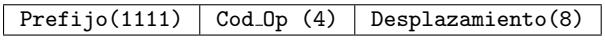
Cod Op (4)	Relleno (12)

B.5. Saltos condicionales

El conjunto de instrucciones de este tipo es el que se detalla en la siguiente tabla.

Operación	Cod Op	Descripción	Condición de Salto
JE	0001	Igual / Cero	Z
JNE	1001	No igual	not Z
JLE	0010	Menor o igual	Z or (N xor V)
JG	1010	Mayor	not (Z or (N xor V))
JL	0011	Menor	N xor V
JGE	1011	Mayor o igual	not (N xor V)
JLEU	0100	Menor o igual sin signo	C or Z
JGU	1100	Mayor sin signo	not (C or Z)
JCS	0101	Carry / Menor sin signo	C
JNEG	0110	Negativo	N
JVS	0111	Overflow	V

El formato de instrucción de este tipo de instrucciones se indica a continuación. Los primeros cuatro bits del código máquina se rellenan con la cadena 1111_2, que se considera como un prefijo. Esto permite tener un código de operación extensible.



B.6. Modos de direccionamiento

Los modos de direccionamiento, cuando aplica, se describen utilizando 6 bits. En los primeros 3 modos de direccionamiento de la siguiente tabla, **H** hace referencia a un posible dígitos hexadecimal. En los ultimos dos modos, **x** referencia a uno de los posibles registros de Q {R0..R7}.

Modo	Codificación	Sintaxis
Inmediato	`000000`	`0xHHHH`
Directo	`001000`	`[0xHHHH]`
Indirecto por memoria	`011000`	`[[0xHHHH]]`
Registro	`100rrr`	`Rx`
Indirecto Registro	`110rrr`	`[Rx]`

Es importante notar que las instrucciones que tienen en Modo Destino operandos del tipo **inmediato** son consideradas como inválidas por la CPU.