
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 0.37
2023

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

Chiara Forti Dono
Denise Pari
Duglas Español
Federico Martínez
Flavia Saldaña
Franco Garcino
Ignacio Mendoza
Ignacio Ferro
Kevin Stanley
Lucía Canosa
Mara Dalponte
Pablo Pissi
Pablo Nieloud
Susana Rosito
Tatiana Molinari
Warmi Guercio

Colaboraron

Axel Lopez Garabal
David Gonzalez
Esteban Dimitroff
Flavia Fernandez
Federico Salguero
Hector Maidana
Martín Enriquez
Maximiliano Díaz
Nahuel Iglesias
Ian Ghioni
Martín Gonzalez Buitrón

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

Índice general

1. Presentación	1
2. Sistemas de cómputos	6
2.1. Evolución de las computadoras	6
2.1.1. Prehistoria	6
2.1.2. Primera generación de computadoras: Tubos de vacío . .	9
2.1.3. Segunda generación de computadoras: Transistores	12
2.1.4. Tercera generación: Circuitos integrados	13
2.1.5. Cuarta generación: Microelectrónica	15
2.2. Arquitectura de Von Neumann	16
3. Sistemas de numeración	19
3.1. Sistema Binario	19
3.1.1. Rango	23
3.1.2. Operaciones Aritméticas	23
3.1.3. Desplazamiento de cadenas	26
3.2. Sistema Hexadecimal	27
3.2.1. Interpretación	28
3.2.2. Representación	29
3.2.3. Rango	29
3.2.4. Operaciones aritméticas	29
3.2.5. Agrupación de bits	30
4. Representación de instrucciones. Q1	32
4.1. Ciclo de vida de los programas	33
4.2. Ejecución de un programa	34
4.3. Arquitectura Q1	35
4.3.1. Ensamblar instrucciones en Q1	37
4.3.2. Ejecución de programas Q1	40
4.3.3. Prueba de programas: prueba de escritorio	41
5. Lógica Digital	43
5.1. Diseño de circuitos	43
5.1.1. Descripción de la interfaz del circuito	44
5.1.2. Formalización de los casos: Tabla de verdad	45
5.1.3. Fórmula de verdad	45
5.2. Mapa del circuito	47
5.2.1. Compuertas lógicas elementales	47

5.2.2. Compuertas lógicas adicionales	48
5.2.3. Conexión de compuertas	49
5.3. Composición de circuitos	50
5.4. Circuitos estándares	51
5.4.1. Circuitos aritméticos	54
5.5. Anexos	59
5.5.1. Lemas	60
6. Memoria y Buses. Q2	62
6.1. Memoria Principal e instrucciones	62
6.1.1. Operaciones sobre la memoria	63
6.1.2. Interconexión	63
6.1.3. Modos de direccionamiento	64
6.2. Arquitectura Q2: alto nivel	64
6.3. Arquitectura Q2: bajo nivel	65
6.3.1. Funcionamiento de la memoria principal	67
6.3.2. Accesos a memoria durante el ciclo de ejecución	69
7. Modularización y Reuso: Rutinas en Q3	72
7.1. Modularización y reuso	72
7.1.1. Haciendo rutinas flexibles: pasaje de parámetros	73
7.1.2. Documentación de las rutinas	74
7.2. Cómo funcionan CALL y RET	75
7.2.1. La estructura de pila	77
7.3. Simulación revisada de la ejecución	78
8. Sistemas de numeración para números enteros	82
8.1. Signo - Magnitud	82
8.2. Complemento a 2	85
8.3. Exceso	89
9. Estructura condicional. Q4	93
9.1. Introducción y motivación	93
9.2. Funcionamiento de los saltos	95
9.2.1. Cómo se implementa una comparación	97
9.3. Verificar las rutinas	100
9.3.1. Diseño de las pruebas	101
9.3.2. Ejecución de las pruebas	102
9.3.3. Otro ejemplo	102
10. Estructura de Repetición	104
10.1. Estructura general	105
11. Máscaras. Q5	106
11.1. Conjunción	107
11.2. Disyunción	108
11.3. Rutinas con máscaras	109

12.Arreglos y recorridos sobre arreglos	111
12.1. Estructura de arreglos	111
12.2. Modos de direccionamiento indirectos	113
12.3. Recorrido de arreglos	116
13.Subsistema de memoria	118
13.1. Memorias de sólo lectura: ROM	119
13.2. Jerarquía de memorias	119
13.3. Memoria Secundaria	120
13.3.1. Discos Magnéticos	120
13.3.2. Discos de estado sólido	121
13.3.3. Redundant Array of Inexpensive Drives	121
14.Memoria Caché	123
14.1. Función de correspondencia	125
14.1.1. Correspondencia asociativa	125
14.1.2. Correspondencia Directa	128
14.1.3. Correspondencia asociativa por conjuntos	130
14.1.4. Fallos y reemplazos	131
14.1.5. Desempeño (performance) de la caché	133
15.Sistemas de numeración fraccionarios	135
15.1. Sistemas de Punto Fijo	135
15.1.1. Interpretación	136
15.1.2. Representación y error	137
15.1.3. Rango y Resolución	139
15.2. Sistemas de Punto Flotante	140
15.2.1. Interpretación	142
15.2.2. Rango y resolución	142
15.2.3. Normalización	144
15.2.4. Estándar IEEE	148
A. Validación de programas	152
B. Especificación de la arquitectura Q	155
B.1. Instrucciones de 2 operandos	155
B.2. Instrucciones de 1 operando Origen	156
B.3. Instrucciones de 1 operando Destino	156
B.4. Instrucciones sin operandos	156
B.5. Saltos condicionales	157
B.6. Modos de direccionamiento	157

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.

2.1.1. 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 complicada máquina de sumar y restar, la cual 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* 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 *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 siglos más tarde para estandarizar la simbología utilizada para procesar la información en las computadoras modernas.

1750 : Tarjetas perforadas

Se usan las tarjetas perforadas para especificar patrones de tejido que luego son interpretadas manualmente por los tejedores.

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. 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 e 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, se convirtió en el paradigma de la primera máquina computacional, desarrollada por Charles Babbage.



Figura 2.2: Máquina de Jackard



Figura 2.3: Cartas perforadas de Jackard

1822: Babbage

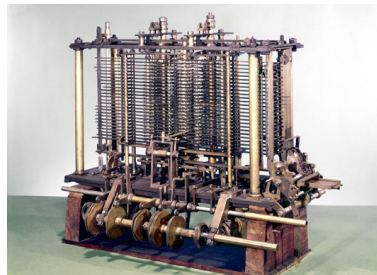


Figura 2.4: Motor de Babbage

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 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 fue terminada. Dos cosas fueron mal. Una era que la fricción y engranajes internos disponibles no eran lo bastante buenos para que los modelos fueran terminados, siendo también las vibraciones un problema constante. La otra fue que Babbage cambiaba incesantemente el diseño de la máquina. En 1833 se habían gastado 17 000 £ sin resultado satisfactorio. 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. La máquina funcionó sin problemas

Entre 1833 y 1842, Babbage lo intentó de nuevo; esta vez, 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. Se considera que la máquina analítica de Babbage fue la primera computadora del mundo. Un diseño inicial plenamente funcional de ella fue terminado en 1835. Sin embargo, debido a problemas similares a los de la máquina diferencial, la máquina analítica nunca fue terminada por Charles. Lady Ada Lovelace, matemática e hija de Lord Byron, se enteró de los esfuerzos de Babbage y se interesó en su máquina. Promovió activamente la máquina analítica, y escribió varios programas. Los diferentes historiadores concuerdan que esas instrucciones hacen de Ada Lovelace la primera programadora de computadoras en el 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).

2.1.2. 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 tableros 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 m². 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.

1952 : IAS (Princeton)

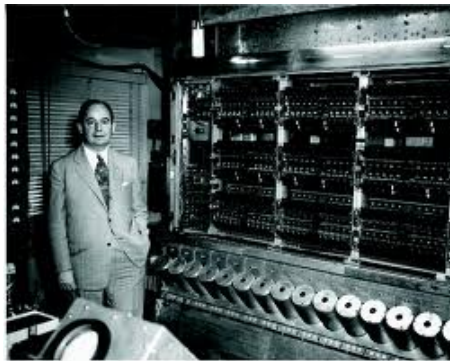


Figura 2.5: John Von Neumann

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.

2.1.3. 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 más 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



Figura 2.6: UNIVAC 1

por mayor capacidad de procesamiento, mayor capacidad de memoria y menor tamaño que la anterior.

1952 IBM 701



Figura 2.7: 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.

2.1.4. 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 2.8: IBM 360

1964 PDP-8



Figura 2.9: PDP8

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.

2.1.5. 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 2.10: Procesador 8080



Figura 2.11: Apple I

2.2. Arquitectura de Von Neumann

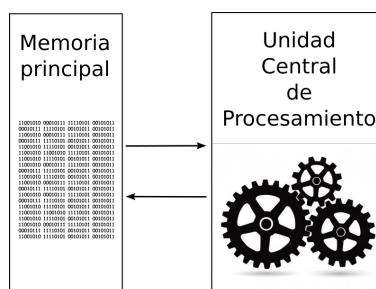


Figura 2.12: Arquitectura conceptual de John Von Neumann

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. Un *algoritmo* es exactamente una secuencia de instrucciones/pasos/ordenes que pertenecen a un conjunto de posibles instrucciones que

conforman un lenguaje. Si ese lenguaje está pensado para un sistema de cómputos que lo *comprende*, entonces los algoritmos son ejecutables automáticamente, es decir que se los define como *programas*. Entonces, el sistema de cómputos (o computadora) resuelve los problemas a través de la **ejecución de programas**.

Como se desarrolló en la sección 2.1, estos sistemas de computos 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 además la complejidad en la operación de sus antecesores, propone separar el soporte físico que traduce las ordenes en acciones mecánicas (o eléctricas) de las órdenes (o instrucciones) en si. La que hoy se conoce como *Arquitectura de Von Neumann* (ver figura 2.12) tiene como componentes principales: por un lado un espacio de almacenamiento de las instrucciones y los datos que se denomina *memoria principal*; y por otro lado un dispositivo que se alimenta de esas instrucciones en un orden determinado y las ejecuta individualmente, denominado *Unidad Central de Procesamiento* o CPU (por sus siglas en inglés *Central Processing Unit*).

Esta propuesta marca un momento de inflexión histórico muy importante que separa lo anterior, donde las computadoras eran diseñadas y operadas por las mismas personas que formaban parte de un grupo muy selecto de la ciencia, de lo posterior, donde se comenzó a identificar varias disciplinas: quienes diseñaban computadoras de propósito general (cada vez mas eficientes, mas pequeñas, mas económicas) de quienes manejan el lenguaje de cada modelo y diseñan los programas que resuelven cada problema. En resumen, tiempo después provoca el surgimiento de la programación y la industria del *software* como una actividad independiente del diseño de computadoras (o *hardware*).

Existen dos Unidades fundamentales que conforman a la CPU, la **ALU** (del inglés *Arithmetic Logic Unit* ó unidad aritmético-lógica) y la **UC** (Unidad de control). La ALU es un circuito lógico digital encargada de realizar las operaciones aritmeticas (suma, resta, multiplicación y división) y las operaciones lógicas como la disyunción, la negación y la conjunción entre dos valores, o mejor dicho cadenas, binarias.

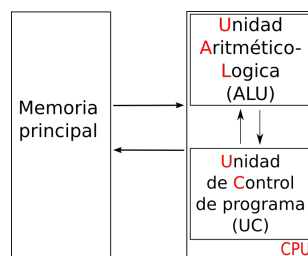


Figura 2.13: Diagrama de la arquitectura de Von Neumann detallando ALU y UC

La UC es el circuito digital principal, responsable de la ejecución de los programas, para lo cual lleva adelante un ciclo de ejecución de instrucciones, que será detallado en el apartado 4.2.

Capítulo 3

Sistemas de numeración

Un sistema de numeración es un conjunto de símbolos y reglas que permiten manipular cadenas de símbolos. Estos sistemas pueden clasificarse en:

- Sistemas no posicionales
- Sistemas posicionales

En esta asignatura se analizarán principalmente los sistemas de numeración posicionales y en particular los de base binaria. Los sistemas posicionales se caracterizan por la importancia de la base del sistema, la que establece la cantidad de símbolos de los que se dispone. Algunos ejemplos de este tipo de sistemas son el sistema decimal, sistema octal, sistema binario y sistema hexadecimal.

3.1. Sistema Binario

Como sistemas de numeración hasta ahora conocemos y manejamos el sistema decimal. Por ejemplo sabemos que **3548** representa el *tres mil quinientos cuarenta y ocho*, pero ¿qué representa ese lenguaje? Razonándolo, esto mismo se puede decir *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 peso'. Dicha potencia de 10 recibe el nombre de *peso del dígito*. De esta manera, 3548 es equivalente a:

$$3 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$$

es decir:

$$3 \times 1000 + 5 \times 100 + 4 \times 10 + 8 \times 1$$

¿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 nos 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).

peso del
dígito

Base:
cantidad
de símbolos
que posee el
sistema

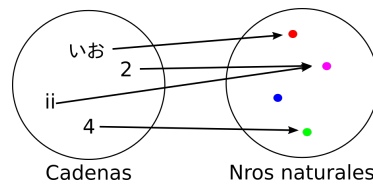


Figura 3.1: Función que relaciona cadenas con los números naturales

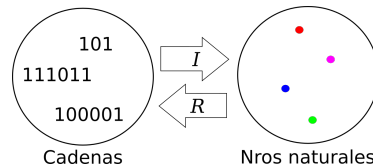


Figura 3.2: Relación dual entre los mecanismos de interpretación y representación

Si se supone el caso donde solo se dispone de un par de símbolos para definir un sistema de numeración, es decir un sistema binario, la base a considerar es 2 y 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 funcionalidades. En este caso podremos:

- Interpretar cadenas binarias
- Representar valores naturales
- Calcular su rango
- Realizar operaciones aritméticas

Existe una dualidad entre la interpretación y la representación. Suponer una cadena c que al ser interpretada se la asocia al valor x , entonces la representación de ese mismo valor debe obtener 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 3.2.

$$I(c) = x \leftrightarrow R(x) = c$$

Interpretación de cadenas

La interpretación es el proceso para determinar qué número (o cantidad) representa una cadena. 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. Pero además esto es coherente con la aplicación de los pesos como se

cadena	valor
00	0
01	1
10	2
11	3

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

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

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

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

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

Finalmente la cadena que representa al valor 26 es $c=11010$, es decir que $R(26) = 11010$. Este proceso se aprecia gráficamente en la Figura 3.3.

Con el objetivo de comprobar la validez del proceso realizado, y teniendo en cuenta la relación dual presentada en la figura 3.2 se interpreta la cadena resultante para compararla con el valor original de x :

$$I(11010) = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26$$

Y con esto se concluye que el proceso fue realizado correctamente.

3.1.1. 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 representable o *rango*. Si bien se dijo que 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*.

Considerar por ejemplo un sistema binario restringido a 3 bits y que sólo contemple los números Naturales, lo llamamos *Sin Signo* y lo denotamos $BSS(3)$. Para analizar su rango se debe determinar el valor mínimo y máximo representables. Para el primer caso se interpreta la primer cadena: 000:

$$I(000) = 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 0$$

Para el segundo caso se interpreta la última cadena: 111:

$$I(111) = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$$

Es decir que el rango de $BSS(3)$ son todos los números naturales comprendidos entre 0 y 7, denotándose: $[0, 7]$. El conjunto de valores representables tiene 8 elementos. Además con 3 bits se pueden construir 8 cadenas de números representables, es decir, $2^3 = 8$. En la tabla 3.3 es posible ver la relación entre las cadenas y los valores que se representan en un sistema $BSS(3)$, a través de ambos mecanismos de representación e interpretación.

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

3.1.2. 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 a sumar “*en papel*” para el sistema decimal, para el sistema binario conviene pensar la suma como un algoritmo por columnas. Es decir que se suman las columnas empezando por 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 deben sumar dos dígitos cuya suma supera el límite de la base -que es 10-, como ser el caso de $9 + 3$, en el resultado se coloca el valor 2 y

rango

sistema
restringido

“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.

$$\begin{array}{r} 1 \\ + 9 \\ 3 \\ \hline 1 2 \end{array}$$

Esta práctica se denomina *acarreo* y representa el concepto de utilizar un nuevo orden de magnitud (en el ejemplo, al tener 10 unidades se *construye* una decena). El algoritmo para el caso del sistema binario es muy similar pero considerando la base 2 (ver algoritmo 3.2).

```

1 Comenzar con la columna menos significativa
2 Sea a el dígito del operando A en la columna actual. Sea b el
  dígito del operando B en la columna actual. Sea c el acarreo
  de la columna anterior.
3 Considerar la cantidad total de 1s:  $t = a + b + c$ 
4 Si  $t = 0$ : el resultado de la columna es 0 y no hay acarreo ( $c = 0$ )
5 Si  $t = 1$ : el resultado de la columna es 1 y no hay acarreo ( $c = 0$ )
6 Si  $t = 2$ : el resultado de la columna es 0 y SI hay acarreo ( $c = 1$ )
7 Si  $t = 3$ : el resultado de la columna es 1 y SI hay acarreo ( $c = 1$ )
8 Volver al paso 2 con la siguiente columna hacia la izquierda

```

Código 3.2: Algoritmo de suma binaria

Suponer por ejemplo que se debe computar la suma $001 + 011$. Entonces para la primera columna (paso 2 del algoritmo 3.2) $a=1$ y $b=1$ pero $c=0$ (pues al ser la primera columna no hay acarreo). Entonces $t=1+1+0=2$ y por lo tanto el resultado de la columna es 0, con acarreo.

$$\begin{array}{r} 1 \\ 0 0 1 \\ + 0 1 1 \\ \hline 0 \end{array}$$

A continuación se computa la segunda columna, considerando $a=0, b=1$ y $c=1$. Entonces $t=0+1+1=2$ y nuevamente el resultado de la columna es 0, con acarreo.

$$\begin{array}{r} 1 1 \\ 0 0 1 \\ + 0 1 1 \\ \hline 0 0 \end{array}$$

Finalmente, en la tercera y última columna se tiene $a=0, b=0$ y $c=1$. Entonces $t=0+0+1=1$ y el resultado de la columna es 1, sin acarreo.

$$\begin{array}{r} 1 1 \\ 0 0 1 \\ + 0 1 1 \\ \hline 1 0 0 \end{array} \leftarrow \text{resultado final}$$

Para comprobar la correctitud del procedimiento anterior, se deben interpretar las cadenas iniciales, calcular el resultado esperado y compararlo con

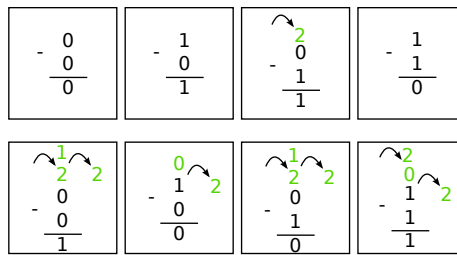


Figura 3.4: Casos posibles en una resta

la interpretación de la cadena resultado. Al interpretar las cadenas iniciales se tiene que:

$$I(001) = 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$

$$I(011) = 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$$

Entonces el resultado esperado es $1 + 3 = 4$, y su representación en BSS es

$$R(4) = 100$$

Esta cadena coincide con la cadena resultante del algoritmo. Esto permite concluir que el procedimiento y el resultado es correcto.

Resta Binaria

Al igual que en la suma, al momento de restar, se realizará un algoritmo que procesa columnas de derecha a izquierda, “pidiendo a la siguiente columna” en caso de ser necesario, es decir cuando el resultado de la columna podría ser menor a cero. Esto se da en dos situaciones: cuando el minuendo es menor al sustraendo, o cuando en la columna inmediata anterior se pidió a la columna actual.

Recuperando el algoritmo del sistema decimal, cuando es necesario se pide una magnitud mayor hacia la izquierda (“se pide 10”), que tiene relación con la base del sistema decimal. En el caso del sistema binario, el préstamo de una unidad en la columna de la izquierda se traduce en 2 unidades de la columna actual, pues la base del sistema es 2. En la Figura 3.4 se describen todas las situaciones posibles al computar la resta en una columna, considerando las combinaciones entre los operandos (que son los 2 bits de la columna) y el posible préstamo que haya quedado pendiente de la columna anterior (la de la derecha), y se explican mediante el algoritmo 3.3 de resta binaria.

```

1 Comenzar con la columna menos significativa
2 Sea a el dígito del operando A en la columna actual. Sea b el
  dígito del operando B en la columna actual. Sea c el préstamo a
  la columna anterior.
3 Si a-c es mayor o igual a b
4 entonces:
5   el resultado es t=a-c-b y no hay un nuevo préstamo (c=0)
6 sino:
7   se necesita pedir a la siguiente columna (c=1) y el resultado es
   t=(a+2)-c-b
8 Volver al paso 2 con la siguiente columna hacia la izquierda

```

Código 3.3: Algoritmo de resta binaria

Suponer por ejemplo que se debe computar la resta $101 - 011$. Entonces para la primer columna (paso 2 del algoritmo 3.3) $a = 1$ y $b = 1$ pero $c = 0$ (como en el caso de la suma, al ser la primer columna no hay préstamo). Entonces $t = 1 - 0 - 1 = 0$ y por lo tanto el resultado de la columna es 0, sin necesidad de un préstamo.

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 1 \\ \hline 0 \end{array}$$

A continuación se computa la segunda columna, y dado que $a - c = 0$ no es mayor a $b = 1$ entonces se pide a la siguiente columna ($c=1$) y el resultado es $t = 0 + 2 - 0 - 1 = 1$.

$$\begin{array}{r} \textcolor{red}{1} \\ 1 \ 0 \ 1 \\ + \ 0 \ 1 \ 1 \\ \hline 1 \ 0 \end{array}$$

Finalmente, en la tercera y última columna se tiene $a = 1$, $b = 0$ y $c = 1$. Entonces, dado que $a - c = 1 - 1 = 0$ es mayor o igual a $b = 0$ entonces el resultado es $t = a - c - b = 1 - 1 - 0 = 0$ y no hace falta hacer un préstamo.

$$\begin{array}{r} \textcolor{red}{1} \\ 1 \ 0 \ 1 \\ + \ 0 \ 1 \ 1 \\ \hline 0 \ 1 \ 0 \end{array} \leftarrow \text{resultado final}$$

También en esta operación es importante poder controlar el resultado. Como en el caso de la suma, se deben interpretar las cadenas iniciales, calcular el resultado esperado y compararlo con la interpretación de la cadena resultado.

Al interpretar las cadenas iniciales se tiene que:

$$I(101) = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

$$I(011) = 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$$

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

$$R(1) = 010$$

Esta cadena coincide con la cadena resultante del algoritmo. Esto permite concluir que el procedimiento y el resultado es correcto.

3.1.3. 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). Otras características que es posible transpolar del sistema decimal al sistema binario es el desplazamiento de cadenas.

Cadena original	Interpretación	Desplazamiento	Representación	Cadena final
10	$I(10) = 2^1$	$2^1 * 2 = 4$	$R(4) = 100$	100
10	$I(10) = 2^1$	$2^1 \% 2 = 1$	$R(1) = 1$	1
11	$I(11) = 2^1 + 2^0$	$3 * 2 = 6$	$R(6) = 110$	110
11	$I(11) = 2^1 + 2^0$	$3 \% 2 = 1$	$R(1) = 1$	1

Cuadro 3.4: Desplazamientos en cadenas binarias

Suponer que se tiene el valor decimal 17, si multiplicamos dicho valor por 10 se obtiene como resultado 170. Si se compara el 17 con el 170 es posible ver que este último se construye con el la cadena 17 y un 0 a su derecha. Ahora bien, si se multiplicara el 17 por 1000 en lugar de 10, se obtendría 17000, es decir, la cadena original 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 un 0 a derecha del numero original, y esto se relaciona con que el 10 puede pensarse como 10^1 ($10^1 = 10$). Similarmente, al multiplicar por 1000, se añaden tres 0s a derecha del numero original, y el 1000 es posible expresarlo 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**, y esto puede pensarse como un desplazamiento de la cadena que representa al valor original hacia la izquierda.

Dualmente, si se quiere conseguir un desplazamiento de **m** posiciones hacia la derecha, lo que se tiene que hacer es **dividir** el valor en 10^m . Por ejemplo, si se divide 250 en 10, el resultado es 25, es decir, se “borra” el ultimo dígito de la cadena original.

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 esto.

3.2. Sistema Hexadecimal

Como se explicó anteriormente las computadoras trabajan con el sistema binario. Para un autómata como la CPU leer cadenas de 0s y 1s no resulta un problema, sin embargo, para el programador que escriba el programa, leer dichas cadenas resulta algo confuso. La solución a este problema es el uso del sistema hexadecimal, el cual facilita la interpretación de las cadenas que representan las instrucciones e información que utilizara nuestro programa

Este es un sistema nuevo. En este caso, tal como su nombre lo indica, el sistema opera con 16 símbolos. Como no alcanza con los conocidos del sistema decimal que van del 0 al 9, se agregan también letras del alfabeto latino, quedando los 16 símbolos 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 podremos:

Dígito	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Cuadro 3.5: Valor de los dígitos hexadecimales

1. Interpretar
2. Representar
3. Calcular su rango
4. Agrupación de bits

3.2.1. Interpretación

Traspolando el mismo razonamiento que hicimos para el sistema binario, en este caso asignamos una potencia de 16 a cada posición comenzando de derecha a izquierda (desde el 0) y ubicamos nuestra cadena respetando las posiciones.

Consideremos los siguientes ejemplos:

- Queremos interpretar la cadena **128**, entonces:

$$1 \times 16^2 + 2 \times 16^1 + 8 \times 16^0$$

El resto es sólo resolver la suma.

- Ahora bien, qué pasa si la cadena que necesitamos interpretar contiene una letra: **2A**, entonces, siguiendo la misma lógica quedaría:

$$2 \times 16^1 + A \times 16^0$$

Pero, ¿cómo se resuelve una multiplicación con una letra en el segundo término? Para ello el sistema cuenta establece una relación entre los dígitos $\{0,1,\dots,9,A,\dots,F\}$ y la cantidad (o valor) que representan individualmente. Esto se describe en la tabla 3.5 y es necesario a la hora de aplicar el mecanismo de interpretación.

Entonces, continuando con el ejemplo anterior, la cadena **2A** tiene el valor:

$$I(2A) = 2 \times 16^1 + 10 \times 16^0$$

- Como último ejemplo, considerar la cadena **A3F**

$$I(A3F) = 10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0$$

3.2.2. Representación

Siguiendo la lógica del sistema binario, para representar valores mediante cadenas 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

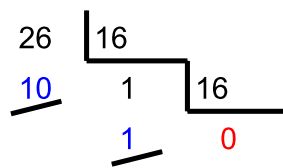


Figura 3.5: Representación del valor 26 en el sistema hexadecimal

Uno de los restos es 10, entonces debemos traducirlo a la letra correspondiente aplicando la tabla de interpretación de hexadecimal. 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 debemos calcular el mínimo número representable interpretando la cadena más chica y la más grande. Siendo el rango todos los números comprendidos entre ambos. Supongamos el sistema hexadecimal de 2 dígitos:

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

$$0 \times 16^1 + 0 \times 16^0 = 0$$

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

$$15 \times 16^1 + 15 \times 16^0 = 255$$

(aplicando la tabla de interpretación de hexadecimal)

Por lo tanto el rango de este sistema es:

$$[0, 255]$$

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 más significativa) y considerando la base de este sistema, que es 16 (ver algoritmos 3.4 y 3.5)

Símbolo	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

Cuadro 3.6: Relación entre cadenas $BSS(4)$ y símbolos del sistema hexadecimal

```

1 Comenzar con la columna menos significativa
2 Sea a el dígito del operando A en la columna actual. Sea b el
  dígito del operando B en la columna actual. Sea c el arrastre
  de la columna anterior.
3 Considerar la suma entre los dígitos de la columna:  $t = a+b+c$ 
4 Si  $t$  es menor a la base (16):
5   entonces el resultado intermedio (i) es  $t$  y no hay acarreo ( $c=0$ )
6   sino el resultado intermedio (i) es  $t-16$  y hay acarreo ( $c=1$ )
7 Si  $i=10$ : el resultado final es  $r=A$ 
8 Si  $i=11$ : el resultado final es  $r=B$ 
9 Si  $i=12$ : el resultado final es  $r=C$ 
10 Si  $i=13$ : el resultado final es  $r=D$ 
11 Si  $i=14$ : el resultado final es  $r=E$ 
12 Si  $i=15$ : el resultado final es  $r=F$ 
13 Volver al paso 2 con la siguiente columna hacia la izquierda

```

Código 3.4: Algoritmo de suma hexadecimal

Para abordar el algoritmo 3.5 es necesario recuperar lo indicado en el cuadro 3.5 que mapea cada dígito hexadecimal con su valor. Este cuadro se utiliza en el paso 3 del algoritmo 3.5.

```

1 Comenzar con la columna menos significativa
2 Sea a el dígito del operando A en la columna actual. Sea b el
  dígito del operando B en la columna actual. Sea c el préstamo a
  la columna anterior (que puede ser 1 o 0).
3 Sea va el valor de A, vb el valor de B y vc el valor de c.
4 Si  $va-vc$  es mayor o igual a vb
5   entonces:
6     el resultado intermedio es  $i=va-vc-vb$  y no hay un nuevo préstamo
      ( $c=0$ )
7   sino:
8     se necesita pedir a la siguiente columna ( $c=1$ ) y el resultado
      intermedio es  $i=(va+16)-vc-vb$ 
9 Si  $i=10$ : el resultado final es  $r=A$ 
10 Si  $i=11$ : el resultado final es  $r=B$ 
11 Si  $i=12$ : el resultado final es  $r=C$ 
12 Si  $i=13$ : el resultado final es  $r=D$ 
13 Si  $i=14$ : el resultado final es  $r=E$ 
14 Si  $i=15$ : el resultado final es  $r=F$ 
15 Volver al paso 2 con la siguiente columna hacia la izquierda

```

Código 3.5: Algoritmo de resta hexadecimal

3.2.5. Agrupación de bits

Este es un método establece una relación directa entre cadenas del sistema binario y cadenas del sistema hexadecimal, que 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.

Dado que cada cuarteto es alguna de las combinaciones de 4 bits del sistema $BSS(4)$ y por lo tanto el rango que cubren es $[0,15]$. Considerando que dichos valores del rango se pueden representar por un solo caracter hexadecimal, entonces se aplica la tabla 3.6 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:

$$I(96A5) = 9 \times 16^3 + 6 \times 16^2 + 10 \times 16^1 + 5 \times 16^0 =$$

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

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

Aplicando la propiedad distributiva:

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

Reemplazando los valores por sus equivalentes potencias de 2:

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

Simplificando los términos:

$$= 2^3 \times 2^{12} + 2^{12} + 2^2 \times 2^8 + 2 \times 2^8 + 2^3 \times 2^4 + 2 \times 2^4 + 2^2 + 1$$

Aplicando la propiedad de potencias de igual base:

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

Y esto ultimo, por definicion, equivale a la interpretación de la cadena 1001011010100101:

$$I(1001011010100101) = 2^{15} + 2^{12} + 2^{10} + 2^9 + 2^7 + 2^5 + 2^2 + 2^0$$

Notar que no se incluyen los bits en cero

Capítulo 4

Representación de instrucciones. Q1

Las computadoras funcionan siguiendo una lista de instrucciones que se les dan, y que les permiten ordenar, encontrar y generar información. Pero estas instrucciones deben estar disponibles de alguna manera, en alguna codificación, que la computadora pueda interpretar y traducir en las acciones que se requieren para cada instrucción. Como se verá en el capítulo 5, las computadoras actuales están construidas con circuitos digitales y por lo tanto necesitan que la mencionada codificación se le provea como 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 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 denomina **repertorio de instrucciones**.

Un punto de encuentro en que la persona que diseña el computador y la persona que lo programa pueden ver la misma máquina es el **repertorio de instrucciones**. Desde el punto de vista de la persona que diseña, el conjunto de instrucciones máquina constituye la especificación o requisitos funcionales del procesador: implementar el procesador es una tarea que, en buena parte, implica construir circuitos que cumplan con esos requisitos. Desde el punto de vista de la persona que programa, las instrucciones de ese lenguaje ensamblador son las herramientas disponibles para expresar los programas. Además, debe conocer la estructura de registros y de memoria, los tipos de datos que acepta directamente la máquina y el funcionamiento (o restricciones) de la ALU (Unidad Aritmético-lógica).

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, etc). La operación se indica mediante un código binario denominado código de operación, o de manera abreviada, *codop* .

- **Referencia a los operandos origen:** la operación puede implicar uno o más operandos origen, es decir operandos que son entradas 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.

Registros de
uso general

4.1. 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 los *codops* se representan mediante abreviaturas, denominadas *nemotécnicos*, que indican la operación en cuestión. Ejemplos usuales son:

ADD	Sumar
SUB	Restar
MUL	Multiplicar
DIV	Dividir
LOAD	Cargar datos desde memoria
STORE	Almacenar datos en memoria

Estos dos últimos no se aplican a la Arquitectura Q.

código
máquina

Los operandos también suelen representarse simbólicamente. Por ejemplo, la instrucción `ADD R,Y` puede significar sumar el valor contenido en la posición de datos "Y" al contenido del registro `R`". En este ejemplo, `Y` hace referencia a la dirección de una posición de memoria, y `R` a un registro particular. Observe que la operación se realiza con el contenido de la posición y no con su dirección.

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. Al proceso de traducir del código fuente a código máquina se lo denomina ensamblar.

Es raro encontrar ya 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 lenguaje ensamblador. Luego este se ensambla y se carga en memoria, aplicando alguna política de administración de memoria y cuando se necesita dicho programa se ejecuta (ver figura 4.1), muchas veces a partir de la acción concreta de un/a usuario/a.

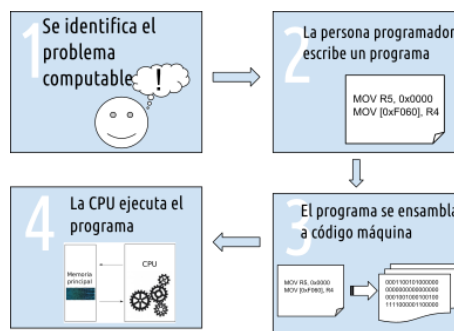


Figura 4.1: Ciclo de Vida de un programa

4.2. Ejecución de un programa

Como se anticipó, la ejecución de un programa es la ejecución ordenada de las instrucciones que lo componen, y la ejecución de cada una respeta un *ciclo de ejecución de instrucciones* que cuenta con 3 etapas: Búsqueda, Decodificación y Ejecución (ver Figura 4.2). Específicamente, la *búsqueda de instrucción* se encarga de recuperar el código máquina completo de la instrucción y ponerlo disponible en un *registro 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* o IR. La etapa de decodificación debe ser capaz de identificar la tarea que resuelve la instrucción y si requiere operandos. La ejecución de la instrucción normalmente se traduce en que la UC delegue a otro circuito la realización de la tarea específica para cada instrucción, por ejemplo en el caso de las instrucciones aritméticas que son delegadas a la ALU.

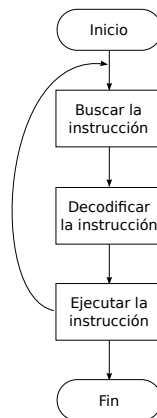


Figura 4.2: Ciclo de ejecución de instrucción

Por ejemplo, para escribir el programa que resuelva $A=A-B$ se necesita poder escribir la instrucción: **restar** LO-QUE-HAY-EN-B a LO-QUE-HAY-EN-A. Entonces surge la necesidad de tener una representación binaria, que pueda ser entendida por la unidad de control, quien le dirá a la ALU que debe activar una resta, y por lo tanto que debe usar el circuito restador. Esa 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 registros de uso general. En este caso necesitaremos un registro para A y otro para B. El resultado se almacena en el registro A: $A=A-B$.

Veamos un ejemplo con una suma entre el registro A y B: $A = A + B$.

- Al comienzo del ciclo de ejecución de esta instrucción, la UC hace la lectura de instrucción, obteniendo la cadena correspondiente (código máquina) y copiándola a su vez en el IR.
- La UC decodifica la instrucción: El codop (en este caso una suma) se transmite a la ALU junto con los operandos. El Operando Destino en este caso es A y el Operando Origen es B
- La ALU ejecuta la suma y el resultado se dirige al registro A.

4.3. Arquitectura Q1

Para poner en práctica los conceptos utilizaremos una arquitectura conceptual que llamaremos Q. La misma está pensada en versiones acumulativas (ver

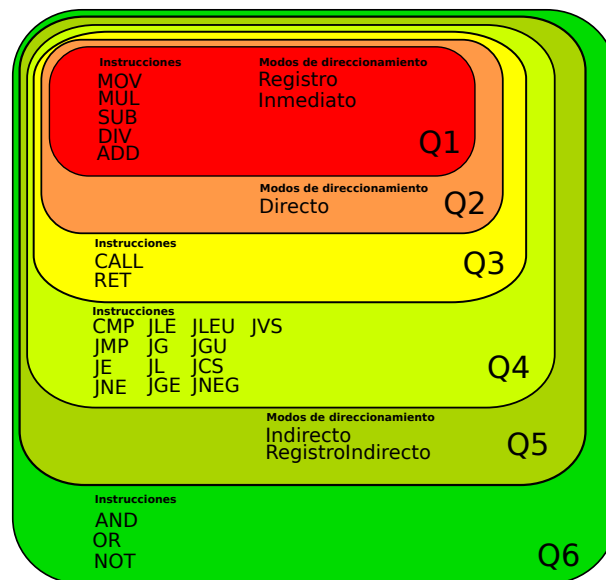


Figura 4.3: Capas de la Arquitectura Q

figura 4.3) y la primera recibe el nombre de **Q1**, con la cual relacionaremos las diferentes etapas que intervienen en el ciclo de vida de un programa, y del ciclo de ejecución de las instrucciones.

Q1 tiene las siguientes características:

- Ocho registros de uso general de 16 bits (también denominados visibles a la persona que programa), es decir que son variables disponibles para usar en los programas, denominados R0 a R7.
- Cinco instrucciones de 2 operandos: ADD (suma), SUB (resta), MUL (multiplicación), DIV (división entera) y MOV (asignación).
- Los operandos pueden ser referencias a variables (uno de los registros) o valores constantes.
- Los valores constantes tienen 16 bits, y se escriben en hexadecimal con la sintaxis: `0xhhhh` (donde 'h' es un caracter hexadecimal)
- Al primer operando se lo denomina *operando destino*, por ser además donde se almacenará el resultado, y al segundo se lo denomina *operando origen*.
- El sistema de numeración soportado por esta arquitectura es BSS, considerando una ALU (Unidad Aritmético-Lógica como se describirá en el capítulo 5

Un ejemplo de una instrucción sintácticamente válida en el lenguaje Q1 es `MOV R5, 0xFF00`, que copia el valor constante FF00 al registro R5. Otro ejemplo

es **ADD R5, R3**, que suma el contenido de R3 con el contenido R5 y almacena el resultado en R5 (operando destino).

4.3.1. Ensamblar instrucciones en Q1

Para que los programas escritos en el lenguaje de Q1 sean ejecutables por una computadora, deben estar escritos en código máquina. Entonces es necesario establecer las reglas de traducción código fuente a código máquina que debe aplicar un ensamblador. Esas reglas se conocen como **formatos de instrucciones**. Estos definen la organización de los bits dentro de una instrucción en términos de las partes que la componen.

La instrucción debe incluir información acerca de **la operación a ejecutar** y sobre **qué datos**. Entonces, mínimamente debe incluir el código de la operación y un mecanismo para llegar hasta el/los operando/s, mediante lo que llamamos *modos de direccionamiento*.

modos de
direccionamiento

Direccionamiento Inmediato Es la forma más sencilla de direccionamiento, el operando está presente en la propia instrucción. Este modo puede utilizarse para definir y utilizar constantes o para fijar valores iniciales (inicializar) de variables.

Direccionamiento Registro El operando ahora se encuentra en un registro del sistema. La ventaja principal de éste modo es que solo es necesario un pequeño campo de direcciones en la instrucción. Además, el tiempo de acceso a un registro interno de CPU es muy reducido. La desventaja principal es que el espacio de direccionamiento está limitado, pues se espera que la cantidad de registros visibles al programador no sea muy grande.

El siguiente es el formato de las instrucciones de la versión 1 de Q (es decir, Q1), que son 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)
-------------	------------------	-----------------	-------------

En los campos modo destino y modo origen se codifica el modo de direccionamiento (inmediato o registro) de cada uno de los operandos. Y en particular, en el modo registro se indica también *de qué registro se trata*. Esos 6 bits se completan considerando el cuadro 4.1. El campo Origen es opcional, pues es necesario sólo cuando el operando origen es Inmediato. Asimismo, las instrucciones que tienen en Modo Destino operandos del tipo Inmediato son consideradas como inválidas por el procesador. Por otro lado, el campo de codop se completa con el **código de operación** correspondiente a cada instrucción, siguiendo la codificación indicada en el cuadro 4.2, donde se detallan las instrucciones de Q1, mostrando el código de operación y el efecto esperado. Es importante notar que dada la naturaleza de la multiplicación, el resultado de la operación MUL es de 32 bits en lugar de 16, ocupando el registro R7 (los 2 bytes mas significativos) y el operando Destino (los 2 bytes menos significativos). Además, la operación DIV es una división entera en el sistema BSS(16).

Para comprender mejor el porqué la multiplicación afecta implícitamente (es decir, aunque no forme parte de la instrucción) a R7, es necesario repasar

Modo	Codificación	Sintaxis
Inmediato	000000	0xHHHH
Registro	100rrr	Rx

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

Cuadro 4.1: Códigos de los modos de direccionamiento en Q1

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

Cuadro 4.2: Codigos de operación y efecto de las instrucciones en Q1

$$\begin{array}{r}
 999 \\
 \times 999 \\
 \hline
 8991 \\
 8991- \\
 8991-- \\
 \hline
 9998001
 \end{array}$$

Figura 4.4: Ejemplo de resultado de la multiplicación en decimal

$$\begin{array}{r}
 1111 \\
 \times 1010 \\
 \hline
 0000 \\
 1111- \\
 0000-- \\
 1111---- \\
 \hline
 10010110
 \end{array}$$

Figura 4.5: Ejemplo de resultado de la multiplicación en binario

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

Cuadro 4.3: Ejemplos de instrucciones ensambladas

cómo funciona dicha operación en otro sistema de uso más cotidiano. Considerar entonces la operación 999×999 , donde se trabaja con cadenas decimales de 3 dígitos (si se anotara de la misma manera que en BSS, sería `decimal(3)`). Esta operación, al realizarse, da como resultado 998001 (ver figura 4.4), es decir que partiendo de dos cadenas con 3 dígitos, se obtiene una cadena de hasta 6 dígitos (el doble de dígitos en comparación a los operadores usados). Esto se debe a la naturaleza de la multiplicación, en donde por cada dígito del segundo operando se desplaza la cadena un lugar hacia la izquierda.

Este mecanismo es aplicable al sistema binario también. Entonces suponer la multiplicación entre las cadenas 1111 y 1010, en el sistema BSS(4) como se muestra en la figura 4.5. En este caso se obtiene una cadena de 8 bits, cuando los operandos eran cada uno de 4 bits.

Si se considera esta característica a la hora de diseñar una arquitectura, se debe asumir el caso del resultado mas grande. En particular a la arquitectura Q, al multiplicar dos cadenas binarias de 16 bits, es posible obtener como máximo una cadena de 32 bits. Sin embargo, ¿cómo almacenar 32 bits en las celdas o los registros de Q, que son de 16 bits? Una posible solución a este problema, es almacenar los 16 bits más significativos del resultado en R7 y los 16 menos significativos en el operando destino, de modo que ante la cadena xxxx xxxx xxxx xxxx yyyy yyyy yyyy yyyy los bits x se guardaran en R7, y los bits y en el destino. En caso de que el resultado sea una cadena con una cantidad de bits menor a 32, la ALU completa con ceros a la izquierda hasta llegar a 32 bits.

De esta manera, las instrucciones arriba mencionadas se ensamblan como indica el cuadro 4.3. 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.

4.3.2. Ejecución de programas Q1

Como se indicó antes, la ejecución de un programa Q1 es la ejecución de un ciclo de ejecución (ver Figura 4.2) por cada instrucción del programa. A modo de integración en un caso de ejemplo, en esta sección se llevará a cabo una simulación de la ejecución del siguiente programa, para lo cual se asume que el registro R5 tiene la cadena 0001

```
1 1940 000F
2 2963
3 0940 0002
```

Código 4.1: Ejemplo de código máquina hexadecimal

En un primer momento (el primer ciclo), la Unidad de Control recibe el código máquina 1940 000F. Al decodificar la instrucción, la UC identifica una operación de copiado de datos (MOV) y aplica el formato de instrucción correspondiente, encontrando que el destino es la variable R5, y el origen es un dato constante (el valor 000F). Finalmente se ejecuta la efecto de la instrucción, ocasionando que el valor que antes tenía R5 se reemplace con 000F.

En el segundo ciclo, la UC recibe el código máquina 2963. De la misma manera que antes, aplica el formato de instrucción y entiende que se trata de una operación de suma entre dos variables, R5 y R3, y que el destino del resultado debe ser R5. Entonces delega el cómputo a la ALU -la suma entre los valores 000F y 0001- obteniendo el resultado 0010. Dicho resultado se almacena en el registro R5.

Al comenzar el tercer ciclo la UC recibe el código máquina 0940 0002. Al decodificar esta tercer instrucción identifica la operación de multiplicación entre el valor almacenado en el registro R5 y el valor constante 0002. También en este caso se delega el cómputo a la ALU con los valores de entrada 0010 y 0002 arrojando como resultado 0020, que se debe almacenar en el registro R5.

En resumen, se tiene que:

- Primer ciclo

Búsqueda Se lee 1940 000F

Decodificación Operación MOV hacia R5 con un inmediato

0001	100101	000000	0000000000001111
------	--------	--------	------------------

Ejecución (efecto) $R5 \leftarrow 000F$

- Segundo ciclo

Búsqueda Se lee 2963

Decodificación Operación ADD entre R5 y R3

0010	100101	100011	--
------	--------	--------	----

Ejecución (efecto) $R5 \leftarrow 000F + 000F = 0010$

- Tercer ciclo

Búsqueda Se lee 0940 0002

Decodificación Operación MUL entre R5 y un inmediato

0000	100101	000000	0000000000000010
------	--------	--------	------------------

Ejecución (efecto) $R5 \leftarrow 0010 \times 0002 = 0020$

4.3.3. Prueba de programas: prueba de escritorio

La *prueba de escritorio* es un método para analizar si un programa cumple con lo que se espera de él. Para realizar esta prueba se debe definir un escenario inicial concreto y un escenario final o esperado, para luego **simular la ejecución del programa a alto nivel**, indicando los cambios en cada instrucción. Finalmente se debe comparar el resultado obtenido con el resultado que se esperaba, para concluir si la prueba falló o fue exitosa.

Suponer, por ejemplo, que se cuenta con un programa en Q1 que calcula el área de un trapecio:

$$A = \frac{(B + b) \times h}{2}$$

donde la base mayor B está almacenada en R0, la base menor b en R1, y la altura h en R2. El programa que calcula el área, dejándolo el resultado en el registro R0, es como sigue:

```
ADD R0, R1
MUL R0, R2
DIV R0, 0x0002
```

Suponiendo que como **escenario inicial** se tiene que el registro R0 contiene la cadena 0003, el registro R1 contiene la cadena 0002, y el registro R2 contiene la cadena 0002, entonces se espera que el resultado sea R0=0005. Este resultado tiene relación con que la interpretación de estas cadenas es:

$$I_{hexa}(0003) = 3 \times 16^0 = 3 \quad I_{hexa}(0002) = 2 \times 16^0 = 2$$

por otro lado el área de un trapecio con esas dimensiones es

$$A = \frac{(2 + 3) \times 2}{2} = 5$$

y finalmente este resultado se representa:

$$R_{hexa(4)}(5) = 0005$$

La ejecución de la primera instrucción, que realiza una suma entre las cadenas almacenadas en R0 y R1, dejando el resultado en R0, y tiene como efecto:

$$R0 \leftarrow 0003 + 0002 = 0005$$

Asimismo, la ejecución de la segunda instrucción es una multiplicación sobre R0, con el valor almacenado en R2. Entonces su efecto es

$$R0 \leftarrow 0005 \times 0002 = 000A$$

Finalmente, la última instrucción divide el contenido de R0 (000A) por la constante 2, describiéndose como efecto:

$$R0 \leftarrow 000A / 0002 = 0005$$

Una vez finalizado el análisis paso a paso, se puede concluir que el resultado esperado es el obtenido, y por lo tanto el programa funciona correctamente en este escenario. En el siguiente esquema se resume lo explicado hasta aquí. ¿Es posible pensar otros escenarios donde el programa no funcione?

Escenario de prueba	
Datos iniciales	R0=0003, R1=0002, R2=0002
Resultado esperado	R0=0005
Ejecución paso a paso	
Instrucción	Efecto
ADD R0, R1	$R0 \leftarrow 0005$
MUL R0, R2	$R0 \leftarrow 000A$
DIV R0, 0x0002	$R0 \leftarrow 0005$
Conclusión	
Resultado esperado = resultado obtenido: ¡Prueba exitosa!	

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 2.2), 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.

circuitos
digitales

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

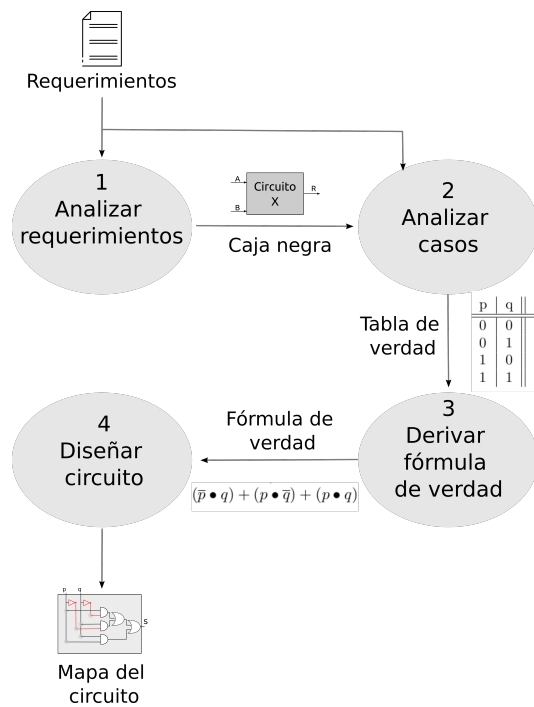


Figura 5.1: Diseño de un circuito digital

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.

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

Cuadro 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.

tabla de
verdad

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.

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 $\rightarrow \bar{p} \bullet q$
1	0	1 $\rightarrow p \bullet \bar{q}$
1	1	1 $\rightarrow p \bullet q$

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

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 fórmula vale 0, por ejemplo:

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

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

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

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:

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

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 ($A \bullet B \bullet C$)

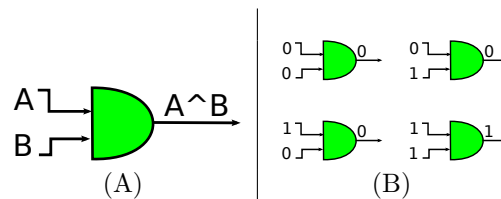
Cuadro 5.2: Casos positivos del circuito **mayoría**

Figura 5.3: Compuerta AND

5.2. Mapa del circuito

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

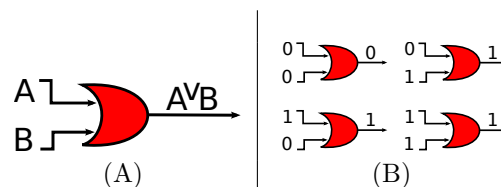


Figura 5.4: Compuerta OR

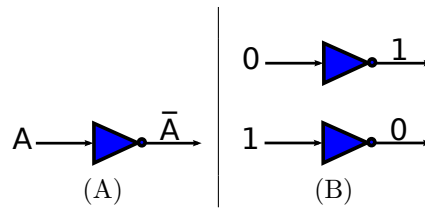


Figura 5.5: Compuerta NOT

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

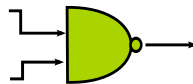


Figura 5.6: Compuerta NAND

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)

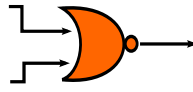


Figura 5.7: Compuerta NOR

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

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

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

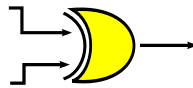


Figura 5.8: Compuerta XOR

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 \bar{e}_2 a una compuerta AND, siguiendo un término ($e_1 \bullet \bar{e}_2$). 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).

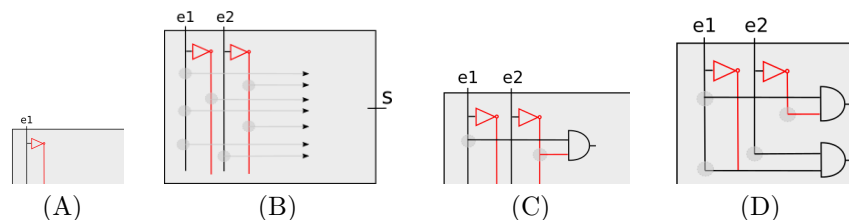


Figura 5.9: Conexión de entradas y compuertas

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 $(a \bullet \bar{b}) + (\bar{a} \bullet b)$.

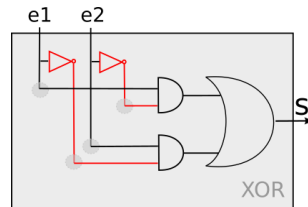


Figura 5.10: Circuito que implementa la función XOR

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:

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

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

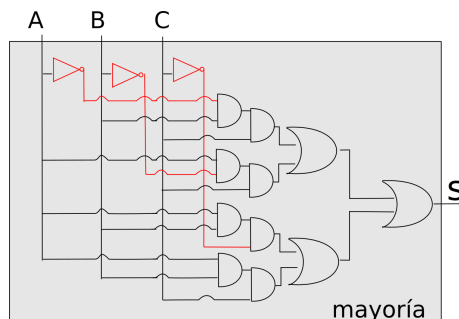


Figura 5.11: Mapa del circuito *Mayoría*

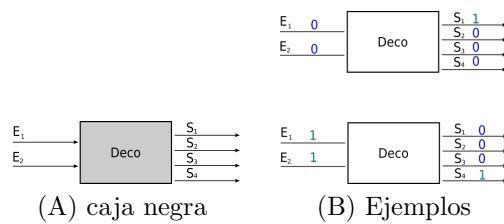


Figura 5.12: Decodificador de N=2

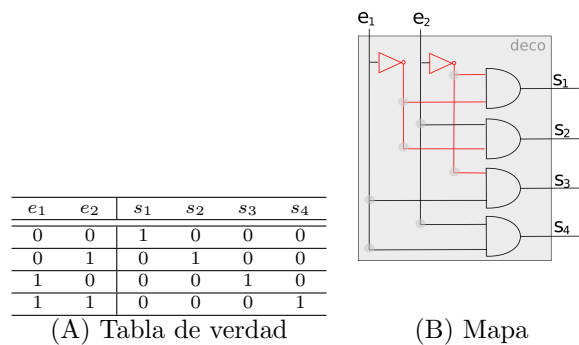


Figura 5.13: Decodificador de N=2

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 descritos 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

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

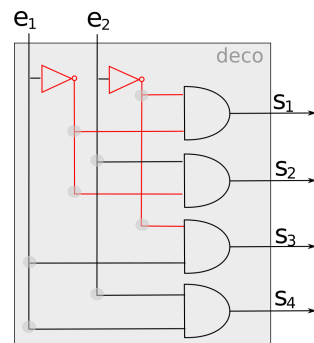


Figura 5.14: Decodificador

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:

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

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

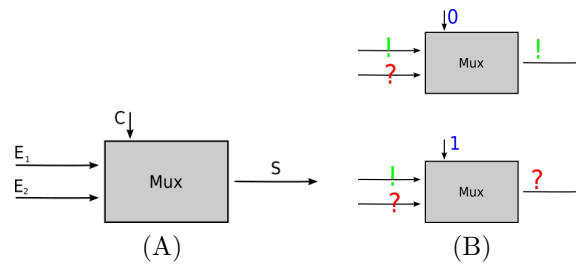
$$s_3 = e_1 \bullet \overline{e_2}$$

$$s_4 = e_1 \bullet e_2$$

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

Cuadro 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:

$$S = (\bar{c} \bullet e_1 \bullet \bar{e}_2) + (\bar{c} \bullet e_1 \bullet e_2) + (c \bullet \bar{e}_1 \bullet e_2) + (c \bullet e_1 \bullet e_2)$$

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 términos 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:

$$\begin{aligned} S &= (\bar{c} \bullet e_1 \bullet \bar{e}_2) + (\bar{c} \bullet e_1 \bullet e_2) + (c \bullet \bar{e}_1 \bullet e_2) + (c \bullet e_1 \bullet e_2) = \\ &= \bar{c} \bullet e_1 \bullet (\bar{e}_2 + e_2) + c \bullet e_2 \bullet (\bar{e}_1 + e_1) = \end{aligned}$$

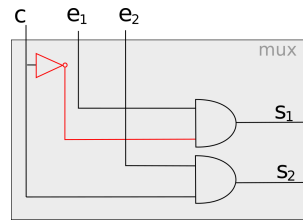


Figura 5.16: Multiplexor

$$\begin{aligned}
 &= \bar{c} \bullet e_1 \bullet 1 + c \bullet e_2 \bullet 1 \\
 &= \bar{c} \bullet e_1 + c \bullet e_2
 \end{aligned}$$

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.

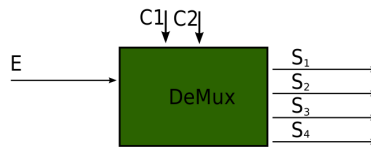


Figura 5.17: Demultiplexor

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, y por lo tanto estos circuitos operan sobre dos cadenas binarias. La ALU se puede implementar mediante circuitos aritméticos, es decir que 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 o no acarreo (*carry*). Por lo tanto recibe 2 entradas (los operandos a sumar)

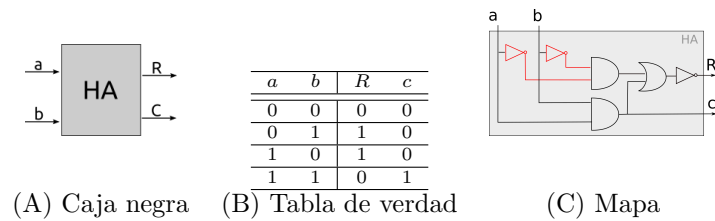


Figura 5.18: Circuito semisumador

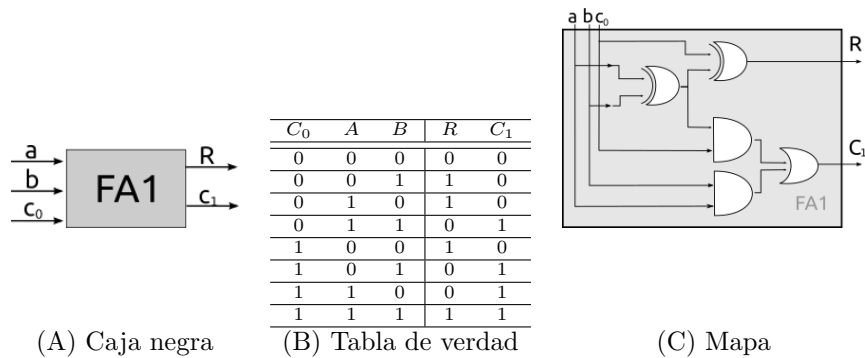


Figura 5.19: Circuito sumador para BSS(1)

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:

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

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

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 descrita 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.

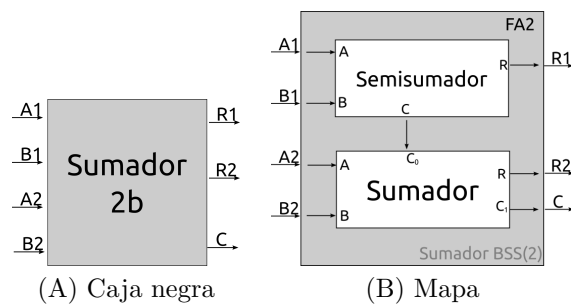
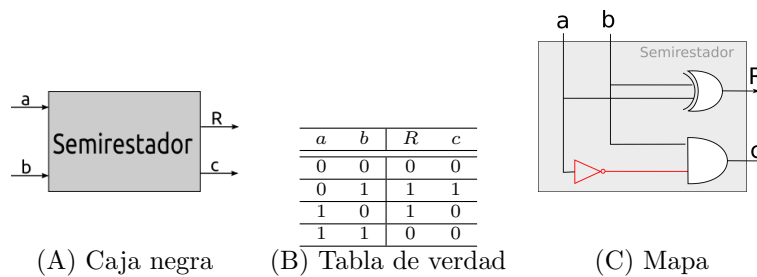
Figura 5.20: Circuito sumador para *BSS(2)*

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).

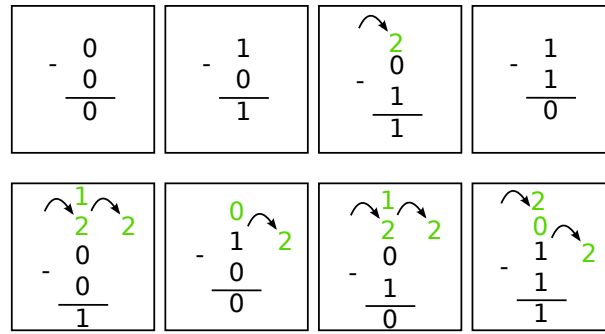


Figura 5.22: Casos posibles en una resta con carry

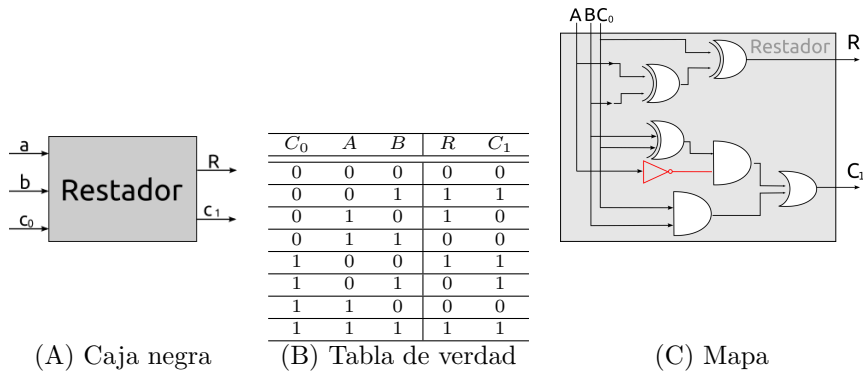


Figura 5.23: Restador BSS(1)

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 $A = \text{"hoy es viernes"}$ o $B = \text{"hoy está lloviendo"}$, o pueden ser complejas, (composición de proposiciones simples) como por ejemplo: $\text{"hoy es viernes y hoy está lloviendo"}$.

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	$\neg A$

Cuadro 5.5: Simbología utilizada para las funciones lógicas

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

- $p = \text{"está lloviendo"}$
- $q = \text{"el sol está brillando"}$
- $r = \text{"hay nubes en el cielo"}$

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: $\bar{p} + (\bar{q} \bullet r)$

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 último, en el caso de la *negación*, si la proposición p es verdadera, su negación será falsa y viceversa (not).

conjunción

disyunción

negación

Conjunción			Disyunción		
p	q	$p \bullet q$	p	q	$p + q$
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	1

Negación	
p	\bar{p}
0	1
1	0

Cuadro 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 $\overline{(a \oplus b)} = (\bar{b} \bullet \bar{a}) + (a \bullet b)$, mediante las propiedades (identidades lógicas) indicadas.*

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

Lemma 5.5.2 *En este lema se demuestra que la Suma de Productos del Carry de salida del restador (apartado 5.4.1), $(\bar{C}_0 \bullet \bar{A} \bullet B) + (C_0 \bullet \bar{A} \bullet \bar{B}) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet B)$, es equivalente a la expresión $\bar{A} \bullet (C_0 \oplus B) + (C_0 \bullet B)$, mediante las propiedades (identidades lógicas) indicadas.*

$$\begin{aligned}
 &(\bar{C}_0 \bullet \bar{A} \bullet B) + (C_0 \bullet \bar{A} \bullet \bar{B}) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet B) \\
 \text{Ley distributiva} &= \bar{A} \bullet (\bar{C}_0 \bullet B + C_0 \bullet \bar{B}) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet B) \\
 \text{Ley distributiva} &= \bar{A} \bullet (\bar{C}_0 \bullet B + C_0 \bullet \bar{B}) + (C_0 \bullet B) \bullet (\bar{A} + A) \\
 \text{Ley del medio excluido} &= \bar{A} \bullet (\bar{C}_0 \bullet B + C_0 \bullet \bar{B}) + (C_0 \bullet B) \bullet (1) \\
 \text{Ley de identidad} &= \bar{A} \bullet (\bar{C}_0 \bullet B + C_0 \bullet \bar{B}) + (C_0 \bullet B) \\
 \text{definición de XOR} &= \bar{A} \bullet (C_0 \oplus B) + (C_0 \bullet B)
 \end{aligned}$$

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

$$\begin{aligned}
& (\bar{C}_0 \bullet \bar{A} \bullet B) + (C_0 \bullet \bar{A} \bullet \bar{B}) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet B) = \\
& \quad \text{por Ley distributiva en términos 2 y 4} \\
& = (\bar{C}_0 \bullet \bar{A} \bullet B) + C_0 \bullet [(\bar{A} \bullet \bar{B}) + (A \bullet B)] + (C_0 \bullet \bar{A} \bullet B) = \\
& \quad \text{por Lema 5.5.1} \\
& = (\bar{C}_0 \bullet \bar{A} \bullet B) + C_0 \bullet [\overline{A \oplus B}] + (C_0 \bullet \bar{A} \bullet B) = \\
& \quad \text{por Ley asociativa} \\
& = (\bar{C}_0 \bullet \bar{A} \bullet B) + (C_0 \bullet \bar{A} \bullet B) + C_0 \bullet [\overline{A \oplus B}] = \\
& \quad \text{por Ley distributiva en términos 1 y 2} \\
& = \bar{A} \bullet B \bullet (\bar{C}_0 + C_0) \bullet \overline{A \oplus B} = \\
& \quad \text{por Ley del medio excluido} \\
& = \bar{A} \bullet B \bullet (1) \bullet \overline{A \oplus B} = \\
& \quad \text{por Ley de identidad} \\
& = \bar{A} \bullet B \bullet \overline{A \oplus B} =
\end{aligned}$$

Lemma 5.5.4 *En este lema se demuestra que $(\bar{C}_0 \bullet A \bullet B) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet \bar{B}) + (C_0 \bullet A \bullet B) = A \bullet B + C_0 \bullet (A \oplus B)$, mediante las propiedades (identidades lógicas) indicadas.*

$$\begin{aligned}
& (\bar{C}_0 \bullet A \bullet B) + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet \bar{B}) + (C_0 \bullet A \bullet B) \\
& \text{Ley distributiva (terminos 1 y 4)} = (\bar{C}_0 + C_0) \bullet A \bullet B + (C_0 \bullet \bar{A} \bullet B) + (C_0 \bullet A \bullet \bar{B}) \\
& \text{Ley distributiva(terminos 3 y 4)} = (\bar{C}_0 + C_0) \bullet A \bullet B + C_0 \bullet (\bar{A} \bullet B + A \bullet \bar{B}) \\
& \text{Ley del medio excluido} = (1) \bullet A \bullet B + C_0 \bullet (\bar{A} \bullet B + A \bullet \bar{B}) \\
& \text{Ley de identidad} = A \bullet B + C_0 \bullet (\bar{A} \bullet B + A \bullet \bar{B}) \\
& \text{Definición XOR} = A \bullet B + C_0 \bullet (A \oplus B)
\end{aligned}$$

Capítulo 6

Memoria y Buses. Q2

La memoria principal es un conjunto de *celdas*, donde cada celda es un dispositivo que almacena una cadena de bits mientras esté alimentada eléctricamente. El conjunto de celdas es homogéneo, es decir que todas las celdas tienen la misma capacidad, expresada en cantidad de bits. Además, cada celda tiene una *dirección* que la identifica para permitir su lectura o escritura, por eso se dice que es la *unidad direccionable más pequeña*.

A la memoria principal se la conoce también como memoria RAM (Memoria de acceso aleatorio). Se denominan de acceso aleatorio porque 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 6.3.1.

En la figura 6.1 se ilustra una memoria de 4 celdas, cuyas direcciones son 00,10,10 y 11.

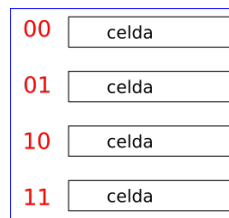


Figura 6.1: memoria de 4 celdas

Las celdas se agrupan en palabras, en general, el tamaño de palabra coincide con la cantidad de datos que se leen en la misma operación. En la arquitectura de Q2, una palabra es equivalente a una celda pues las variables son de 16 bits (registros y celdas de memoria) y las instrucciones aritméticas manejan esa cantidad de bits al mismo tiempo.

6.1. Memoria Principal e instrucciones

Para comprender la ejecución de los programas, es necesario considerar los requisitos que tiene que cumplir el procesador (CPU):

1. Buscar instrucciones desde la memoria o desde un dispositivo de entrada y salida
2. Decodificar instrucciones para determinar que acción es necesaria
3. Buscar los datos que la ejecución de la instrucción puede requerir
4. Procesar los datos que la ejecución puede necesitar a través de una operación lógica o aritmética
5. Almacenar los resultados donde corresponda

Para comprender los pasos anteriores (ver figura 4.2). En este apartado se detallará cómo participa la mencionada memoria principal, como almacenamiento de las instrucciones pero también de datos u operandos que se utilizan en las instrucciones (pasos 3 y 5 mencionados arriba).

6.1.1. Operaciones sobre la memoria

La memoria admite dos operaciones: **lectura** y **escritura** de celdas. Para resolver la lectura, la memoria necesita conocer la dirección de una celda y recibir una señal de lectura. Luego de esto, pone a disposición de la Unidad de Control (UC) el contenido de la celda correspondiente y activa otra señal de control para indicar su finalización.

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

6.1.2. Interconexión

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

La información que se necesita transmitir incluye datos desde y hacia la memoria, así como direcciones hacia la memoria y señales de control. Es por esto que el bus de sistema está formado por tres buses para cumplir los diferentes objetivos y que reciben los nombres de **Bus de direcciones**, **Bus de datos** y **Bus de control** (ver figura 6.2).

Cada bus tiene una determinada cantidad de líneas -esto se denomina ancho del bus- y cada línea transmite un bit a la vez. Entonces 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 describir direcciones de la memoria, la cantidad máxima de combinaciones, y por lo tanto de celdas a direccionar, es 2^m .

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.

Bus del
Sistema

espacio
direccionable

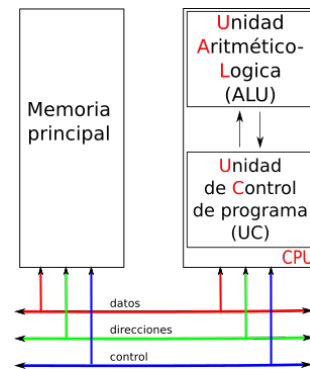


Figura 6.2: Buses del sistema

6.1.3. Modos de direccionamiento

Como se presentó en la sección al comienzo de este capítulo, las celdas de memoria pueden utilizarse como variables que el programador dispone desde las instrucciones, y entonces se necesitan otros mecanismos para hacer referencia a los operandos, es decir un modo de direccionamiento más, denominado modo de direccionamiento directo. En este modo de direccionamiento el campo de operando contiene la dirección efectiva del operando.

Esta técnica fue común en las primeras generaciones de computadoras y se encuentra aún en diversos sistemas y la limitación más evidente es que el tamaño de dicho campo restringe la cantidad de bits destinados a una dirección y por lo tanto el espacio direccionable. En la sección 6.2 se verá por qué también restringe el tamaño de las constantes.

6.2. Arquitectura Q2: alto nivel

La arquitectura Q2 agrega a la arquitectura Q1 un nuevo modo de direccionamiento: **Directo**, el cual especifica la dirección de memoria donde se encuentra el valor del operando.

Para usar este nuevo modo de direccionamiento la dirección del operando se escribe entre corchetes, 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.

Al momento del ensamblado de la instrucción el método de direccionamiento directo tiene la codificación 001000. De esta manera, el código máquina correspondiente a la instrucción: SUB [0x000A], 0x0001 es como se indica a continuación:

CodOp	modo destino	modo origen	destino	origen
0011	001000	000000	0000000000001010	0000000000000001
SUB	DIRECTO	INMEDIATO	000A	0001

En el cuadro 6.1 se muestran dos *mapas de memoria principal* (esto es: un segmento de la memoria) ilustrando el estado inicial (antes de ejecutar la

Estado inicial \Rightarrow	0x000A 0x000B	...	Estado final \Rightarrow	0x000A 0x000B	...
		29C8			29C7
		A0A0			A0A0
	

Cuadro 6.1: Mapas de memoria para SUB [0x000A], 0x0001

Estado inicial \Rightarrow	0x0004 0x0005	...	Estado final \Rightarrow	0x0004 0x0005	...
		0019			0019
		A0A0			A0B9
	

Cuadro 6.2: Mapas de memoria para ADD [0x0005], [0x0004]

instrucción) y el estado final luego de la ejecución de la instrucción analizada. Es importante notar que cuando se utiliza el modo de direccionamiento directo en el destino, el efecto se denota con la dirección de la celda entre corchetes. En particular en esta instrucción el efecto se denota: [000A] \leftarrow 29C7.

Como segundo ejemplo, considerar la instrucción: ADD [0x0005], [0x0004], cuyo efecto se describe en el cuadro 6.2.

6.3. Arquitectura Q2: bajo nivel

Para llevar a cabo estas tareas es claro que el procesador debe almacenar algunos datos temporalmente: debe recordar la posición de la última instrucción de forma de poder determinar de dónde tomar la siguiente, y necesita almacenar instrucciones y datos temporalmente mientras una instrucción está ejecutándose. En otras palabras, el procesador necesita una pequeña memoria interna.

Dentro del procesador hay un conjunto de registros clasificados en dos tipos:

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

Registros de
uso general

Registros
de uso
específico

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 por el usuario, pero alguno de ellos puede ser visible por ciertas instrucciones máquina ejecutadas en un modo privilegiado o de sistema operativo.

Naturalmente, diferentes máquinas tendrán distinta organización de registros y usarán distinta terminología, pero esencialmente se necesitan los siguientes registros para la ejecución de las instrucciones:

- **Contador de programa** (*Program Counter- PC*): Contiene la dirección de la instrucción a buscar.

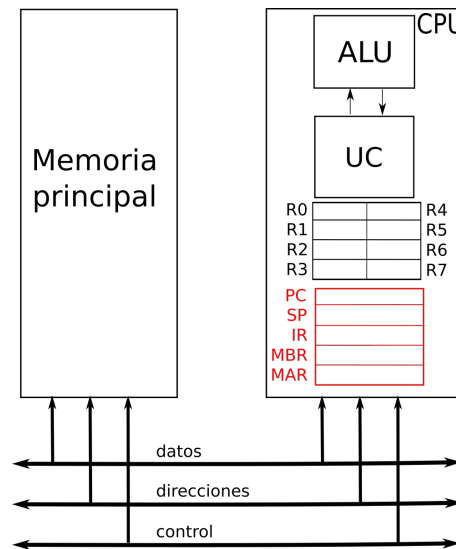


Figura 6.3: Registros de uso específico (en color rojo) y registros de uso general

- **Puntero de pila (*Stack Pointer*- SP):** Contiene la dirección del tope de pila (ver apartado 7.2)
- **Registro de instrucción (*Instruction Register*- IR):** Contiene la instrucción buscada mas recientemente. Muchas veces este registro tiene mas capacidad que los demás, a los fines de *almacenar la instrucción completa*.
- **Registro de dirección de memoria (*Memory Address Register*-MAR)** Contiene la dirección de una posición de memoria
- **Registro amortiguador de memoria (*Memory Buffer Register* - MBR)** Contiene el dato a escribir en una posición de memoria o el dato contenido en una posición de memoria leído mas recientemente

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 y se almacenen los bits leídos por el bus de datos. En la figura 6.3 se ven los registros de control 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 más adelante en el capítulo 9.

6.3.1. Funcionamiento de la memoria principal

Como se dijo en la sección 6.1.1, la memoria principal es un circuito formado por **celdas** que se acceden a través de dos operaciones: lectura y escritura. En esta sección se revisarán dichas operaciones a partir de los registros y los buses mencionados arriba.

La operación de **lectura** ocurre cuando la Unidad de Control (UC) pide un dato almacenado en una determinada celda de la memoria principal. Para esto, la UC pone la dirección en el registro **MAR** y luego activa una **señal de lectura** en el **bus de control**. Luego, la Memoria Principal activa un circuito decodificador (ver figura 6.5) que permite elegir la celda específica y copiar su contenido en el **bus de datos**. Finalmente, el dato se encontrará disponible en el registro **MBR** de la CPU y mediante el bus de control activará una señal que indica que el dato está disponible

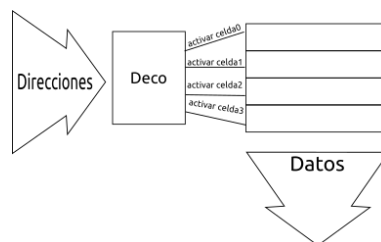


Figura 6.4: Memoria conectada a través de los buses

La operación de **escritura** ocurre cuando la UC necesita almacenar un dato en una determinada celda de memoria. Para hacerlo, la UC pone el dato en el registro **MBR** y una dirección en el registro **MAR**. Luego activa una **señal de escritura** en el bus de control para indicar la operación de escritura. También en este caso, la Memoria activa un circuito decodificador que permite elegir la

celda específica, pero además toma el dato del **bus de datos** y lo copia en dicha celda. Finalmente, mediante el bus de control se indica a la UC que la operación ha finalizado.

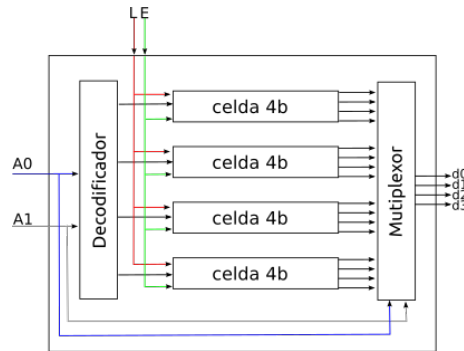


Figura 6.5: Circuito de una memoria de 4 celdas

En esta sección se revisará el ciclo de ejecución de instrucciones que recorre el procesador durante la ejecución de un programa, detallando los dispositivos digitales que entran en juego.

Típicamente, el procesador actualiza el **registro PC** luego de cada búsqueda de instrucción, de manera que siempre quede preparado para la siguiente instrucción a ejecutar. La instrucción buscada se carga en **registro IR**, donde son analizados las distintas partes de la instrucción, en particular el código de operación y los **modos de direccionamiento** de los operandos. A continuación se intercambian datos con la memoria por medio del MAR y el MBR, tal como se describió en la sección anterior.

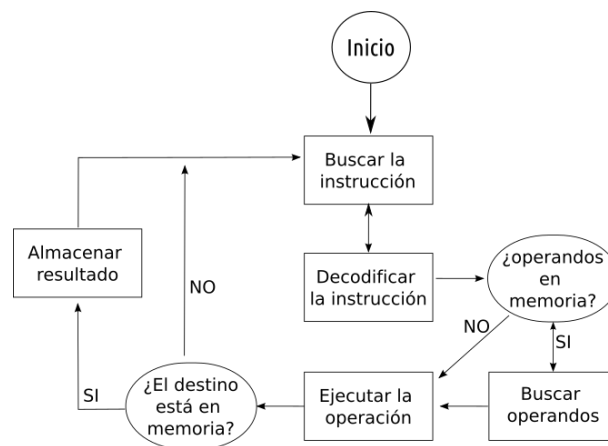


Figura 6.6: Ciclo de ejecución de instrucción

Los registros mencionados se usan para la transferencia de datos entre el procesador y la memoria principal. Dentro del procesador, los datos tienen que ofrecerse a la ALU para su procesamiento. La ALU puede tener acceso direc-

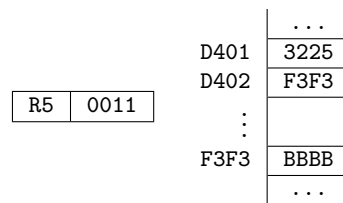


Figura 6.7: Estado inicial de la simulación de ejemplo

to al MBR y a los registros visibles por el programador, pero alternativamente puede haber registros intermedios en torno a la ALU, que le sirven como entrada y salida e intercambian datos con el MBR y los registros visibles al programador.

Luego de la ejecución de la instrucción, y en función del modo de direccionamiento indicado en la instrucción para el **operando destino**, se almacena el resultado. Esto puede resultar en una escritura a memoria (si el modo correspondiente es directo)

6.3.2. Accesos a memoria durante el ciclo de ejecución

En esta sección, se desarrollará el ciclo de ejecución de una instrucción, poniendo atención a las etapas del ciclo y a los accesos a la memoria. Sabiendo que la ejecución de un programa no es ni más ni menos que la ejecución en secuencia de sus instrucciones, entonces para llevar cuenta de la instrucción actual, o mejor dicho, la siguiente a ejecutar, se utiliza el registro *Program Counter*.

Se considerará como ejemplo una instrucción que está ensamblada a partir de la celda $0xD401m$, y por lo tanto $PC=D401$. El estado inicial de las celdas de memoria y el registro R5 es como se ilustra en la figura 6.7, y a continuación se analizan las etapas del ciclo de ejecución de instrucción (ver Figura 6.6 del ciclo de ejecución de instrucción).

1. **Búsqueda de la instrucción:** Se lee la celda indicada por el PC y se almacenan en **IR**. La lectura a memoria se describirá en término del estado de los buses, en el siguiente cuadro:

Etapas	Bus de ctrl	Bus de dir.	Bus de datos
Búsq. de instrucción	$R/\overline{W}=1$	D401	3225

Nota: en el bus de control se debe indicar $R/\overline{W}=1$ cuando es una lectura o $R/\overline{W}=0$ cuando es una escritura

Por lo tanto $IR=3225$. Además, se actualiza el PC para que quede preparado para la siguiente lectura $PC=D402$

2. **Decodificación de la instrucción:** A partir de la interpretación de la cadena en **IR**, la Unidad de Control entiende lo siguiente:

codop(4b)	M.D. (6b)	M.O. (6b)
0011	001000	100101

Dado que el modo de direccionamiento del operando destino es **directo**, la UC entiende que hace falta la lectura de otra celda

3. **Completar la búsqueda de la instrucción:** Se lee la celda indicada por el PC (D402) y se almacena también en **IR**. Por lo tanto **IR=3225F3F3**

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

4. **Completar la decodificación de la instrucción:** Con esta nueva lectura puede concluirse que:

codop	M.D.	M.O.	Dest (16b)
0011	001000	100101	1111001111110011

A los fines de la comprensión de nuestro ejemplo, es importante notar que dicho código máquina corresponde a la instrucción

SUB [0xF3F3], R5

5. **Búsqueda de operandos:** La Unidad de control solicita una lectura de la celda **F3F3** para obtener el operando destino. Para esto realiza los siguientes pasos:

- Copia la dirección **F3F3** del registro **IR** al registro **MAR**
- Indica a la Memoria Principal una lectura a través de las líneas correspondientes en el **Bus de control**.
- Cuando la Memoria Principal indica que el dato está disponible (también a través del bus de control), la UC tiene disponible el dato en el registro **MBR**.

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

6. **Ejecución de la instrucción:** La UC le indica a la **ALU** la operación **SUB** y le suministra los dos operandos:

- Operando destino: desde el registro **MBR** (BBBB)
- Operando origen: desde el registro **R5** (0011)

7. **Almacenamiento de resultado:** La Unidad de Control solicita una **escritura** en Memoria Principal para almacenar el resultado en el operando destino. Para esto realiza los pasos:

- Copia la dirección **F3F3** del registro **IR** al registro **MAR**
- Copia el resultado arrojado por la **ALU** en el registro **MBR**
- Indica a la Memoria Principal una **escritura** a través de las líneas correspondientes en el **Bus de control**.

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

8. Comienza un nuevo ciclo con la siguiente instrucción

Concluyendo, los accesos a memoria principal se pueden describir recopilando los cuadros del desarrollo anterior como se muestra a continuación:

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
Busq. de instrucción	$R/\overline{W}=1$	D401	3225
Busq. de instrucción	$R/\overline{W}=1$	D402	F3F3
Busq. de operandos	$R/\overline{W}=1$	F3F3	BBBB
Alm. de resultados	$R/\overline{W}=0$	F3F3	BBAA

Este cuadro será denominado *cuadro de accesos a memoria principal* y se utilizará para describir el contenido de los buses en cada operación de lectura o escritura a memoria al ejecutar todo el ciclo de la instrucción.

cuadro de
accesos
a memoria
principal

A continuación se desarrolla otro ejemplo para ilustrar la construcción del cuadro. Asumir que la instrucción `MOV R0, [0x9001]` está ensamblada a partir de la celda A000, y que se tiene el siguiente estado parcial de la memoria principal:

	...
9000	AB02
9001	9004
9002	0043
9003	BBBB
9004	0FFF
	...

Por lo tanto, los accesos provocados por el ciclo de ejecución de la mencionada instrucción se resumen en el siguiente cuadro. Notar que los dos primeros accesos recuperan el código máquina de las instrucciones y el último acceso el valor del operando.

Etapa	Bus de ctrl	Bus de dir.	Bus de datos
Busq. de instrucción	$R/\overline{W}=1$	A000	1808
Busq. de instrucción	$R/\overline{W}=1$	A001	9001
Busq. de operando	$R/\overline{W}=1$	9001	9004

Capítulo 7

Modularización y Reuso: Rutinas en Q3

7.1. Modularización y reuso

A menudo un problema complejo se transforma en algo mas abordable si se lo descompone en problemas más pequeños. Esta descomposición recibe el nombre de **modularización**.

Una rutina o subrutina (pues en general se la utiliza como parte de un programa más grande y específico) es un programa que se espera usar más de una vez. Muchas veces tenemos un problema complejo y necesitamos partirlo en problemas más simples. Para ello utilizamos subrutinas, las cuales nos permiten descomponer un programa grande en sub partes reusando la idea de solución de las mismas.

El uso de rutinas requiere llevar a cabo dos tareas, el **encapsulamiento** y la **invocación**. Para lo primero es indispensable determinar el alcance de la rutina (teniendo en cuenta que debe ser acotada). Una vez hecho esto se crea el código que cumpla con la función de la rutina y se delimita su comienzo y su fin, para lo cual se utiliza una **etiqueta** que la identifique inequívocamente, ubicada en la primera instrucción para que la rutina pueda ser invocada por su nombre, y una instrucción especial **RET** al final para que al finalizar su ejecución vuelva al programa que la invocó. Cabe destacar que no necesariamente la instrucción **RET** estará sólo al final de la rutina, también puede encontrarse en el cuerpo de la misma.

Considerar como ejemplo la siguiente rutina **agregar**, que incrementa en uno el valor de R0.

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

La rutina esta delimitada en su inicio por la etiqueta **agregar** por lo cual para su invocación desde el *programa principal o programa cliente* se utiliza la instrucción especial **CALL** haciendo referencia a la misma.

Siguiendo el ejemplo anterior, la siguiente es una invocación a la rutina **agregar**.

```
1 CALL agregar
```

La etiqueta **agregar** se traducirá en un **inmediato** cuyo valor será la dirección de memoria a partir de la cual está ensamblada la rutina, por lo cual la primera instrucción de la misma será la siguiente a ejecutar.

Si la rutina **agregar** se encuentra ensamblada a partir de la celda de memoria 0x568A la invocación anterior equivale a la siguiente:

```
1 CALL 0x568A
```

Por otro lado, si se utilizara esta rutina para incrementar en 3 en vez de en uno el valor de R0, es posible utilizarla varias veces, provocando el **reuso de la subrutina**. En ese caso se modifica el programa principal que la invoca. Por ejemplo:

```
1 CALL agregar
2 CALL agregar
3 CALL agregar
```

7.1.1. Haciendo rutinas flexibles: pasaje de parámetros

Considerar la siguiente rutina **avg** (*average* = promedio), que calcula el promedio entre los valores 20 y 30 (0x0014 y 0x001E en hexadecimal respectivamente):

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

El programa cliente que utilice esta rutina podría ser:

```
1 CALL avg
```

En este ejemplo se ve claramente una posible mejora para hacer: que la rutina pueda calcular el promedio entre *cualquier par de valores*. Para lograr esto, la rutina debe ser **parametrizable** ya que no conocemos los valores involucrados. Haciendo una analogía con las matemáticas, lo que se intenta hacer es pasar de una expresión

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

a esta otra:

$$avg = \frac{a + b}{2}$$

siendo a y b dos variables.

A diferencia de otros lenguajes de programación, Q3 no cuenta con una sintaxis específica para el *pasaje de parámetros a las rutinas*. Pero es posible que la rutina se alimente de esos valores de entrada para el cálculo del promedio a través del uso de variables, que en esta arquitectura serían registros o celdas de la memoria principal.

Con esta idea, la rutina puede mejorarse de la siguiente manera:

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

pasaje de
parámetros a
las rutinas

De esta manera, la rutina `avg` ahora calcula el promedio entre dos valores cualesquiera que estén almacenados en `R1` y `R2`. Es importante notar que por esta razón el programa que la usa también debe modificarse para colocar los valores respectivos para calcular el promedio de `R1` y `R2`:

```
1 MOV R1, 0x0014
2 MOV R2, 0x001E
3 CALL avg
```

7.1.2. Documentación de las rutinas

Para usar apropiadamente cualquier rutina es importante contar con información sobre qué hace, cómo pasarle los datos que necesita (parámetros), qué modifica del 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 tiene tres piezas de información:

- **Requiere:** se utiliza para describir los parámetros de entrada. Qué registro o celda contiene cada uno y qué naturaleza tienen.
- **Modifica:** se utiliza para describir qué variables se modifican previniendo posibles *efectos colaterales* de la ejecución de la rutina.
- **Retorna:** se utiliza para indicar en qué variable (registro o celda) se retorna el/los resultado/s y en qué consiste el mismo.

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

Por ejemplo, la documentación que acompañe la rutina `avg` mencionada arriba podría ser:

avg	
Requiere	dos valores a promediar en los registros R1 y R2 y que la suma de ambos se pueda representar en 16 bits
Modifica	R1
Retorna	en R0 el promedio (entero) entre R1 y R2

Es importante notar en la documentación anterior que el dato colocado en `R1` se perderá.

La documentación de una rutina es un **contrato** entre el autor de la rutina y los usuarios de la misma. Esto implica que la persona autora de la rutina se compromete a: si las especificaciones escritas en la documentación son respetadas a la hora de invocar la rutina (respetar el 'requiere'), ésta funcionará de manera correcta con los resultados esperados (respetando el 'modifica' y el 'retorna').

¿Cuál es la relación entre los campos *modifica* y *retorna*?

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.

7.2. Cómo funcionan CALL y RET

El objetivo de la programación mediante rutinas es el de abstraer al programador/a de la ubicación de dichas rutinas en memoria principal a través del encapsulamiento. La figura 7.1 describe gráficamente un ejemplo donde una rutina principal se apoya en una rutina de nombre `rutinaA`, que a su vez se apoya en la rutina `rutinaB`.

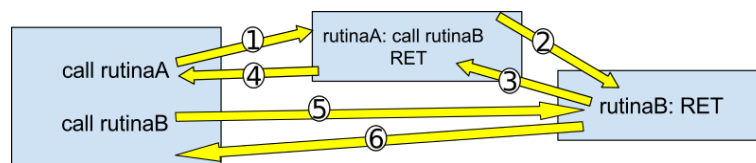


Figura 7.1: flujo de ejecución entre 3 rutinas de ejemplo

En ese ejemplo se destacan varias discusiones importantes. En primer lugar, cuando una rutina llama otra subrutina, provocando que se *aniden* varios llamados; en segundo lugar, que 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 una oportunidad la rutina `rutinaB` es invocada desde la rutina `rutinaA` y en la otra desde el programa principal por lo que vemos que el punto de retorno varía.

Invocaciones
anidadas

Como se ha presentado en capítulos anteriores, las rutinas son ensambladas y ubicadas en memoria por el ensamblador, quien no necesariamente lo hace en ubicaciones contiguas de memoria. Siguiendo el ejemplo anterior, una posibilidad sería:

	...
(programa ppal)0x9999	<call>
0x999A	<rutinaA>
0x999B	<call>
0x999C	<rutinaB>
	:
(rutina A)AAAA	<call>
0xAAAB	<rutinaB>
0xAAAC	<ret>
	:
(rutina B)0xBAA0	<ret>
	...

Para llevar control de cuál es la instrucción que se debe ejecutar, se utiliza el registro *PC*. Este registro se incrementa al finalizar la etapa de *búsqueda de instrucción* para representar en todo momento la dirección de la primer celda

de la siguiente instrucción a ejecutar.

A diferencia de las instrucciones aritméticas en las que el PC no se modifica hasta que comience un ciclo nuevo, la instrucción **CALL** requiere alterar el valor del PC para poder causar el efecto de *desvío de flujo* necesario. Considerar la ejecución del programa del ejemplo:

desvío de
flujo

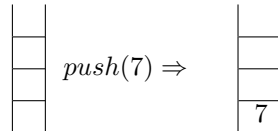
1. PC=9999
2. Se busca la instrucción **CALL rutinaA** (teniendo en cuenta que la instrucción ocupa dos celdas de la memoria). Por lo cual el nuevo valor de PC es 0x999B, ya debe quedar preparado para ejecutar la siguiente instrucción del programa principal.
3. Se realiza la ejecución de **CALL rutinaA**, que principalmente prepara PC para que se desvíe el flujo a la rutina **rutinaA**, por lo cual su nuevo valor es 0xAAAA. Pero además se debe **recordar el valor anterior de PC** (0x999B) para restituir el flujo al retornar de la rutina.
4. Se busca la instrucción **CALL rutinaB** (ensamblada en la dirección de memoria 0xAAAA). Entonces el valor de pc es ahora 0xAAAC, pues debe quedar preparado para la siguiente instrucción.
5. Se realiza la ejecución de **CALL rutinaB**, entonces PC=BAA0. Nuevamente se debe **recordar el valor de PC** (AAAC).
6. Se busca la instrucción **RET** (ensamblada en BAA0). Entonces PC=BAA1, pues debe quedar preparado para la siguiente instrucción.
7. Se realiza la ejecución de **RET**, es decir **se restituye el último valor de PC: PC=AAAC**
8. Se busca la instrucción **RET** (ensamblada en AAAC). Entonces PC=AAAD, pues debe quedar preparado para la siguiente instrucción.
9. Se realiza la ejecución de **RET**, es decir **se restituye el anterior valor de PC: PC=999B**
10. Se busca la instrucción **CALL rutinaB** (ensamblada en 999B). Entonces PC=999D, pues debe quedar preparado para la siguiente instrucción.
11. Se realiza la ejecución de **CALL rutinaB**, entonces PC=BAA0
12. Se busca la instrucción **RET** (ensamblada en BAA0). Entonces PC=BAA1, pues debe quedar preparado para la siguiente instrucción.
13. Se realiza la ejecución de **RET**, es decir **se restituye el último valor de PC: PC=999D**

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

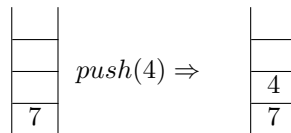
7.2.1. La estructura de pila

Los cambios en el PC producidos por la ejecución de CALL dan lugar a la necesidad de contar con una estructura de datos que permita registrar los diferentes valores de retorno y en el orden necesario para ser consumidos de manera inversa a la que fueron registrados. Esta estructura se denomina **pila** y está caracterizada por dos operaciones: apilado (*push*) y desapilado (*pop*).

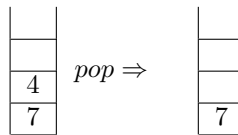
Suponer una estructura de pila vacía, donde se apila el valor 7:



Si luego se apila el valor 4:



Si a continuación se desapila:



El registro que se usará como **tope de pila** se denomina **SP** (*Stack pointer*, puntero de pila), y su valor inicial, que se corresponde con la situación donde la pila está vacía, es FFEF. Con esta idea, se debe revisar el funcionamiento esperado para el CALL y el RET, 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 PC anterior y luego reemplazarlo por la dirección de inicio de la rutina. Particularmente, se realizan los siguientes pasos:

1. Se copia PC a la celda indicada por **SP**
2. Se decrementa (hay un elemento más en la pila) **SP**
3. Se actualiza PC

Efecto para
CALL

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

1. Se incrementa (hay un elemento menos en la pila) **SP**
2. Se copia a PC el valor de celda indicada por **SP**

Efecto para
RET

En la Figura 7.2 se explican gráficamente los cambios de la pila que ocurren por la ejecución del ejemplo des programa con dos subrutinas (*rutinaA* y *rutinaB*) de la sección anterior.

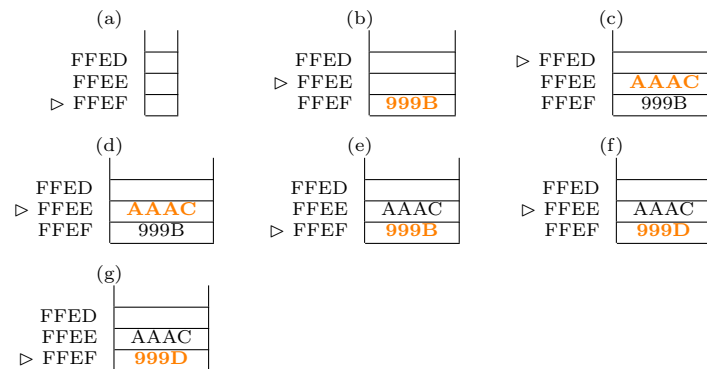


Figura 7.2: (a)Pila vacía (b) Luego de ejecutar **CALL rutinaA** (c) Luego de ejecutar **CALL rutinaB** (d) Luego de ejecutar **RET** en **rutinaB** (e) Luego de ejecutar **RET** en **rutinaA** (f) Luego de ejecutar **CALL rutinaB** (g) Luego de ejecutar **RET** en **rutinaB**

7.3. 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 6.3.2) se realizan cambios en los registros de uso específico PC, IR y SP, así como en la pila, que dependerán de las instrucciones que lo conformen.

Por ejemplo, suponer la ejecución del siguiente bloque de código que está ensamblado a partir de la celda 0000:

```
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 0123, 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 es 0000 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 que describe el ciclo de ejecución de cada instrucción:

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).

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	1800 4567 0008	0003	MOV	Dir	Inm	[4567]	0008	{ }	FFEF	0003	[4567] <- 0008
0003	1800 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	--

Cuadro 7.1: Cuadro de simulación de la rutina

El primer ciclo de la simulación considera:

- **Búsqueda de la instrucción** PC comienza con el valor 0000. IR contiene el código máquina de la instrucción leída. 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 180045670008. Inmediatamente después de la búsqueda de instrucción, el PC contiene la dirección de memoria donde comienza la siguiente instrucción a ejecutar. Entonces el valor del registro PC luego de la búsqueda de instrucción es 0003, pues como se puede observar en el registro IR, la instrucción ocupa 3 celdas.
- **Decodificación de la instrucción** Se identifica la operación a realizar y sobre qué operandos. En este caso es un MOV, que se trata de una instrucción de dos operandos.
- **Operandos** El operando destino es la celda 4567, y el operando origen de este ejemplo 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.
- **Almacenamiento de resultados** 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 el cuadro 7.1. 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 se agregan cuadro aplicando 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 la rutina **rutinaA** esta ensamblada a partir de la celda 0123, y la rutina **rutinaB** esta ensamblada a partir de la celda 0678. Por último, considerando que el valor inicial del registro PC es 0008 y que la pila está vacía, entonces la primer entrada de la tabla se completa como sigue:

- **Búsqueda de la instrucción** PC comienza con el valor 0008. IR contiene el código máquina de la instrucción leída: (CALL rutinaA), que ocupa 32 bits (2 celdas), y el código máquina correspondiente es B0000123. Luego de la búsqueda de instrucción, el PC toma el valor 000A.
- **Decodificación de la instrucción** Se identifica la operación a realizar y sobre qué operandos. En este caso es un CALL, que se trata de una instrucción de un solo operando origen, dado que el operando destino es el registro PC y por lo tanto está implícito (la instrucción no debe indicarlo).
- **Operandos** El operando origen de este ejemplo es el nuevo valor que tendrá PC: 0123, y se deduce del valor de la etiqueta rutinaA.
- **Ejecución de la operación** Como indica el **efecto de la instrucción CALL**, es decir: [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
 2. Decrementar SP: El valor del registro SP se actualiza para que se apunte a un lugar libre, es decir que toma el valor FFEE.
 3. Actualizar PC: Finalmente, valor de PC luego de la ejecución se carga con el valor Origen, y como se trata de una etiqueta (valor inmediato), dicho valor se leyó con el código máquina de la instrucción.
- **Almacenamiento de resultados** Dado que no se tiene un operando destino (o este está implícito y es el registro PC), entonces esta etapa no se lleva a cabo.

Estos datos se escriben en la primer fila de la tabla como se muestra en la primer fila del cuadro 7.2. 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 PC y la pila. 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 tercer instrucción (RET), se restituye el flujo a la última rutina que ejecutó un RET, en este caso se trata de rutinaA y para esto se desapila el tope de la pila y se lo carga 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 las siguientes instrucciones, en el orden del **flujo de ejecución** incluyendo el llamado a subrutinas (y las instrucciones de las mismas) hasta la última instrucción del programa principal. La tabla terminada con la simulación de ejecución del programa anterior puede verse en el cuadro 7.2.

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	--

Cuadro 7.2: Simulación completa de la rutina principal

Capítulo 8

Sistemas de numeración para números enteros

En apuntes anteriores vimos como utilizar binario para representar números naturales. En este capítulo veremos como trabajar con números enteros. En decimal, solemos utilizar el signo “-” para indicar que un número es negativo, pero este no es representable en las computadoras como tal, donde solo se cuenta con 0s y 1s. Veremos entonces, como salvar esto de distintas maneras, cada una con sus particularidades.

8.1. 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 primer bit de una cadena (aquel del extremo izquierdo) como indicador y se lo denomina **bit de signo**. Si el bit de signo es un 1 se trata de un número negativo, y en caso contrario 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 primer bit es el signo, y la magnitud es de $n - 1$ bits.

Interpretación de cadenas en SM(n)

Para interpretar una cadena en Signo Magnitud se utiliza la interpretación del sistema Binario Sin Signo 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 mas arriba, en la sección anterior: El bit 1 indica un valor negativo, y 0 un valor positivo. En el ejemplo, la cadena comienza con 1, de modo que se trata de un negativo.

La magnitud, se interpreta como BSS: $I_{BSS}(010) = 0*2^0 + 1*2^1 + 0*2^2 = 2$

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

$$\begin{aligned} I_{sm(4)}(1010) &= -1 \times I_{BSS}(010) = \\ &= -1 \times (0 * 2^0 + 1 * 2^1 + 0 * 2^2) = -1 \times 2 \end{aligned}$$

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, pero este último permite representar sólo números positivos, por lo que es necesario **tomar el valor absoluto del valor dado para representar**.

Suponiendo como ejemplo que se necesita representar el valor -5 en SM(4). Este sistema destina 1 bit para el signo y 3 bits para la magnitud.

Lo primero a resolver es definir el signo, y como en este caso el número es negativo el valor del bit de signo que va a tener la cadena resultante es **1**. Luego 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:

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

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, va a ser negativo, es decir va a comenzar con 1. Su magnitud va a ser la mas grande posible. De esta manera, para n bits, el mínimo va a ser:

$$\begin{aligned} &\underbrace{1}_{\text{signo}} \underbrace{1\dots,1}_{n-1\text{magnitud}} \\ &-(2^0 + \dots + 2^{n-2}) \\ &-(2^{n-1} - 1) \end{aligned}$$

El numero máximo se obtendrá utilizando signo positivo, es decir 0; y nuevamente la mayor magnitud.

$$\begin{aligned} &\underbrace{0}_{\text{signo}} \underbrace{1\dots,1}_{n-1\text{magnitud}} \\ &2^0 + \dots + 2^{n-2} \\ &2^{n-1} - 1 \end{aligned}$$

Por lo tanto, que en este caso, el rango es $[-(2^{n-1} - 1), 2^{n-1} - 1]$.

Si, por ejemplo, se restringe el sistema a 8 bits, el rango para el sistema SM(8) tendría, como mínimo la cadena 11111111, y como máximo la cadena 01111111

Es interesante notar que en dicho intervalo no hay 2^n números distintos. Por ejemplo, si $n = 3$, el rango del sistema SM es:

$$[-(2^{3-1} - 1), 2^{3-1} - 1] = [-3, 3]$$

y en dicho intervalo hay 7 números: $\{-3, -2, -1, 0, 1, 2, 3\}$. En binario sin signo, con 3 bits se tenían 8 números representables diferentes. ¿Dónde está el número que falta? El número que falta dentro del rango se genera a causa de que el sistema posee una *doble representación del 0*, ya que tanto la cadena 000 como la cadena 100 representan dicho valor.

doble
representación
del 0

Esto trae consigo dos desventajas: la primera es el hecho de desaprovechar una cadena, y la segunda es que esta doble representación complica la aritmética (y los circuitos que la implementan) 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 equilibrado**. 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 $[-3, -2, -1]$

Aritmética

Suma

La suma en *SM* considera 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 como vimos para *BSS* y tomando como signo el signo del resultado.

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 $101 + 001 = 110$. Por lo tanto la cadena resultado de la suma en *SM* es 1110 .

Si las cadenas a sumar tienen diferente signo, se debe primero identificar qué cadena tiene la mayor magnitud (sea A la cadena de mayor magnitud y B la de menor magnitud). El signo del resultado va a ser el signo que tenga A, y la magnitud resultado se obtiene restando la magnitud de B a la magnitud de A.

Considerar el siguiente ejemplo: $1101 + 0001$. La mayor magnitud es la de la cadena 1101 , ya que su magnitud es 101 (5) y es mayor que la magnitud 001 (1). Es por esta razón que ya podemos afirmar que el signo del resultado va a ser 1. Tenemos ahora que restar las magnitudes (a la mayor magnitud, la menor). Esta resta auxiliar la tenemos que hacer en *BSS* y nos queda: $101 - 001 = 100$. Entonces el resultado final es 1100

Resta

El algoritmo para calcular la resta entre las cadenas C_1 y C_2 , puede pensarse como una suma, aplicando la siguiente equivalencia:

$$C_1 - C_2 = C_1 + (-C_2)$$

En el sistema SM, es posible construir el inverso de la cadena C_2 ($-C_2$) a partir de invertir el bit de signo sobre C_2 .

Luego de ese paso, la operación se traduce en una suma y se deben analizar los casos que se describieron en el apartado anterior.

Si se considera, por ejemplo, la resta $1101 - 1001$, entonces como primer paso se la traduce en la suma $1101 + 0001$ y dicha suma se resuelve considerando los signos adecuadamente, según el algoritmo de la sección anterior.

8.2. Complemento a 2

La idea de este sistema es lograr tener números negativos pero manteniendo la aritmética del sistema *BSS*, que se implementa con los circuitos aritméticos presentados en la sección 5.4.1. Esto es muy deseable porque permite que una maquina no necesite dentro de su *ALU* diferentes circuitos para sumar números naturales y enteros; y esto se traduce en abaratar costos.

Suponer que se tiene un solo dígito en el sistema decimal (denotado $dec(1)$) y se tiene que poder representar negativos (sin usar el signo “-”). Para resolverlo, se ubica las **cadenas** del sistema $dec(1)$ en el exterior de una rueda como se indica en la figura 8.1 (con color blanco), y se le asignan **valores** en la parte interna de la rueda (en color amarillo) de manera que cada cadena (en este caso, dígito) quede aparejada con el opuesto a su complemento a la base. En este escenario, el complemento a la base es el valor 10. Así, el 9 queda asociado al -1 pues:

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

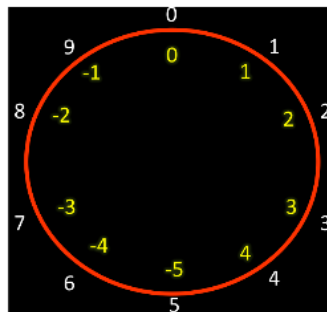


Figura 8.1: Distribución de las cadenas del sistema Dec(1)

Este aparejamiento 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 en el sistema decimal restringido a un dígito, $8+2 = 0$ (con acarreo de 1). Entonces, si se tiene en cuenta la interpretación de cada cadena:

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

Considerar otro ejemplo: $7+4=1$ (con acarreo 1) entonces

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

Esta idea se puede llevar al sistema binario, para lo cual se ubican las cadenas en un círculo ordenadas lexicográficamente, y luego asignan números positivos en sentido horario y negativos en sentido anti-horario, como se muestra en la figura 8.2. De esta manera se consigue que algunas cadenas (las mas pequeñas en orden lexicográfico) queden asociadas a valores positivos y otras a valores negativos (las mayores en orden lexicográfico). Este sistema es el *Complemento a 2*, y se denota $CA2(n)$.

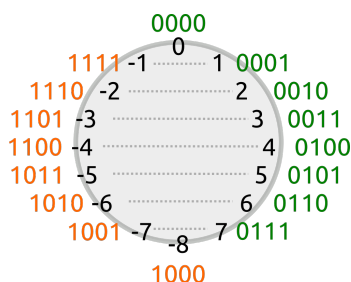


Figura 8.2: Distribución de las cadenas de 4 bits sobre los valores enteros en un sistema $CA2(4)$

En este sistema se utilizarán las cadenas que empiezan en 0 para los números positivos y las cadenas que comiencen en 1 para los negativos. Es importante notar que no se utilizará ese bit para signo como es el caso del sistema signo-magnitud, pero si ocurre que el bit da información a la hora de interpretar.

Adicionalmente, se necesita definir la operación *complemento()* sobre una cadena. El complemento de una cadena ω es otra cadena ω' que se obtiene a partir de invertir los bits de ω y sumarle 1 al resultado. Por ejemplo, el complemento a 2 de la cadena 0001 es 1111, y se denota:

$$\text{complemento}(0001) = 1111$$

ya que en primer lugar se invierten los bits obteniendo 1110 y luego se suma 1. La operación *complemento* cumple con la siguiente propiedad:

$$\omega + \omega' = 0_{n-1} \dots 0_0$$

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 interesante porque justifica la asociación entre cadenas y valores como se ve en la Figura 8.2. En particular:

$$I(\omega) + I(\omega') = I(0000) = 0$$

Representación

A la hora de representar se tiene en cuenta si el número es positivo (o cero), o si es negativo. Los números positivos se representan en binario sin signo y los números negativos se representan con el **complemento a dos de la cadena** que representa su **valor absoluto**.

Por ejemplo, en CA2(4), la representación del 3 es 0011, es decir, igual que en BSS. Para representar -3 en CA2(4), primero se debe representar el valor 3 en binario sin signo (es decir 0011) y a continuación se toma el complemento a 2 de dicha cadena $0011 \rightarrow 1100 + 1 \rightarrow 1101$. Entonces, la representación en el sistema Complemento a 2 de un número negativo se puede formalizar como:

$$\text{complemento}(R(|x|))$$

Interpretación

Al momento de interpretar se recorre el camino inverso a la representación, por lo tanto también se debe determinar si la cadena comienza con 0 ($b_{n-1} = 0$) o con 1 ($b_{n-1} = 1$). Si $b_{n-1} = 0$, 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, luego se interpreta el resultado en BSS(n) y finalmente se le agrega el signo negativo.

Por ejemplo, la cadena 1111 comienza con 1, por lo que representa un número negativo.

1. Calculamos su complemento a dos: $\text{complemento}(1111) = 0000 + 1 = 0001$.
2. Interpretamos la cadena resultante 0001 en BSS, lo cual nos da 1
3. Ahora, como sabemos que se trataba de un negativo, agregamos el signo, siendo el resultado final -1.

Entonces, la interpretación en Complemento a 2 de un número negativo se puede formalizar como:

$$-I_{BSS}(\text{complemento}(x))$$

Rango

Para calcular el rango, vamos a obtener las cadenas que nos dan el máximo y el mínimo. Para el primer caso, sabemos que necesitamos una cadena positiva y en particular a la que nos de el número mas grande. Dado que las cadenas positivas se comportan como binario sin signo, si trabajamos con complemento a 2 de n bits, la cadena que nos da al máximo es 011...11 (un 0 seguido de todos unos). Dicha cadena, si la interpretamos nos da:

$$2^0 + \dots + 2^{n-2}$$

$$2^{n-1} - 1$$

Para el caso negativo, como al momento de interpretarlo vamos a aplicar la operación de complementar, lo que queremos es buscar al número cuyo complemento sea lo mas grande posible. De todas las cadenas negativas, esto lo vamos a obtener con 100...00 (un 1 seguido de todos ceros). Si complementamos

$$100 \dots 00$$

$$011 \dots 11 + 1$$



$$100 \dots 00$$

Que luego al interpretar (ahora usando binario sin signo, porque ya complementamos) nos da:

$$2^{n-1}$$

Recordamos que era negativa, por lo que el resultado es:

$$-2^{n-1}$$

Finalmente, el rango es:

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

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 circuitos de suma (*Full adder* y *restador*). 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 los operandos valen:

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

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

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:

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

8.3. 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 desplazarse 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.

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 las figuras 8.3 y 8.4 es posible comparar el sistema Excedido en 4 de 3 bits con el sistema BSS(3) y observar el espíritu de los sistemas excedidos.

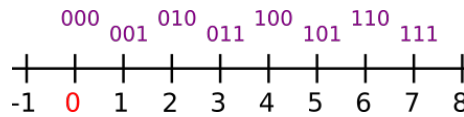


Figura 8.3: Sistema BSS(3)

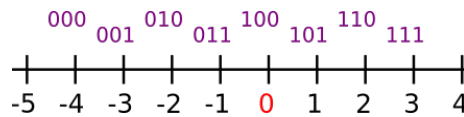


Figura 8.4: Sistema Exc(3,4)

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 figuras 8.3 y 8.4, 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, se intentará reutilizar los criterios válidos en binario sin signo para así también economizar en el diseño de la ALU. Es decir que si la aritmética del sistema Excedido se puede resolver con el sumador y restador definidos antes para binario sin signo, entonces la ALU como se la conocía *soportará nativamente este nuevo sistema*.

Si se analiza qué ocurre al sumar dos cadenas excedidas con un circuito *full adder* se encuentra que el resultado estará demasiado excedido (o desplazado en el sentido que corresponda a Δ) y por lo tanto se necesita **anular un desplazamiento**.

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:

$$\begin{array}{r} 1001 \\ + 0011 \\ \hline 1100 \end{array}$$

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, o tiene un Δ adicional. Para solucionarlo, hace falta ajustar su desplazamiento, restando el Δ a la cadena que se obtuvo:

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

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):

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



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 :

$$\begin{array}{r} 00001001 \\ + \quad 00100000 \\ \hline 00101001 \end{array}$$

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 9

Estructura condicional. Q4

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 rutinas, los programas 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 **verificación por igualdad** entre dos valores de dos variables o entre una variable y una constante. Por ejemplo, suponer la siguiente condición, expresada en un lenguaje de pseudo-código:

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

Problema
de ejemplo:
pseudocódigo

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

```
1 {Comparar el contenido de R0 con 1}
2 {si es diferente saltar hacia la etiqueta 'abajo'}
3 MOV R6, 0x000A
4 abajo: fin
```

Para esto, la arquitectura Q propone una nueva versión, Q4, que incluye **instrucciones de salto** y una instrucción de **comparación**. Los **Saltos** son instrucciones que permiten **desviar el flujo**, esto es, modificar el valor del PC logrando que el programa 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.

Los saltos disponibles se clasifican en condicionales e incondicionales.

- **Saltos Condicionales:** el desvío del flujo (actualización del PC) se lleva

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

Cuadro 9.1: Saltos condicionales

a cabo durante de la ejecución del salto, **solo si se cumple una determinada condición**

- **Saltos Incondicionales:** el desvío del flujo (actualización del PC) se lleva a cabo siempre que se ejecute el salto.

Las instrucciones de Salto Condicional 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**. La intención es modificar el PC si la condición del salto se cumple, de lo contrario no tiene efecto alguno.

La instrucción de salto incondicional es **JMP** y tiene la sintaxis: **JMP etiqueta** y al ejecutarse carga el PC con la dirección de memoria donde está **etiqueta**.

Estas herramientas permiten detallar un poco mas el problema que se intentaba resolver aplicando el salto JE y el salto JMP, como sigue:

```

1          {Comparar el contenido de R0 con 1}
2          JE saltarParaAsignarA
3          JMP salir
4 saltarParaAsignarA: MOV R6, 0x000A
5          salir: RET

```

Por último se necesita una instrucción que permita comparar dos valores de 16 bits para concluir, o no, si son iguales, y es posible expresar esto en términos de una resta:

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

y entonces el problema se reduce a indicar si el resultado es cero.

El **CMP** es la instrucción que se incluye entre las instrucciones de dos operandos cuyo objetivo es comparar dos operandos por medio de una resta, antes de usar un salto condicional. Dicha 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).

Finalmente el problema que se estaba analizando se resuelve mediante la siguiente rutina:

```

1      rutina:  CMP R0, 0x0001
2              JE saltarParaAsignarA
3              JMP salir
4 saltarParaAsignarA: MOV R6, 0x000A
5              salir: RET

```

9.2. Funcionamiento de los saltos

La instrucción de salto incondicional **JMP** funciona similar a un **CALL** dado que tiene la forma **JMP etiqueta** y al ejecutarse carga el PC con la dirección de memoria donde está **etiqueta**, pero a diferencia del **CALL** no modifica la pila.

Por otro lado, los saltos pueden modificar el registro PC de dos maneras:

- **Salto Absoluto:** el nuevo valor del PC se expresa en términos de una dirección de memoria
- **Salto Relativo:** el nuevo valor del PC se expresa en términos de un desplazamiento con respecto a la siguiente instrucción

Ensamblado de CMP

Para ensamblar la instrucción **CMP** es importante notar que es una instrucción de dos operandos, como las instrucciones aritméticas que se conocían en las versiones anteriores de la arquitectura **Q**.

Operación	Cod Op	Efecto
CMP	0110	Dest - Origen

Entonces la instrucción **CMP** se ensambla siguiendo el formato de instrucción descripto a continuación:

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

Por ejemplo, el código máquina de la instrucción **CMP R0, 0x4567** es 0110 100000 000000 0100 0101 0110 0111 y esto comprimido a hexadecimal es: 6800 4567

Ensamblado del salto incondicional

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

Operación	Cod Op	Efecto
JMP	1010	PC ← Origen

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 siguiente instrucción (a donde se quiere bifurcar), sino que se calcula dónde se tiene que saltar utilizando un desplazamiento. Es decir que al valor del PC se 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.

1111	CodOp (4)	Desplazamiento(8)
------	-----------	-------------------

Los primeros cuatro bits del formato de instrucción se corresponden con la cadena 1111₂, funcionando como prefijo del **CodOp**. Si **al evaluar la condición de salto** (ver Cuadro 9.2) esta se cumple, se le suma al PC el valor del **desplazamiento**, representado en $CA2(8)$. En caso contrario la instrucción no altera PC.

Entonces el ciclo de ejecución para los flags condicionales puede detallarse como se indica en la Figura 9.1.

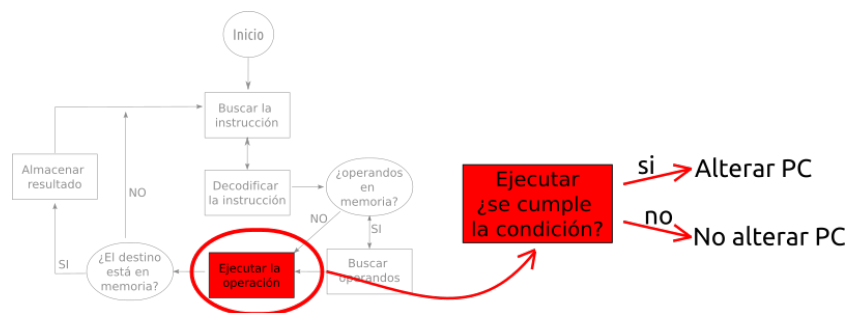


Figura 9.1: Ciclo de ejecución de instrucción

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 del programa (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 del programa (o anterior en la memoria).

Por ejemplo, suponer el siguiente código ensamblado a partir de la celda 0x00AA

```

1 rutina:  CMP R0, 0x0001
2           JE abajo
3           MOV R5, R6
4 abajo:  RET

```

Para obtener el código máquina de la instrucción JE abajo ver la tabla 9.2 de códigos de operación. Entonces, el código máquina es como sigue: 1111 0001 0000 0001 y con esto se indica un desplazamiento de 1, pues $I_{ca2}(00000001) = I_{bss}(00000001) = 1$.

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

Cuadro 9.2: Código de operación y condiciones de cada salto condicional

Suponer un segundo ejemplo, también ensamblado a partir de la celda 0x00AA orga

```

1 rutina: JMP seguir ---> 00AA y 00AB
2 arriba: JMP fin ---> 00AC y 00AD
3 seguir: CMP R1,R0 ---> 00AE
4         JE arriba ---> 00AF
5         MOV R5,R6 ---> 00B0
6 fin:    RET ---> 00B1

```

El código máquina de la instrucción **JE arriba** es como sigue: 1111 0001 1111 1100 pues se intenta indicar un desplazamiento de -4 y

$$\begin{aligned}
 R_{ca2(8)}(-4) &= \text{complemento}(R_{bs(8)}(4)) = \text{complemento}(00000100) = \\
 &= 11111011 + 1 = 11111100
 \end{aligned}$$

Al ejecutarse la instrucción **JE arriba** (en particular la etapa de ejecución) el PC está preparado para el **MOV R5, R6**, es decir, en 00B0. Si la intención es restarle un desplazamiento **w** tal que luego quede en el valor 00AC (donde está la instrucción etiquetada como “arriba”) entonces debe ocurrir que $00B0 + w = 00AC$. Con esta idea se debe comprobar si lo obtenido antes como desplazamiento (la cadena que representa al valor -4) es correcto:

$$\begin{array}{r}
 0000\ 0000\ 1011\ 0000 \\
 +\ 1111\ 1111\ 1111\ 1100 \\
 \hline
 0000\ 0000\ 1010\ 1100
 \end{array}$$

Notese que la cadena obtenida 0000 0000 1010 1100 se comprime en hexadecimal con la cadena 00AC.

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, o no, si son iguales, es posible reformular el problema en términos de una resta: $A = B \iff A - B = 0$ y entonces el problema se reduce a indicar si el resultado es cero.

Con la misma idea de restar los valores, se los puede comparar por mayor (o menor) como sigue:

$$A < B \iff A - B < 0$$

y este otro problema se traslada a determinar si el resultado es negativo. En un sistema en complemento a 2 esto puede resolverse con una señal que se encienda cuando el primer bit del resultado de operar $A - B$ es 1. Sin embargo, el primer bit no indica nada desde el punto de vista de 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 prestamo si es una resta (en inglés *borrow*) que la ALU genera como parte del resultado de la resta.

Entonces, cuando la ALU ejecuta una instrucción aritmética, además de realizar la operación y generar un resultado, calcula un **conjunto de indicadores que caracterizan ese resultado**. Estos indicadores se denominan *flags* o banderas y son bits que se almacenan conjuntamente en un registro de la CPU.

En el caso de la arquitectura Q se tienen 4 flags y cada uno indica una situación distinta:

Flag Z (Zero)

Toma el valor 1 cuando el resultado de una operación es una cadena completa de ceros. Es útil para la comparación por igual ($A=B$) mencionada arriba.

$$\begin{array}{r} \text{Ejemplo en una resta:} \quad - \quad \begin{array}{r} 111 \\ 111 \\ \hline 000 \end{array} \quad Z=1 \end{array}$$

$$\begin{array}{r} \text{Ejemplo en una suma:} \quad + \quad \begin{array}{r} 111 \\ 001 \\ \hline 000 \end{array} \quad Z=1 \end{array}$$

Flag N (Negative)

Toma el valor 1 cuando el primer bit del resultado es 1. Es útil para concluir si un valor es menor a otro en el contexto de CA2, como se describió antes.

$$\begin{array}{r} \text{Ejemplo en una resta:} \quad - \quad \begin{array}{r} 100 \\ 001 \\ \hline 011 \end{array} \quad N=0 \end{array}$$

$$\begin{array}{r} \text{Ejemplo en una suma:} \quad + \quad \begin{array}{r} 101 \\ 001 \\ \hline 110 \end{array} \quad N=1 \end{array}$$

Flag C (Carry)

Toma el valor 1 cuando la resta o la suma tuvieron acarreo o borrow. Es útil para concluir si un valor es menor a otro en el contexto de BSS, como se describió antes.

$$\begin{array}{r} \text{Ejemplo en una resta:} \quad - \quad \begin{array}{r} 100 \\ 001 \\ \hline 011 \end{array} \quad C=0 \end{array}$$

Ejemplo en una suma:

$$\begin{array}{r} 111 \\ + 001 \\ \hline 000 \end{array} \quad C=1$$

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)

$$\begin{array}{r} 011 \Rightarrow 3 \\ + 011 \Rightarrow 3 \\ \hline 110 \Rightarrow 6 \end{array} \quad C=0 \quad \checkmark$$

$$+ \frac{101 \Rightarrow 5}{010 \Rightarrow 2?} \quad C=1 \quad \times \quad - \frac{011 \Rightarrow 3}{110 \Rightarrow 6?} \quad C=1 \quad \times$$

Figura 9.2: Significado del flag Carry en un contexto BSS

$$\begin{array}{r} 011 \Rightarrow 3 \\ + 011 \Rightarrow 3 \\ \hline 110 \Rightarrow 2 \end{array} \quad C=0 \quad \times \quad + \frac{011 \Rightarrow 3}{101 \Rightarrow 3} \quad C=1 \quad \checkmark$$

$$\begin{array}{r} 011 \Rightarrow 3 \\ - 101 \Rightarrow 3 \\ \hline 110 \Rightarrow 2 \end{array} \quad C=1 \quad \times \quad - \frac{111 \Rightarrow 1}{010 \Rightarrow 2} \quad C=0 \quad \checkmark$$

Figura 9.3: Significado del flag Carry en un contexto CA2

Flag V (Overflow)

Toma el valor 1 cuando el resultado no tiene el signo esperado, es decir en los siguientes casos

- (a) Cuando al sumar dos positivos se obtiene un negativo

$$+ \frac{\text{positivo}}{\text{positivo}} \quad \times \quad + \frac{010 \Rightarrow 2}{100 \Rightarrow -4}, V=1$$

- (b) Cuando al sumar dos negativos se obtiene un positivo

$$+ \frac{\text{negativo}}{\text{negativo}} \quad \times \quad + \frac{100 \Rightarrow -4}{000 \Rightarrow 0}, V=1$$

- (c) Cuando al restarle algo positivo a algo negativo se obtiene algo positivo

$$- \frac{\text{negativo}}{\text{positivo}} \quad \times \quad - \frac{110 \Rightarrow -2}{011 \Rightarrow 3}, V=1$$

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

Cuadro 9.3: Detalle de la condición de cada salto condicional

(d) Cuando al restarle algo negativo a algo positivo se obtiene algo negativo

$$\begin{array}{r}
 \text{positivo} \\
 - \text{negativo} \\
 \hline
 \text{negativo}
 \end{array}
 \quad \times \quad
 \begin{array}{r}
 001 \Rightarrow 1 \\
 - 100 \Rightarrow -4 \\
 \hline
 101 \Rightarrow -3
 \end{array}, V=1$$

A partir de los *flags* descriptos antes es posible describir las condiciones de los saltos condicionales, como se indica en el cuadro 9.3.

9.3. Verificar las rutinas

Ya habiendo hablado de los conceptos de rutinas, programas y reuso se presenta un inconveniente, y es que a veces creamos rutinas específicamente para un conjunto de datos sin contemplar ciertos casos particulares. Y fallan ante ciertos valores.

Para corroborar que las rutinas cumplen con la funcionalidad esperada se deben implementar otras rutinas o programas de *test* (pruebas/validación) para hacer un control de calidad sobre las primeras. Con este objetivo, se utiliza un registro que funciona como flag o indicador: si luego de ejecutar la rutina de test ese registro tiene una determinada cadena (por ejemplo F000) entonces la rutina pasó correctamente la validación. Si el registro tiene otra cadena (por ejemplo FFFF) entonces el resultado obtenido no es el que se esperaba. Notar que las cadenas que describen error o éxito son elección arbitraria de la persona que escribe la rutina de test.

En cada caso se deben proveer valores que cumplan lo requerido por la rutina **miRutina** (aquello que se indica en el campo **requiere** de la documentación) y luego de invocarla se debe comparar los resultados obtenidos con aquellos valores que se esperaban según la documentación de **miRutina**. Si coincide se carga el registro flag con el valor F000, y en caso contrario con el valor FFFF.

A continuación se describe el *esquema general de una rutina de test*

```

Pasaje de parámetros ⇒ testRutina:  MOV Rx, 0x...
Invocación ⇒              CALL rutinaAValidar
Comparar con dato esperado ⇒ CMP Ry, 0x...
salto al caso de éxito ⇒    JE funcionaBien
registrar fallo ⇒          MOV R0, 0xFFFF
                          JMP fin
registrar éxito ⇒         funcionaBien: MOV R0, 0xF000
                          fin:  RET

```

testRutina	
Requiere	Nada
Modifica	–
Retorna	en R0 la cadena F000 si la rutina rutinaAValidar retornó el valor que se esperaba, o FFFF en caso contrario

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 sm de 16 bits
Modifica	R4
Retorna	en R6 0000 si es menor que cero, o 0001 si no es menor que cero

A partir de la documentación anterior -y sin conocer el código fuente de la rutina- se puede definir/diseñar el siguiente conjunto de programas de prueba

Test1 Este caso de prueba se realiza con el dato 0000

```

1  test1MQCSM: 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 se realiza con el dato 0008 (que es mayor a cero)

```

1  test2MQCSM: MOV R5, 0x0008
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 se realiza con el dato 8008 (que es menor a cero)

```

1  test3MQCSM: MOV R5, 0x8008
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

```

Test4 Este caso de prueba se realiza con el dato 8000

```

1  test4MQCSM: 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

```

9.3.2. Ejecución de las pruebas

En esta sección se ejecutan los programas de prueba que se hayan definido en la etapa anterior y se analizan los resultados para sacar conclusiones.

Considerando la documentación de la rutina `esMenorQueCeroSM` y la siguiente implementación de esta:

```

1 esMenorQueCeroSM:  MOV R4, R5
2                   DIV R4, 0x8000
3                   RET

```

El resultado de la ejecución de los programas de prueba anteriormente definidos es:

Test1 El resultado obtenido fue R0=F000

Test2 El resultado obtenido fue R0=F000

Test3 El resultado obtenido fue R0=F000

Test4 El resultado obtenido fue R0=FFFF

Como se puede analizar en esta serie de tests la rutina `esMenorQueCeroSM` funciona de forma esperada para todos los valores, excepto para 8000. Al analizar este caso específico se concluye que cero es menor que cero, lo cual no es cierto, y se deberá modificar la rutina teniendo en cuenta este caso de borde.

9.3.3. Otro ejemplo

En esta sección se realiza el diseño de un programa de test para la rutina `avg` y luego se ejecuta para analizar la validez de la rutina. Considerando la documentación de la rutina como sigue:

avg	
Requiere	dos valores de 16 bits en R6 y R7
Modifica	—
Retorna	en R6 el promedio ENTERO entre R6 y R7

Un programa de prueba que podría diseñarse es el que sigue, usando los valores 0008 y 000A, y en cuyo caso se espera el resultado 0009

```
1 testAvg: MOV R6, 0x0008
2          MOV R7, 0x000A
3          CALL avg
4          CMP R6, 0x0009
5          JE funcionabien
6          MOV R0, 0xFFFF
7          JMP fin
8 funcionabien: MOV R0, 0xF000
9 fin: RET
```

Una posible implementación de la rutina `avg` es:

```
1 avg: ADD R6, R7
2      DIV R6, 0x0002
3      RET
```

Al ejecutar el programa `testAvg` se obtuvo el resultado `R0=F000` y por lo se puede concluir que la rutina esta funcionando de la forma esperada.

Capítulo 10

Estructura de 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 normalmente conocidas como repeticiones (también conocidas como iteraciones, ciclos o bucles). Mediante el uso de los saltos es posible actualizar el PC en dirección a una instrucción que ya se ha ejecutado y así lograr que una secuencia de instrucciones se ejecuten mas de una vez.

Además es importante que se tenga alguna forma de terminar esta repetición, ya que de lo contrario el programa entraría en un ciclo infinito cuya ejecución nunca termina. En general esto se realiza con saltos condicionales, considerando alguna condición que puede cambiar cada vez que se ejecuta la secuencia de instrucciones.

Existen muchos problemas que requieren repeticiones para ser resueltos, es decir, requieren realizar la misma operación varias veces hasta llegar al resultado final. Por ejemplo, suponer que se quiere realizar una multiplicación sin utilizar la instrucción `MUL`. Calcular $A*B$ se podría realizar sumando B veces el valor A (o viceversa).

Considerar la siguiente documentación de la rutina `mulSinMul`

mulSinMul	
Requiere	un valor A en el registro $R7$ y un valor B en el registro $R6$
Modifica	El registro $R6$
Retorna	En $R5$ el resultado de $A*B$

El algoritmo (expresado en pseudocódigo) que permite cumplir con esta documentación es como sigue:

```
1 1. Inicializar un acumulador R en cero.
2 2. Verificar si B es cero.
3 3. Si B es cero, salir.
4 4. Asignar R <-- R + A (Notar que la primera vez R es cero)
5 5. Asignar B <-- B - 1 (Al ir restando 1 a B se va acercando al
   final del problema)
6 6. Volver al paso 2.
```

Este problema se puede resolver con lo que se conoce de la arquitectura Q, ya que con saltos condicionales podemos realizar el paso 3.

```

1 mulSinMul: MOV R5, 0x0000 # R5 es el acumulador
2   repetir: CMP R6, 0x0000 # R6 contiene el valor B
3             JE salir
4             ADD R5, R7      # R<-- R + A (en R7)
5             SUB R6, 0x0001
6             JMP repetir
7   salir: RET

```

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 repetición en las rutinas en lenguaje Q. Dicho esquema se conforma con las siguientes partes: *inicialización* donde se inicializan las variables en las que se almacenan (y construyen) los resultados, usualmente contadores y/o acumuladores; la *condición de corte* compuesta por una comparación y un salto condicional hacia fuera de la repetición, el *cuerpo del ciclo* que son las instrucciones que deben repetirse y un *retorno* que se realiza con un salto incondicional hacia la condición de corte. Finalmente, por fuera del ciclo, normalmente se ubica la etiqueta que identifica la primer instrucción fuera de la repetición.

```

1      INICIALIZACION ==> Inicializar contadores o acumuladores
2 repetir: CMP X, Y      ==> X e Y pueden ser registros o celdas
3      JE salir         ==> Puede ser cualquier salto condicional
4      CUERPO DEL CICLO ==> Deberian modificar a X o Y
5      JMP repetir      ==> Retorno
6   salir: ...          ==> Finaliza repeticion, continua el
      programa

```

Entonces, al aplicar la estructura anterior en la rutina `mulSinMul` de la sección anterior se puede identificar la estructura:

```

1 mulSinMul: MOV R5, 0x0000 # Inicializacion del acumulador
2   repetir: CMP R6, 0x0000 # Condicion de corte
3             JE salir
4             ADD R5, R7      # Cuerpo del ciclo
5             SUB R6, 0x0001
6             JMP repetir    # Retorno
7   salir: RET

```

Capítulo 11

Máscaras. Q5

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 (*slice*/rebanada) de la cadena.

Conceptualmente se trata de mantener solo los bits que son de interés, para ocultar los demás detrás de valores determinados o controlados.

Suponer un primer escenario donde una cadena de 8 bits codifica dos datos independientes (dos campos o *slices*): los 4 bits mas significativos son un desplazamiento en X y los 4 menos significativos un desplazamiento en Y. En la figura 11.1 se ilustra una cadena de 8 bits y una máscara de 4 *ventanas* en los bits menos significativos (correspondientes, en este ejemplo, a la coordenada Y), mientras los bits mas significativos se encuentran *bloqueados*. En la figura se muestra el resultado de tapar una parte de la cadena para dejar los bits que si eran de interés.

La tarea de aplicar una máscara es en realidad una operación lógica entre dos cadenas: la cadena con la información a analizar y otra cadena del mismo tamaño con una estructura determinada (bloqueos y ventanas), que se denomina **máscara**. Las operaciones lógicas entre cadenas así mismo son operaciones que se llevan a cabo 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

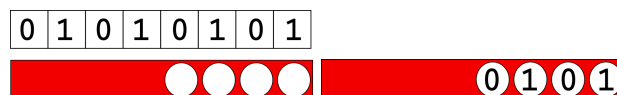


Figura 11.1: Ejemplo de aplicación de máscara (izq) y resultado (der)

$$\begin{array}{rcl}
 \text{AND} & \begin{array}{r} 0101 \\ 0011 \\ \hline ???? \\ 0101 \\ 0011 \\ \hline ???? \end{array} & \Rightarrow \text{AND} \begin{array}{r} 0101 \\ 0011 \\ \hline 0001 \\ 0101 \\ 0011 \\ \hline 0110 \end{array} \\
 \text{XOR} & & \Rightarrow \text{XOR}
 \end{array}$$

Cuadro 11.1: Ejemplo de operaciones bit a bit

como sigue:

$$\begin{array}{r}
 b_3b_2b_1b_0 \\
 \triangle \\
 m_1m_2m_1m_0 \\
 \hline
 r_3r_2r_1r_0
 \end{array}$$

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 mas ejemplos de operaciones lógicas ver ejemplo ver el cuadro 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 valor 1 en determinados bits, el resultado de la operación en esas mismas posiciones coincidirá con el valor 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***

$$X \wedge 1 = X$$

- En cambio, donde la máscara tenga valor 0, el resultado tendrá valor 0. **Es decir que, en la conjunción, un bit en 0 en la máscara determina un *Bloqueo*.**

$$X \wedge 0 = 0$$

De esta manera, siguiendo el ejemplo anterior de las cadenas con coordenadas X e Y, pero llevándolo a cadenas de 16 bits, la cadena de la mascara que se necesita es 0000000011111111, ya que la aplicación bit a bit de la conjunción es como sigue (notar que en color rojo se marcan los bloqueos):

$$\begin{array}{r}
 0101010101010101 \\
 \wedge \quad 0000000011111111 \\
 \hline
 0000000001010101
 \end{array}$$

Esta cadena de resultado representa el desplazamiento en Y que ahora puede usarse de manera independiente. Veamos entonces como sería el uso de máscaras en una rutina, mediante la instrucción **AND**, que calcula la conjunción bit a bit.

```

1 MOV R3, 0x55AA // Cadena de ejemplo 0101010110101010
2 MOV R4, R3     // Copiar el ejemplo
3 AND R3, 0x00FF // mascara 0000000011111111
4              // En R3 queda despY: 0000000010101010
5 AND R4, 0xFF00 // mascara 1111111100000000
6              // En R4 queda despX: 0101010100000000

```

11.2. Disyunción

De la tabla de verdad de la disyunción se deduce que:

- Al utilizar en la máscara valor 1 en determinados bits, 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 valor 0, el resultado coincidirá con el valor que tenga la cadena. **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:

$$\begin{array}{r}
 0101010101010101 \\
 \vee \quad 1111111100000000 \\
 \hline
 1111111101010101
 \end{array}$$

Esta cadena de resultado representa el desplazamiento en Y que ahora puede usarse de manera independiente. Veamos entonces como sería el uso de máscaras en una rutina, mediante la instrucción `OR`, que calcula la disyunción bit a bit.

```

1 MOV R3, 0x5555 ---> 0101010101010101
2 OR  R3, 0xFF00 ---> 1111111100000000 (En R4 queda despY)

```

Or exclusivo

De la tabla de verdad del Or Exclusivo se deduce que:

- Al utilizar en la máscara el valor 1 en determinados bits el valor del resultado de la operación en esas mismas posiciones será el contrario al original.

$$X \oplus 1 = \overline{X}$$

- En cambio, donde la máscara tenga valor 0, el resultado coincidirá con el valor que tenga la cadena, lo que denominamos una ventana

$$X \oplus 0 = X$$

Ejemplo de aplicación:

$$\begin{array}{r} \text{XOR} \quad \begin{array}{r} 0110 \\ 0101 \\ \hline 0011 \end{array} \end{array}$$

11.3. Rutinas con máscaras

Como se dijo anteriormente, 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. En la versión 5 del lenguaje Q (Q_5) se incorporan las siguientes instrucciones:

Instrucción	Tipo	Efecto	Ejemplo
AND	2 operandos	$\text{Dest} \leftarrow \text{Dest} \wedge \text{Origen}$	AND R1,R5
OR	2 operandos	$\text{Dest} \leftarrow \text{Dest} \vee \text{Origen}$	OR R4, 0xFFFF
NOT	1 operando	$\text{Dest} \leftarrow \text{NOT Dest}$	NOT [0xAABB]

Suponer, por ejemplo, que se debe escribir una rutina **esPar** considerando la siguiente documentación:

esPar	
Requiere	Un valor BSS(16) en el registro R6
Modifica	??
Retorna	El código 000F en R5 si R6 es par y el código 000A en caso contrario

Una posible solución para esta rutina es:

```

1 esPar: MOV R4, R6
2       AND R4, 0x0001 // descartar bits restantes
3       CMP R4, 0x0001
4       JNE siEs
5       MOV R5, 0x000A // valor para indicar que es impar
6       JMP fin
7 siEs: MOV R5, 0x000F // valor para indicar que es par
8 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 el valor F000 en caso de éxito o el valor FFFF en caso de fallo.

Retomando el caso de la rutina **esPar**, el siguiente test controla que, al usar como parámetro el valor 8 (que es par) la rutina **esPar** retorna el código esperado (000F).

```

1 testEsPar: MOV R6, 0x0008
2           CALL esPar
3           CMP R5, 0x000F
4           JE funcionabien
5           MOV R0, 0xFFFF
6           JMP fin
7 funcionabien: MOV R0, 0xF000
8 fin:  RET

```

Por otro lado, el siguiente test controla el caso impar, dándole a la rutina el valor AAA1 y esperando encontrar el código 000A.

```
1 testEsImpar: MOV R6, 0xAAA1
2             CALL esPar
3             CMP R5, 0x000A
4             JE funcionabien
5             MOV R0, 0xFFFF
6             JMP fin
7 funcionabien: MOV R0, 0xF000
8 fin: RET
```

Capítulo 12

Arreglos y recorridos sobre arreglos

Mediante el lenguaje de Q podemos resolver muchos de los problemas que se abordan con los lenguajes de mas alto nivel, pero hasta ahora carece de una estructura de datos que permita representar una colección de datos.

12.1. Estructura de arreglos

Un arreglo es un conjunto homogéneo de valores. Esto quiere decir que todos los valores 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 y para eso es necesario indicar el formato con el que se los interpreta.

Por ejemplo, el siguiente es un ejemplo de arreglo con cuatro valores.

0	0050
1	008A
2	00C5
3	000F

En el ejemplo, cada valor representa el monto vendido en un determinado día en una panadería en el sistema CA2(16). Es decir que el *tamaño* del dato es 16 bits y el *tipo* es complemento a 2, pues así se los debe interpretar. En particular, el primer dato (en la posición 0) se tiene la cadena 0050, que representa el valor \$80, correspondiente a las ganancias del día lunes. En la posición 1 se tiene la cadena 008A que representa lo ganado el día martes (\$138). Y así se pueden interpretar las siguientes celdas.

Los arreglos se almacenan en un bloque de celdas de memoria consecutivas y cada uno de sus valores puede ocupar una sola celda, o podrían ser más grandes y ocupar varias celdas. El **tamaño** del arreglo se determina por la cantidad de elementos que tiene (notar que no siempre es la cantidad de celdas que ocupa).

Como otro ejemplo se presenta el siguiente 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 mas datos ver sección 15.2.4).

Tamaño del
arreglo

	...
2000	C26B
2001	8000
2002	0020
2003	B800
2004	C26B
2005	000F
	...

El primer valor es 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 el modo de interpretar estos datos, es decir cómo se documenta en las precondiciones (o especificación) que debe hacerse.

Como tercer ejemplo, considerar el siguiente arreglo con los permisos de un conjunto de 3 archivos en un sistema operativo Linux, donde 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, escribir o ejecutar, 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 puede usar el siguiente formato:

0000000 $R_u W_u X_u$ $R_g W_g X_g$ $R_o W_o X_o$

donde R_i indica si el archivo tiene permiso de lectura para el modo i (el modo es usuario/a, grupo u otros/as). Respectivamente W_i indica el permiso de escritura y X_i el permiso de ejecución. En el ejemplo, el primer elemento es la cadena 01C0 que representa los permisos `rw- --- ---`, es decir que el usuario/a puede leer, escribir y ejecutar el archivo, pero ni los miembros del grupo ni otras personas pueden hacer ninguna operación sobre el archivo. El segundo archivo tiene permisos `rw- rw- rw-`, y entonces puede ser leído, escrito y ejecutado por cualquier persona. Finalmente el último archivo tiene 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

A la hora de trabajar con arreglos es necesario conocer dónde comienza y cuántos elementos tiene. Para lo segundo (tamaño del arreglo) es posible usar diferentes criterios, por ejemplo sabiendo la cantidad de elementos (de manera relativa), o conociendo la posición del último elemento (de forma absoluta) o bien estableciendo una condición de fin (un valor inválido). Ver figura 12.1 con ejemplos de estas estrategias.

Motivación: recorrido de arreglos

Los arreglos son colecciones de datos que deben recorrerse para ser procesa-

Estrategia	A Cant fija de elementos	B Fin en celda fija	C Condicion de fin																																				
Inicio	celda 2000	celda A000	celda 0090																																				
Fin	4 elementos	celda A003	valor FFFF																																				
Elementos	16b	16b	16b																																				
	<table><tr><td></td><td>...</td></tr><tr><td>2000</td><td>0050</td></tr><tr><td>2001</td><td>008A</td></tr><tr><td>2002</td><td>00C5</td></tr><tr><td>2003</td><td>000F</td></tr><tr><td></td><td>...</td></tr></table>		...	2000	0050	2001	008A	2002	00C5	2003	000F		...	<table><tr><td></td><td>...</td></tr><tr><td>A000</td><td>0050</td></tr><tr><td>A001</td><td>008A</td></tr><tr><td>A002</td><td>00C5</td></tr><tr><td>A003</td><td>000F</td></tr><tr><td></td><td>...</td></tr></table>		...	A000	0050	A001	008A	A002	00C5	A003	000F		...	<table><tr><td></td><td>...</td></tr><tr><td>0090</td><td>0050</td></tr><tr><td>0091</td><td>008A</td></tr><tr><td>0092</td><td>00C5</td></tr><tr><td>0093</td><td>FFFF</td></tr><tr><td></td><td>...</td></tr></table>		...	0090	0050	0091	008A	0092	00C5	0093	FFFF		...
	...																																						
2000	0050																																						
2001	008A																																						
2002	00C5																																						
2003	000F																																						
	...																																						
	...																																						
A000	0050																																						
A001	008A																																						
A002	00C5																																						
A003	000F																																						
	...																																						
	...																																						
0090	0050																																						
0091	008A																																						
0092	00C5																																						
0093	FFFF																																						
	...																																						

Figura 12.1: Ejemplos de arreglos

dos, por ejemplo para acumular un resultado (sumatoria, promedio, etc) o para aplicarles a todos los elementos una transformación (aplicarles un descuento, invertir el signo, etc). Ejemplos de estos recorridos de arreglos pueden verse en la figura 12.2

Para llevar a cabo esta tarea de recorrer un arreglo es natural pensar en una repetición, donde en cada vuelta de ciclo se procese un elemento. Pensemos en el caso de calcular la sumatoria de los elementos del arreglo, asumiendo que contamos con el arreglo A de la figura 12.1:

```

1 recorrido: MOV R0, 0x0000 <-- inicializar el acumulador
2           MOV R1, 0x0000 <-- inicializar el contador
3           ciclo: CMP R1, 0x0004
4                 JG fin
5                 ADD R0, [0x2000] <-- problema!
6                 ADD R1, 0x0001
7                 JMP ciclo
8           fin: RET

```

Recorrido con
problema

En el enfoque anterior se ve claramente que con la instrucción `ADD R0, [0x2000]` siempre se suma el mismo elemento, llevándonos a pensar que necesitamos poder cambiar dinámicamente la dirección (en este caso 2000) del elemento. Para solucionar esta problemática es necesario incorporar los modos de direccionamiento indirectos que se explican en el siguiente apartado.

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 el operando mediante una expresión* y no de manera constante como se vio 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*.

En los modos de direccionamiento indirectos, 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**.

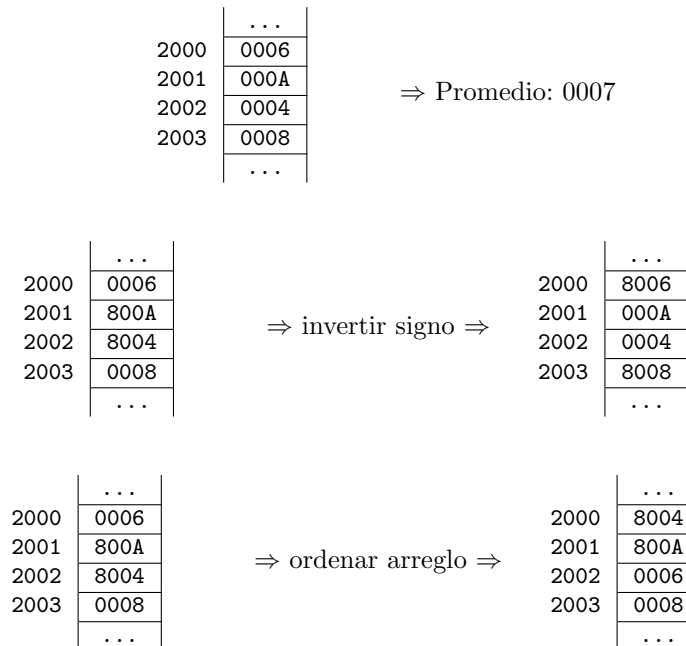
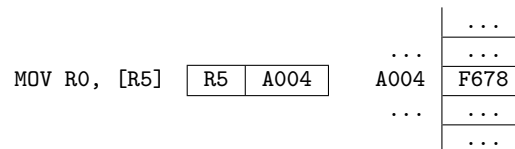


Figura 12.2: Ejemplos de recorrido de un arreglo

También en la vida cotidiana se encuentran *ejemplos de indirecciones*. Suponer como ejemplo la necesidad de encontrar una determinada oficina de la universidad y al preguntarle a una docente de la carrera, esta nos *redirecciona* a la oficina de portería donde hay personas que tienen esa información.

Indirecto por registro:

En este caso se indica el nombre de un registro que, en lugar de contener al operando tiene su dirección. Esto es aplicable al operando origen como al operando destino y la sintaxis necesaria para hacerlo es `[Rx]`. Veamos el siguiente ejemplo:



Entonces el efecto de la ejecución de esta instrucción es

`R0 <-- F678`

Para analizar los accesos a memoria que provoca la ejecución de esta instrucción, es necesario primero poder ensamblarla y para eso se necesita agregar un código para este modo de direccionamiento (ver cuadro 12.1)

Modo	Codificación
Inmediato	000000
Directo	001000
Indirecto memoria	011000
Registro	100rrr
Indirecto registro	110rrr

Cuadro 12.1: Modos de direccionamiento en Q6

De esta manera, el código máquina de la instrucción es 0001 100000 110101 y comprimido en hexadecimal 1835. Esto quiere decir que esta instrucción ocupa una celda en 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 lo está en la celda 0000, y por lo tanto los accesos a memoria pueden describirse en un cuadro como el que sigue:

Instrucción	B.Inst.	B.Op.	Alm.Op.
MOV R0, [R5]	0000	A004	—

Es importante notar que el modo indirecto de esta instrucción provoca un acceso a memoria (en la celda A004 durante la búsqueda de operandos.

Indirecto por memoria:

De manera similar al caso de la Indirección por Registro, la Indirección por Memoria agrega un paso mas para alcanzar el operando, utilizando la sintaxis [[0x00AA]]. Nuevamente, este modo es aplicable tanto al operando origen como al destino. Veamos el siguiente ejemplo

		...

	00AA	A005
MOV R0, [[0x00AA]]
	A005	B010

		...

Para llevar adelante el análisis del ciclo de ejecución es necesario analizar cómo es el código máquina de una instrucción que utilice este nuevo modo de direccionamiento. Teniendo en cuenta nuevamente el cuadro 12.1, vemos que la instrucción MOV R0, [[0x00AA]] se ensambla: 0001 100000 011000 0000 0000 1010 1010 y compactado en hexadecimal esto es 1818 00AA, es decir que el código máquina ocupa 2 celdas. Asumiendo que se lo ubica a partir de la celda 0000, confeccionamos un cuadro como el que sigue:

Instrucción	B.Inst.	B.Op.	Alm.Op.
MOV R0, [[0x00AA]]	0000, 0001	00AA, A005	—

12.3. Recorrido de arreglos

Supongamos una situación donde se cuenta con el siguiente arreglo que contiene las edades de 4 niños, y donde cada edad (es decir cada elemento del arreglo) ocupa una celda:

	...
2000	0005
2001	0006
2002	0006
2003	0005
	...

Entonces supongamos que nos dicen que a partir de la celda 2000 está almacenado el arreglo y que su tamaño es 4. El objetivo es calcular la sumatoria de las edades de los niños, y para esto habíamos realizado un primer abordaje en la sección 12.1, cuando vimos que los modos de direccionamiento conocidos hasta entonces (directo, registro, inmediato) 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 paso del recorrido. Esta variable se denomina **Índice del Arreglo**.

Con este objetivo, cargaremos en el registro R2 el valor 2000, pues esta es la dirección de inicio del arreglo. De esta manera la rutina es como sigue:

```

1 recorrido: MOV R0, 0x0000 <-- inicializar el acumulador
2           MOV R1, 0x0000 <-- inicializar el contador
3           MOV R2, 0x2000 <-- inicializar el 'índice'
4           ciclo: CMP R1, 0x0004 <-- Condición de fin
5                   JGE fin
6                   ADD R0, [??] <-- problema!
7                   ADD R1, 0x0001 <-- contar un elemento
8                   ADD R2, 0x0001 <-- mover el 'índice'
9                   JMP ciclo
10          fin: RET

```

Como último desafío debemos denotar el acceso a cada elemento del arreglo dentro del ciclo (problema), y para esto utilizaremos un modo indirecto vía registro:

ADD R0, [R2]

Finalmente el código de la rutina recorrido se muestra a continuación:

```

1 recorrido: MOV R0, 0x0000 <-- inicializar el acumulador
2           MOV R1, 0x0000 <-- inicializar el contador
3           MOV R2, 0x2000 <-- inicializar el 'índice'
4           ciclo: CMP R1, 0x0004 <-- Condición de fin
5                   JGE fin
6                   ADD R0, [R2] <-- se acumula el elemento actual
7                   ADD R1, 0x0001 <-- contar un elemento
8                   ADD R2, 0x0001 <-- mover el 'índice'
9                   JMP ciclo
10          fin: RET

```

Recorrer un arreglo de tamaño variable

Para este segundo ejemplo consideraremos que el tamaño del arreglo no es fijo por lo cual se delimita el arreglo con un valor especial FFFF, pues no tiene sentido que se almacene que algún niño tiene 65.535 años de edad. Se debe recorrer el arreglo para calcular la sumatoria entre los valores almacenados a partir de la celda 2000 hasta el primer valor FFFF (y ese valor no debe sumarse).

```
1 recoVariable: MOV R0, 0x0000 <-- inicializar el acumulador
2               MOV R2, 0x2000 <-- inicializar el INDICE
3               MOV R1, [R2]   <-- Cargar el primer valor
4               ciclo: CMP R1, 0xFFFF <-- Condición de fin
5                   JE fin      <-- OJO!: es otro salto
6                   ADD R0, R1   <-- se acumula el elemento actual
7                   ADD R2, 0x0001 <-- mover el 'indice
8                   MOV R1, [R2] <-- Cargar un nuevo valor valor
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:

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.
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 2.2 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 sobrescriba 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.

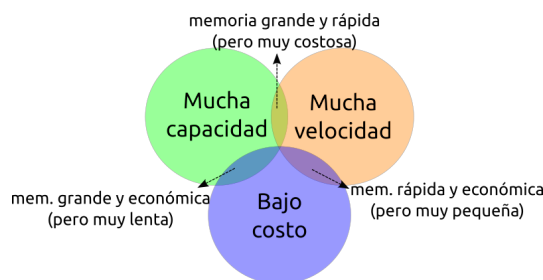


Figura 13.1: Relación de compromiso entre capacidad, velocidad y costo

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.

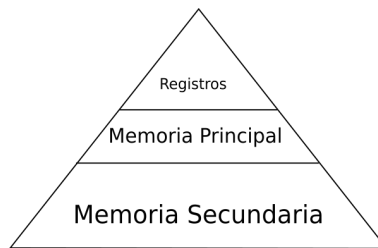


Figura 13.2: Jerarquía de memoria

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 contruidos 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 pérdida 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 pérdida 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

Desde hace un par de décadas el progreso tecnológico de las computadoras está marcando una tendencia de duplicar la velocidad de los procesadores y el tamaño de la memoria principal, pero la velocidad de las memorias apenas crece un 10 %. Esto ocasiona un crecimiento de 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, siendo que al menos una vez es necesario su acceso dentro del ciclo de ejecución de instrucción (ver figura 6.6).

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 un arreglo. De modo que sería este otro ejemplo de celdas de memoria consecutivas relacionadas semánticamente.

Por otro lado, hay situaciones en las que una celda de memoria se accede mas de una vez, por ejemplo cuando contienen 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 de localidad espacial (que se ve en el primer ejemplo de las celdas de un programa o de un arreglo) y el de localidad temporal (como en el caso de las instrucciones del ciclo o la rutina) .

Formalmente, estos patrones de acceso se definen:

1. El principio de **localidad espacial** enuncia que las posiciones de memoria cercanas a alguna celda accedida recientemente son más probables de ser requeridas que las mas distantes.

Principios
de localidad
espacial y
temporal

2. El principio de **localidad temporal** dice que cuando un programa hace referencia a una posición 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 la CPU. Entonces, 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').

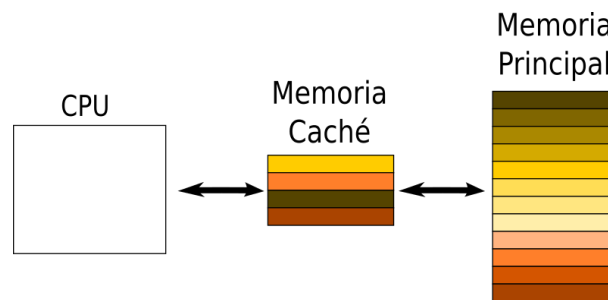


Figura 14.1: Memoria Caché como intermediaria

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

La memoria caché tiene como objetivo proveer una velocidad de acceso cercana a la de los registros pero con un mayor tamaño de memoria. Para ello, la caché contiene una copia de porciones de la memoria principal, que llamaremos bloques, y en el momento de leer una celda, primero se verifica si se encuentra en la caché; **si esto no ocurre** se debe traer de la memoria principal un bloque de celdas para incorporarlos a la misma, para que finalmente, la celda pueda ser entregada a la CPU (ver figura 14.3). El manejo de bloques respeta el principio de localidad espacial, dado al traer a la caché uno de los bloque de datos (porque se necesitó una de sus celdas) será muy probable que próximamente se necesiten otras celdas del mismo bloque.

De manera general el funcionamiento con una memoria caché requiere que la caché resuelva qué celdas son 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.

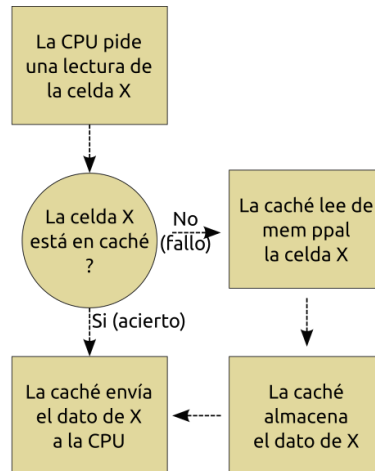


Figura 14.3: Mecanismo de acceso a Memoria Caché

14.1. Función de correspondencia

Las funciones de correspondencia proponen distintos criterios para corresponder los bloques de memoria principal con las líneas de memoria caché, es decir: con qué flexibilidad se almacenan los bloques en las líneas.

Cuando la Unidad de Control pide una determinada celda, la memoria caché debe, en primer lugar, determinar si la dirección corresponde a una celda cachéada. Para responder esta pregunta, debe aplicar una determinada **función de correspondencia**

14.1.1. Correspondencia asociativa

El enfoque más natural es la **correspondencia completamente asociativa**, donde cada línea de caché se puede llenar con cualquier dato de la memoria principal, y por lo tanto se requiere almacenar una etiqueta o *tag* que permita identificar al *dato* que se almacena.

Dicho en otras palabras, el contenido de cada celda leída se asocia a una etiqueta que identifica su origen, es decir, su dirección de memoria principal. Esto permite que las celdas puedan ser almacenadas en cualquier línea¹, y por lo tanto se debe chequear en todas ellas si alguna contiene la dirección buscada como tag.

Por ejemplo, considerar una memoria principal con direcciones de 6 bits (es decir que tiene 64 celdas) y donde cada línea tiene capacidad para almacenar 8

¹ranura=línea=slot

00	01000011111111
01	10101011111111
10	11100011111111
11	01001111111111

Figura 14.4: Ejemplo de caché asociativa, de una celda por línea

00	11000101010111111111111111111111
01	10101111111111111111111111111111
10	00000101010111111111111111111111
11	01101111111111111111111111111111

Figura 14.5: Ejemplo de bloques de 4 celdas

bits. Suponer además una memoria caché como se muestra en la Figura 14.4, con 4 líneas donde cada línea puede almacenar una celda y su identificador (*tag*). Es importante notar que en este caso el *tag* es la dirección completa de la celda. En la figura se ilustran las direcciones de las líneas (00 a 11) y el contenido de las celdas 010000, 101010, 111000 y 010011, ubicando el *tag* en los 6 bits de la izquierda y el *dato* en los 8 bits de la derecha.

Sin embargo, normalmente las memorias caché tienen capacidad para almacenar varias celdas en una misma línea, con el fin de aprovechar el principio de localidad espacial. En estos casos el *tag* ya no es simplemente la dirección de una celda, sino que representa un *bloque* (conjunto de celdas), que está relacionado con las direcciones que lo componen, o mejor dicho, deriva de ellas; y cada uno se identifica como *número de bloque*.

Considerando la memoria principal mencionada, pero utilizando una memoria caché de 4 líneas donde cada línea puede almacenar un bloque de 4 celdas, se puede segmentar la memoria principal en $64/4 = 16$ bloques, y entonces el número de bloque tiene 4 bits, pues $2^4 = 16$. De aquí que el número de bloque 0000 tiene las celdas 000000 a 000011. El número de bloque 0001 tiene las celdas 000100 a 000111. Notar que se subrayó el número de bloque dentro de la dirección de la celda.

A la hora de realizar un acceso de lectura, en primer lugar la caché debe determinar si la celda está *cacheada*. Suponer por ejemplo que se necesita leer la celda 011010, y para eso la caché utiliza los primeros 4 bits que corresponden al número de bloque (0110). En segundo lugar, se debe determinar qué parte del bloque debe enviarse como respuesta a la CPU. Para eso utiliza los últimos 2 bits (en el ejemplo: 10), que se denominan *bits de índice*², que permiten proyectar una porción del mismo. Esto puede resolverse a través de un multiplexor cuyas líneas de control se alimentan con los mencionados bits de índice. En la figura 14.6 se ve que, para el índice 10, el dato que envía es el tercer byte (11111111).

²bits de índice = bits de palabra

11111111	10101010	11111111	10101010
00	01	10	11

Figura 14.6: datos dentro del bloque

00		000000	10111000
01		000001	00011111
10		000010	11110000
11		000011	11111111
		000100	00011100
		000101	11100011
		000110	01100110
		000111	10101010
			...

Figura 14.7: Estado inicial de memoria caché (izq) y memoria principal (der).

00	000010111000000111111111000011111111	000000	10111000
01		000001	00011111
10		000010	11110000
11		000011	11111111
		000100	00011100
		000101	11100011
		000110	01100110
		000111	10101010
			...

Figura 14.8: Estado de la caché luego del primer fallo. En color verde se indican el número de bloque y los bits de tag, y en color rojo los bits de palabra

Ejemplo de accesos de a memoria cache con correspondencia asociativa

En este apartado se explicarán tres lecturas a memoria principal en una arquitectura como la mencionada en la sección anterior, es decir: Una memoria principal con 64 celdas de 8 bits y una memoria caché asociativa de 4 líneas con capacidad de 4 celdas cada una. Se asume que inicialmente la caché está vacía y el contenido de la memoria principal se describe en la Figura 14.7.

El primer acceso es una lectura de la celda 000010. Como primer paso, la caché debe confirmar si la celda buscada está cacheada, y para eso debe comparar el número de bloque con el *tag* de cada línea (que es un sector específico dentro de ésta), de manera simultánea. En este tipo de correspondencia, lo que se utiliza como *tag* es el número de bloque, que en este caso sería la cadena 0000. Dado que está vacía, esa evaluación resulta en un fallo y se lee el bloque completo de memoria principal, alojándose en la primer línea de la caché (00), como se muestra en la Figura 14.8, identificando estos 32 bits de datos (pues son 4 celdas de 8 bits), con el tag 0000 ubicado en el extremo izquierdo de la línea (y marcado en color verde en la figura). En último lugar, se debe servir a la CPU con el contenido de la celda que pidió, y para ello se utilizan los dos bits de índice de la dirección en cuestión (en este caso, 10, marcado en color rojo en la figura) como líneas de control en un multiplexor, proyectando de esta manera el dato 11110000, que es enviado a la CPU.

El segundo acceso es una lectura de la celda 000011, y dado que entonces la caché, al buscar la cadena 0000 en el *tag* de todas las ranuras, lo encuentra en la ranura 00 entonces se trata de un **acierto** y no se realizan lecturas adicionales de memoria principal. Finalmente se proyecta la palabra 11111111 usando los

00	00001011100000011111111000011111111	000000	10111000
01	0001000111001110001101100110101010	000001	00011111
10		000010	11110000
11		000011	11111111
		000100	00011100
		000101	11100011
		000110	01100110
		000111	10101010
			...

Figura 14.9: Estado de la caché luego del segundo fallo.

bits de palabra 11.

Similarmente, la lectura de la celda 000100 es un **fallo** pues el tag 01 no se encuentra en ninguna de las cuatro líneas. Esto ocasiona que se lea un nuevo bloque (el bloque 0001) y se cargue en la línea 01 el contenido de sus celdas, como se muestra en la figura 14.9.

14.1.2. Correspondencia Directa

La implementación de una correspondencia asociativa tiene un costo elevado debido a que se requiere la búsqueda del *tag* (dirección de la celda o número de bloque) en todas las líneas en forma simultánea y esto requiere una tecnología de acceso aleatorio. Para reducir este costo es posible presentar otro mecanismo de correspondencia donde cada bloque de memoria principal esté asignado a una línea determinada.

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), cada línea tiene un conjunto de bloques candidatos que “compiten” entre sí. Este mecanismo se denomina **correspondencia directa** y cada línea de caché es considerada como un recurso compartido entre un conjunto de bloques de memoria. Es importante marcar que este conjunto de bloques asociado a una misma línea no es consecutivo, con el objetivo de aprovechar el principio de localidad y maximizar la tasa de aciertos. El criterio que se usa para corresponder bloques con líneas es circular, como describe la figura 14.10.

A la hora de verificar si una celda está *cacheada*, la memoria caché debe comparar solamente en una línea (la que corresponde al bloque) para determinar si contiene el *tag* buscado, simplificando la tecnología que se necesita para implementar la caché.

Continuando el ejemplo de la memoria descrita en la sección anterior, considerar una caché con 8 líneas y bloques de 4 celdas. En esta correspondencia directa el algoritmo para determinar si la celda pedida está ya almacenada en caché requiere que se determine, en primer lugar, qué bloque se necesita, y en segundo lugar en qué línea se debe buscar el tag.

En este ejemplo se tienen 16 bloques, pues los bloques son de 4 celdas:

$$\frac{64 \text{ celdas}}{4 \text{ celdas x bloque}} = 16 \text{ bloques}$$

Entonces, con 8 líneas se tienen $16/8 = 2$ bloques por línea, y por lo tanto el *tag* debe permitir identificar uno de esos bloques candidatos. Es decir que el

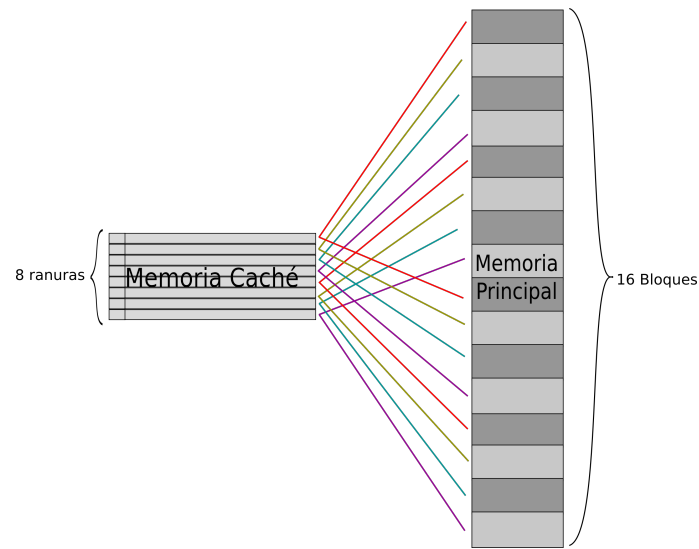


Figura 14.10: Correspondencia directa

tag debe representar 2 posibles estados, y entonces requiere sólo un bit. De esta manera, en la correspondencia directa el *tag* es mas pequeño que en el caso de la correspondencia asociativa.

En este ejemplo, la ranura 000 almacena el bloque 0000 o el bloque 1000, y de aquí que el bit mas significativo dentro del número de bloque puede usarse como *tag*.

Ejemplo de accesos a memoria caché de correspondencia directa

En este apartado se explicarán tres lecturas a memoria principal en una arquitectura co una memoria principal con 64 celdas de 8 bits y una memoria caché de correspondencia directa de 8 líneas con capacidad de 4 celdas cada una. Se asume que inicialmente la caché está vacía y el contenido de la memoria principal se describe en la Figura 14.11.

1. En la primera lectura la CPU pide la celda 011010. El número de bloque son los primeros 4 bits (0110) y a ese bloque le corresponde la línea 110. Entonces debe verificarse que esa línea contenga el tag 0. Como la caché está vacía ocurre un **fallo**. Entonces se lee el bloque 0110, que corresponde a las celdas 011000 a 011011. Finalmente, se utilizan los **bits de palabra** (10) para extraer una porción de la línea como ocurre en el caso asociativo. La caché queda cargada como se indica en la Figura 14.11.
2. En la segunda lectura la CPU pide la celda 011011. El número de bloque son los primeros 4 bits (0110) y a ese bloque le corresponde la línea 110. Entonces debe verificarse que esa línea contenga el tag 0. Como esto coincide, se trata de un **acierto** y por lo tanto se utilizan los **bits de palabra** (11) para extraer el dato 11111111 y enviarlo a la CPU por el bus de datos. No ocurren cambios en la caché.

000		011000	10111000
001		011001	00011111
010		011010	11110000
011		011011	11111111
100		011100	00011100
101		011101	11100011
110	010111000000111111111000011111111	011110	01100110
111		011111	10101010
			...

Figura 14.11: Estado de la caché luego del primer fallo. En color verde se indican el número de bloque y los bits de tag, y en color rojo los bits de palabra

000		011000	10111000
001		011001	00011111
010		011010	11110000
011		011011	11111111
100		011100	00011100
101		011101	11100011
110	010111000000111111111000011111111	011110	01100110
111	000011100111000110110011010101010	011111	10101010
			...

Figura 14.12: Estado de la caché luego del segundo fallo.

- En la tercer lectura la CPU pide la celda 011100. El número de bloque son los primeros 4 bits (0111) y a ese bloque le corresponde la línea 111. Entonces debe verificarse que esa línea contenga el tag 0. Como esa línea está disponible, se trata de un **fallo** y debe leerse el bloque 0111, que corresponde a las celdas 011100 a 011111. Finalmente, se utilizan los **bits de palabra** (00) para extraer el dato 00011100 y enviarlo a la CPU por el bus de datos. La caché queda cargada como se ve en la Figura 14.12

14.1.3. 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 14.13 se ejemplifica esta nueva forma de correspondencia.

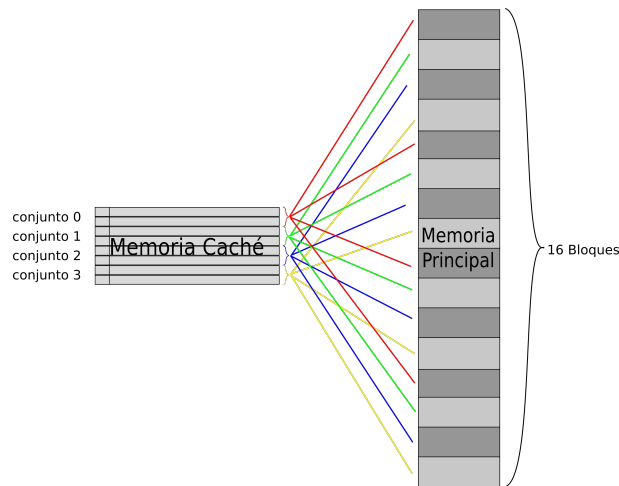


Figura 14.13: Correspondencia asociativa por conjuntos

Con este mecanismo, las líneas 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 qué 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 14.13, con 16 bloques de 4 celdas en la memoria principal, y una memoria caché con 4 conjuntos de 2 líneas cada uno.

Por ejemplo, el conjunto 00 puede 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 líneas del conjunto 00, buscando el tag 11.

14.1.4. Fallos y reemplazos

Cuando se produce un **fallo** y un nuevo bloque debe ser cargado en la caché, debe elegirse una ranura 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 ranura 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.

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 ultimo acceso, indicando cuan reciente fue la última referencia a cada bloque. Si se produce un fallo, se reemplaza aquel bloque que está ultimo 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

Cuadro 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.

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 cargado 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 por conjuntos 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 setea el bit (de 0 a 1) de la siguiente línea.

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.

Algoritmo Aleatorio

Con esta política, 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. Esta política es contraria 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 alguna para cada bloque.

14.1.5. 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 debe 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. La condición mínima es 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 B110 y se sabe que $R1 = AC00$, $R3 = A702$, $SP = FFEE$ y que la celda A702 tiene el valor 1.

	...
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 a la memoria principal, y para cada una si 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, es decir que se tiene una **tasa de fallos** de $\frac{4}{8} = 0,5$. Si el tiempo de acceso a la memoria es de $0,5ms$ y el de la cache es $0,05ms$, se sabe que la ejecución de ese programa llevó en total:

$$4 * (0,5ms + 0,05ms) + 4 * 0,05ms = 2,4ms$$

Capítulo 15

Sistemas de numeración fraccionarios

Para representar números reales en una computadora es necesario tener una manera de codificar la "coma raíz". La forma mas simple se implementa conviniendo la posición de la coma en un lugar fijo, separando de esta manera la cadena en dos partes: parte entera y parte fraccionaria, y eso es lo que se conoce como **punto fijo**. 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).

15.1. Sistemas de Punto Fijo

El sistema numérico de punto fijo es una generalización de los sistemas enteros que permite representar números fraccionarios. En un sistema de punto fijo, se destinan 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. Sin embargo, 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 2 negativa, 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. Entonces, para interpretar una cadena en punto fijo, se debe interpretar por un lado la parte entera en el sistema correspondiente y por otro lado la parte fraccionaria.

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 5 bits para la parte entera:

$$2^4 + 2^1 + 2^0$$

y 3 bits para la parte fraccionaria:

$$2^{-1} + 2^{-3}$$

Quedando la composición de ambas partes de la siguiente manera:

$$\begin{aligned} I_{BSS(8,3)}(10011101) &= 2^4 + 2^1 + 2^0 + 2^{-1} + 2^{-3} \\ &= 16 + 2 + 1 + 0,5 + 0,125 = 19,625 \end{aligned}$$

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} .

$$\text{cadena} \xrightarrow{I_{bss}} \text{valor entero} \xrightarrow{\text{escala}} \text{valor fraccionario}$$

Aplicando este mecanismo al ejemplo anterior, la cadena 10011101 se interpreta en BSS como:

$$\begin{aligned} I_{BSS(8,3)}(10011101) &= 2^7 + 2^4 + 2^3 + 2^2 + 2^0 \\ &= 128 + 16 + 8 + 4 + 1 = 157 \end{aligned}$$

Finalmente se escala el valor entero:

$$157 \cdot 2^{-3} = \frac{157}{2^3} = \frac{157}{8} = 19,625$$

(notar que 2^{-3} es la escala del sistema, con respecto a BSS)

Pasando en limpio:

$$10011101 \xrightarrow{I_{bss}} \mathbf{157} \xrightarrow{\times 2^{-3}} \text{valor } \mathbf{19,625}$$

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 aproximarlos con 0,00 o aproximarlos con 0,01. En el último 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

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 (o *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 *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 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 .

$$\text{valor fraccionario} \xrightarrow{\text{escala}} \text{valor entero} \xrightarrow{R_{bss}} \text{cadena}$$

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. $R_{bss(7)}(61) = 0111101$

$$\text{valor } 3,8 \xrightarrow{\times 2^4} 60,8 \xrightarrow{\text{redondeo}} 61 \xrightarrow{R_{bss}} 0111101$$

Al haber una cantidad limitada de bits de 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 racionales¹ pero siempre existe en los irracionales.

El error absoluto indica la diferencia entre el número x (que se quería representar) y el número x' (valor representable que es la mejor aproximación de x). Se calcula mediante la distancia:

$$EA(x) = |x - x'|$$

Notar que el error absoluto debe ser menor a la mitad de la resolución, es decir $EA(x) \leq R/2$.

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 X 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 mas importante que 20 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 $[0,1]$.

¹Un número x es racional si existen $a \in \mathbb{Z}$ y $b \in \mathbb{Z}$ tales que $x = \frac{a}{b}$. Los racionales tiene 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*.

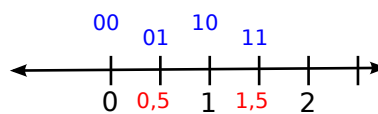


Figura 15.1: BSS(2,1)

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.

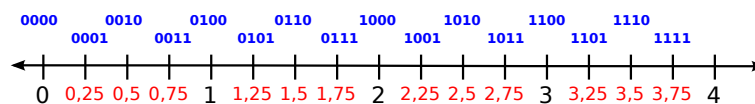


Figura 15.2: BSS(4,2)

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, Δ) 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 fraccionaria
	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 a 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. Por eso es importante relativizar el error, pensándolo como una proporción (o un porcentaje). Por ejemplo, suponer que una persona llega 10 minutos tarde a una clase. Eso es mucho más "grave" que llegar 10 minutos más tarde de lo previsto luego de un viaje de 2000 kilómetros. Digamos que 10 minutos de retraso en un tiempo total más grande resulta aceptable.

Esto indica una limitación en los sistemas de punto fijo, donde al representar valores muy pequeños (cerca del 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.

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 normalmente toma un valor en el intervalo $(0, 9]$; y el *exponente*, que permite “recordar” dónde estaba la coma antes de abreviar. Ejemplos:

exponente

$$0,00000000000000000000000062 = 62 * 10^{-20}$$

$$m \times 2^e$$
$$I(11000000) = 2^7 + 2^6 = 64 + 128 = 192$$
$$I(11000000) = I(11) * 2^6 = (2^2 + 2^1) * 2^6 = 3 * 64 = 192$$
$$I_{bss}(11) \times 2^{I_{bss}(110)}$$
$$I(11000000000000000) = I(11) * 2^{15}$$

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

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 siguiente tabla 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	$I_{bss}(11) = 3$	$I_{bss}(11) = 3$	$N = 3 \times 2^3 = 24$

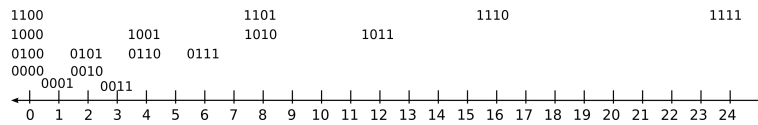


Figura 15.3: Mantisa en BSS(2) y Exponente en BSS(2)

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 (o rango de resoluciones) de cualquier sistema de punto flotante está determinado por los sistemas subyacentes. Por ejemplo, considerar el caso donde tanto mantisa como exponente permiten representar un rango positivo (ver figura 15.3) en contraposición a un sistema donde la mantisa permite representar números negativos (ver figura 15.4).

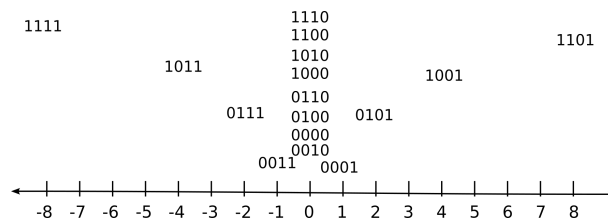


Figura 15.4: Mantisa en SM(2) y exponente en BSS(2)

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).

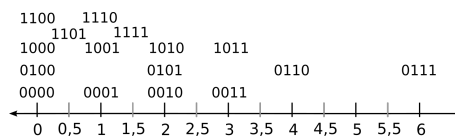


Figura 15.5: Mantisa en BSS(2) y exponente en SM(2)

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.

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

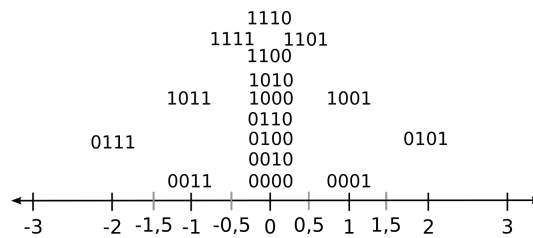


Figura 15.6: Mantisa: SM(2), Exponente: SM(2)

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. Interpretar 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 resolución en un sistema normalizado

Calculemos la resolución máxima y mínima de un sistema de punto flotante normalizado con bit implícito como el siguiente: sistema de punto flotante con

Mantisa: $SM(4+1, 4)$

Exponente: $CA2(4)$

- a) Para la resolución mínima se necesita el mínimo exponente, es decir, 1000 en este sistema. El cual representa, como ya se vió, al número -8. Las cadenas que voy a utilizar son 0000 1000 y 0001 1000, las cuales son consecutivas y como las mantisas están normalizadas y tienen bit implícito, la primera vale

$$2^{-1} \times 2^{-8}$$

y la segunda

$$(2^{-1} + 2^{-4}) \times 2^{-8}$$

Y al restar los valores se obtiene:

$$2^{-1} \times 2^{-8} - (2^{-1} + 2^{-4}) \times 2^{-8} = 2^{-1} \times 2^{-8} - 2^{-1} \times 2^{-8} + 2^{-4} \times 2^{-8}$$

Por lo tanto, la **resolución mínima** es:

$$2^{-4} \times 2^{-8}$$

- b) Para la resolución máxima es necesario el máximo exponente, es decir, 0111, que representa, como ya vimos, 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 primera vale

$$2^{-1} \times 2^7$$

y la segunda

$$(2^{-1} + 2^{-4}) \times 2^7$$

Y al restar ambas expresiones se obtiene:

$$2^{-1} \times 2^7 - (2^{-1} + 2^{-4}) \times 2^7 = 2^{-1} \times 2^7 - 2^{-1} \times 2^7 + 2^{-4} \times 2^7$$

Por lo tanto, la **resolución máxima** es:

$$2^{-4} \times 2^7$$

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:

$$\text{Resolución mínima} = I_m(0..01) \times 2^e$$

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:

$$\text{Resolución máxima} = I_m(0..01) \times 2^E$$

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.

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. sim- ple	P. do- ble
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 $\text{emin}-1$ y $\text{emax}+1$ en ambas precisiones para representar valores especiales. Nótese que esta elección no es arbitraria: la cadena que representa $\text{emin}-1$ está compuesta de ceros y la cadena que representa el valor $\text{emax}+1$ 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):

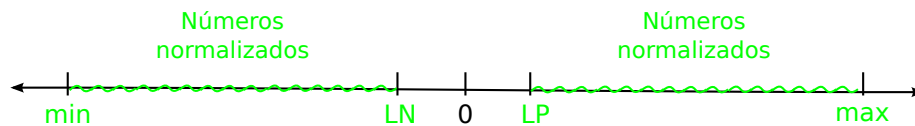


Figura 15.7: Normalizados del estándar IEEE 754

Números denormalizados Las cadenas de esta clase tienen exponente nulo pero mantisa no nula. Gráficamente, la clase de números denormalizados se distribuye como sigue:

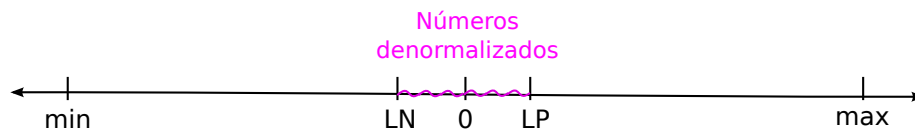


Figura 15.8: Denormalizados del estándar IEEE 754

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 $\text{Exc}(8,127)$

$$e = I_{ex}(10000100) = I_{bss}(10000100) - 127 = 2^7 + 2^2 - 127 = 5$$

- Mantisa: Dado que hay un bit implícito cuyo peso es $2^0 = 1$.

$$\begin{aligned} m &= -(1 + I_{bss(23,23)}(11010111000000000000000)) = \\ &= -(1 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-7} + 2^{-8}) \end{aligned}$$

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**

$$\begin{aligned} m &= I_{bss(23,23)}(01000001011100000000000) = \\ &= 2^{-2} + 2^{-8} + 2^{-10} + 2^{-11} + 2^{-12} \end{aligned}$$

Apéndice A

Validación de programas

Prueba de escritorio

Como se presentó en el apartado 4.3.3, 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 (0000_{16} o 0001_{16}). Entonces es posible concluir que por cómo está construido este programa no funciona con valores mayores a 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 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 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	$\{R7, \text{Dest}\} \leftarrow \text{Dest} * \text{Origen}$
MOV	0001	$\text{Dest} \leftarrow \text{Origen}$
ADD	0010	$\text{Dest} \leftarrow \text{Dest} + \text{Origen}$
SUB	0011	$\text{Dest} \leftarrow \text{Dest} - \text{Origen}$
AND	0100	$\text{Dest} \leftarrow \text{Dest} \wedge \text{Origen}$
OR	0101	$\text{Dest} \leftarrow \text{Dest} \vee \text{Origen}$
CMP	0110	Modifica los Flags según el resultado de $\text{Dest} - \text{Origen}$
DIV	0111	$\text{Dest} \leftarrow \text{Dest} \% \text{Origen}$

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	$PC \leftarrow \text{Origen}$
CALL	1011	$[SP] \leftarrow PC; SP \leftarrow SP - 1; PC \leftarrow \text{Origen}$

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	$\text{Dest} \leftarrow \text{NOT Dest (bit a bit)}$

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	$SP \leftarrow SP + 1; PC \leftarrow [SP];$

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.

Prefijo(1111)	Cod.Op (4)	Desplazamiento(8)
---------------	------------	-------------------

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 últimos 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.