Una Introducción al Diseño Digital de Computadoras

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Mayo 2006

Índice general

Prefacio

Estas notas describen cómo funcionan las computadoras. Están escritas para personas que usan pequeñas computadoras, pretenden usarlas, o que simplemente están interesadas en general por ellas. Comenzando con las compuertas más elementales y construyendo hacia una computadora completa, se discuten todas las fases de desarrollo de computadoras. También se presenta y discute el sistema numérico binario y la forma en que una computadora funciona con tales números.

Además de describir las partes de una computadora, se discute la programación elemental en lenguaje de máquina, y se habla de lenguajes ensamblador y de alto nivel. El objetivo de estas notas, sin embargo, no es la enseñanza de lenguajes específicos, sino proveer de un entendimiento de qué lenguajes hay y cómo operan con la computadora. Aun cuando estas notas son para el lector individual, pueden usarse también como textos para un curso básico de arquitectura de computadoras.

Jorge L. Ortega Arjona
Mayo 2006

Capítulo 1

Introducción

La computadora digital moderna era originalmente un dispositivo grande, tanto que requería una gran habitación para contenerla. Además, era cara, costando comúnmente más de un millón de dólares. Debido a su tamaño y costo, las primeras computadoras pertenecían sólo a grandes compañías y universidades, y se utilizaban principalmente para operaciones de procesamiento de datos.

Los desarrollos tecnológicos permitieron que cada vez un número mayor de personas hicieran uso de las computadoras. Sin embargo, tal número de personas era muy limitado. Las computadoras se usaban generalmente para resolver problemas matemáticos complejos de ingeniería o ciencia, o también para realizar nóminas y otras operaciones de procesamiento de datos de grandes compañías.

Las computadoras digitales originales utilizaban tubos al vacío o bulbos. La introducción del transistor permitió el desarrollo de poderosas computadoras que eran físicamente más pequeñas y más confiables. Sin embargo, esta computadoras eran todavía grandes y costosas. El desarrollo de los circuitos integrados cambió todo eso. Muchos transistores y circuitería asociada pueden construirse en un pequeñísimo circuito de silicio. El costo y tamaño de un circuito complejo llegó a ser similar a aquél que utilizaba transistores solo unos años antes. Los circuitos integrados de alta escala han llevado esto un paso más adelante. Actualmente, circuitos de computadora extremadamente complejos pueden construirse en un fragmento de silicio. Esto ha dado como resultado computadoras pequeñas y relativamente baratas.

Una computadora hoy está disponible para un mayor número de personas, y con diversos usos. Se utilizan, por ejemplo, desde el control de inyección de combustible de los automóviles, hasta la dirección de operaciones de casi todas las compañías. Se utilizan también en todo tipo de instrumentos para proveer de información inmediata que antes no era disponible. A nivel personal, se usan para controlar las finanzas, jugar, y hacer otras operaciones, que se limitan tan solo por la imaginación del usuario.

En estas notas se discute cómo funciona una computadora digital. Además de presentar estas ideas básicas, también se mencionan algunas características generales de computadoras disponibles, así como de los componentes que las constituyen.

1.1. La computadora y las cosas que se pueden hacer con ella

Considérese una computadora en operación. Un ingeniero electrónico, por ejemplo, le suministra las especificaciones de un amplificador de audio de alta fidelidad, y en segundos, la computadora imprime los componentes del amplificador. O tómese al usuario que juega ajedrez con su computadora. Para toda movida que hace, la computadora responde con un movimiento. Tal proceso continúa hasta que el juego termina, muchas veces ganando la computadora. En otro caso aún, el gerente de un departamento de crédito le proporciona una lista de números de cuenta de clientes, así como sus compras y pagos del mes. La computadora se encarga entonces de enviar a cada cliente su estado de cuentas y cobros, basándose en sus compras y pagos, el balance anterior y los cargos por servicios.

En todos estos ejemplos parece como si la computadora acepta datos, "piensa" y produce una salida. En realidad, un proceso algo diferente toma lugar. La computadora no *piensa*. Debe ser *programada*, es decir, debe ser dirigida para realizar un conjunto específico de operaciones. En todos los ejemplos anteriores, tal programa (o secuencia de instrucciones) ha sido almacenado previamente en la computadora.

Las computadoras actuales pueden almacenar una gran cantidad de información. Tal información consiste tanto de programas que pueden ejecutarse, como de datos sobre los cuales se ejecutan los programas. En el ejemplo del departamento de crédito, los datos almacenados pueden ser el número de cuenta, nombre, balance de crédito, cargo por servicio, pagos y

compras. Además, también es necesario tener almacenado un conjunto de instrucciones (o programa) que dirige la acción de la computadora, a fin de, por ejemplo, imprimir los recibos y estados de cuenta.

El programa puede dirigir a la computadora para realizar, paso a paso, complejas operaciones, usando tan solo operaciones aritméticas básicas (suma, resta, multiplicación y división) y algunos otros relativamente simples procedimientos. Por ejemplo, dos números pueden compararse y la computadora puede determinar si son iguales, o si uno es mayor que el otro. Usando procedimientos sencillos, una computadora puede hacer que se realicen operaciones complejas, produciendo la apariencia de que algún proceso de pensamiento se lleva a cabo. Por ejemplo, si el balance de pago de un cliente excede una cierta cantidad, se puede añadir un mensaje de advertencia para el cliente.

Aun cuando en estas notas se busca explicar cómo funciona una computadora, por otro lado no se discute en detalle cómo puede ser programada. Para esto, se sugiere al lector referirse a alguno de los tantos libros sobre programación disponibles.

La introducción tecnológica del *microprocesador* ha permitido reducir el tamaño, y en la actualidad, el costo de las computadoras, que pueden ser adquiridas por personas individuales para utilizarlas en su trabajo o entretenimiento. Estas *computadoras personales* (o PCs, por sus siglas en inglés) pueden utilizarse para jugar una variedad de juegos como el ajedrez, o algún otro que haga al usuario imaginar que viaja por el espacio o explora cavernas y laberintos.

El propietario de una computadora personal puede usarla tal y como una gran compañía utiliza una computadora grande para, por ejemplo, llevar control de sus finanzas domésticas, impuestos y presupuestos. Llevado en ocasiones al extremo, la computadora puede utilizarse para controlar varias otras actividades en el hogar. Por ejemplo, la calefacción puede controlarse, de tal modo que se logren ahorros en el consumo de energía. Una computadora puede automáticamente leer información de sensores, a fin de que las alarmas contra incendio o ladrones se hagan más sensibles: la más pequeña aparición de luz en una habitación obscura durante la noche puede automáticamente disparar una alarma.

Muchas de las aplicaciones de la computadora se encuentran en las áreas de ingeniería, ciencia y matemáticas, donde es frecuente la necesidad de re-

solver conjuntos de ecuaciones simultáneas con muchas incógnitas. Si esto se hiciera a mano (o con una simple calculadora), la solución requeriría horas, o hasta días. Dependiendo de la complejidad del cálculo, la computadora puede realizarlo en minutos, o tal vez, hasta en segundos. Hay, por supuesto, muchas otras aplicaciones de esta naturaleza. Cuando ondas de radar rebotaron en la superficie de Venus, la señal de retorno era tan débil que no podía distinguirse del ruido o la interferencia. Las computadoras se utilizaron entonces para realizar complejos cálculos, que permitieron distinguir las señales del ruido. También, las computadoras se utilizan ámpliamente en los negocios a fin de llevar a cabo labores contables, como inventarios, balances y operaciones diarias de envío y recepción de mercancías.

Se han mencionado algunos usos de la computadora personal. Es posible que entender la operación de este tipo de computadoras pueda expresar más claramente sus capacidades y limitaciones, de tal modo que se puedan descubrir nuevos usos. Tal entendimiento es el objetivo de estas notas.

1.2. Ideas básicas sobre computadoras digitales – componentes de una computadora digital

Las computadoras digitales operan sobre números. A veces, estos números son códigos que representan instrucciones de un programa, o datos tales como el nombre de una persona. Por ejemplo, si se introduce el nombre de una persona a la computadora, debe convertirse primero en un conjunto de códigos numéricos apropiados. Más aún, los números pueden representar también números verdaderos.

Si el objetivo es entender cómo funciona una computadora, es necesario conocer su *sistema numérico*. En general, los seres humanos trabajamos con un sistema numérico decimal que utiliza diez dígitos. Se considera que este sistema se desarrolló debido a que las personas normalmente cuentan con diez dedos. Sin embargo, las computadoras son esencialmente un conjutno de *interruptores* que se encuentran en uno de dos estados: encendido (*on*) o apagado (*off*). Así, los componentes de una computadora trabajan con un sistema numérico de dos dígitos, llamado *sistema numérico binario*, el cual usa 0 y 1 para representar los estados de apagado y encendido respectivamente.

Considérese ahora cómo es la organización interna de una computadora digital. En los capítulos subsecuentes, las ideas a continuación se presentan

con mayor detalle. Sin embargo, parece necesario aclarar que la terminología usada para describir la organización interna de una computadora digital no está completamente estandarizada, de tal modo que la descripción aquí utiliza terminología común utilizada para computadoras pequeñas. Por lo tanto, una computadora digital puede describirse como un diagrama de bloques (Figura 1.1). A continuación, se describe brevemente cada uno de sus componentes genéricos.

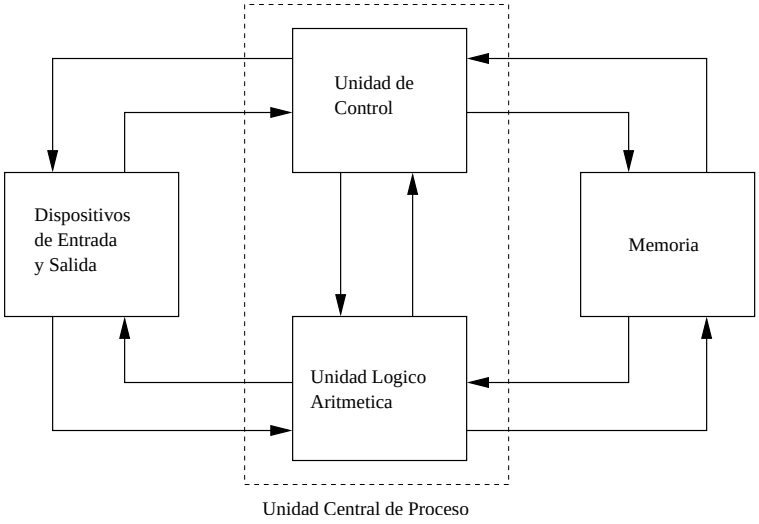

Figura 1.1: Unidades básicas de una computadora digital.

Unidad Lógico Aritmética

La *Unidad Lógico Aritmética* (o ALU, por sus siglas en inglés) se encarga de realizar todas las operaciones lógicas y aritméticas definidas sobre los valores numéricos binarios. Por ejemplo, dos números binarios pueden sumarse para producir un tercero.

Unidad de Control

La *Unidad de Control* (o CU, por sus siglas en inglés) dirige la operación de la computadora. Es el elemento que se encarga de controlar el flujo de instrucciones que la computadora va llevando a cabo. Por ejemplo, puede dar las instrucciones a la ALU para la suma de los dos números binarios.

Unidad Central de Proceso

Normalmente, la ALU y la CU se conjuntan en un circuito único llamando *Unidad Central de Proceso* (o CPU, por sus siglas en inglés). Sin embargo, tal terminología no es completamente estándar, ya que en algunas computadoras ambas unidades aparecen como elementos independientes de una computadora.

Memoria

Todos los datos y los programas cuyas instrucciones los modifican se encuentran almacenados en la *memoria* de la computadora digital. En realidad, en una computadora digital, existen varios tipos de unidades de memoria. Por ejemplo, la *Unidad Principal de Memoria* (o MMU, por sus siglas en inglés) es el almacenamiento principal y de trabajo de una computadora. Es en ella donde se encuentran los datos, así como las instrucciones de los programas que se ejecutan sobre tales datos. Normalmente, en computadoras modernas la memoria principal consiste de circuitos semiconductores.

Un tipo particular de memoria semiconductora es la *Memoria de Sólo Lectura* (o ROM, por sus siglas en inglés). Este tipo de memoria contiene datos permanentemente almacenados, como por ejemplo, podría ser una tabla de valores de funciones trigonométricas. Tales datos generalmente, aunque no siempre, se incorporan a la memoria durante su manufactura.

Otro tipo particular de memoria semiconductora, pero con mayor versatilidad en su uso durante la ejecución de instrucciones, es la *Memoria de Acceso Aleatorio* (o RAM, por sus siglas en inglés). Esta memoria es capaz de modifiar su estado mientras permanezca encendida, permitiendo tanto la lectura como la escritura de valores numéricos binarios al ejecutar las instrucciones de un programa.

Además de la memoria principal, una computadora digital cuenta también con *Unidades de Memoria Auxiliares*. Tales memorias son capaces de almacenar una gran cantidad de información en componentes magnéticos, como son discos o cintas. Las memorias auxiliares se consideran de bajo costo en el sentido de que permiten almacenar una gran cantidad de datos a un costo razonable. Sin embargo, las operaciones para escribir y leer información de este tipo de memoria toman mucho más tiempo comparado con el tiempo de acceso de la memoria principal semiconductora.

Dispositivos de Entrada/Salida

Una computadora sería inútil a menos que pudieran comunicarse con el usuario, a fin de que éste proporcionase información a ser procesada y recibiese los resultados de tal procesamiento. Esta es la función de los *Dispositivos de Entrada/Salida* (o I/O, por sus siglas en inglés). Un dispositivo de salida típico es el monitor de una computadora personal, mientras que un dispositivo de entrada típico es el teclado de la misma. El usuario teclea los símbolos sobre el teclado, que genera las señales eléctricas, en forma de ceros y unos, para introducirlas a la computadora. De forma similar, otras señales eléctricas se envían de la computadora al monitor, de tal modo que éste muestra la información que el usuario teclea.

Otra forma de dispositivo de salida es la impresora, que puede imprimir la información de la computadora en una hoja de papel, a fin de que pueda ser examinada por el usuario. Existe un gran número de tipos de impresoras, que operan a diferentes velocidades y resoluciones, permitiendo que grandes cantidades de datos puedan ser analizados.

1.3. Construcción de computadoras – microprocesadores

Los circuitos electrónicos digitales dentro de la Unidad Central de Proceso y la memoria principal se componen de "interruptores", que pueden encontrarse encendidos o apagados. Las primeras computadoras realmente utilizaban interruptores controlados eléctricamente, llamados *relevadores*. Los relevadores tenían dos desventajas: eran relativamente lentos, y ocupaban mucho espacio. Eran necesarios varios milisegundos para que un relevador cambiara su estado de abierto a cerrado. Al principio, tal velocidad puede parecer muy rápida. Sin embargo, durante la ejecución de un programa de computadora común, se requiere abrir y cerrar los relevadores varios miles de millones por segundo, a fin de obtener una respuesta en un tiempo aceptable para el usuario. De tal modo, una computadora basada en relevadores requería mucho tiempo para operar. Además, ya que los relevadores ocupan un espacio relativamente grande, las computadoras basadas en relevadores también tendían a ser grandes.

A mediados de la década de 1940, los relevadores fueron reemplazados por tubos al vacío o bulbos, los cuales operaban como interruptores eléctricos controlados por electricidad. Tales bulbos eran más pequeños y rápidos

que los relevadores. De hecho, un bulbo rápido puede operar en el orden de microsegundos ($\mu s = 10^{-6} segundos$). Así, la velocidad de las computadoras se incrementó grandemente. Sin embargo, los bulbos tienen una gran desventaja: se funden y deben ser cambiados con mucha frecuencia. Y dado que una computadora de bulbos tenía un gran número de éstos, la posibilidad de la falla de un bulbo durante la operación, y la falla subsecuente de toda la computadora, era muy alta.

Durante los años 1950s y 1960s, se desarrollaron las computadoras que usaban transistores en lugar de bulbos. Los transistores son mucho más pequeños que los bulbos, pueden operar a una velocidad en el orden de nanosegundos ($ns = 10^{-9} segundos$), y no se funden. Con estos dispositivos, la computadora digital ha llegado a velocidades lo suficientemente razonables como para operar sobre programas cada vez más complejos y largos. Más aun, ya que varios programas "cortos" podían ejecutarse muy rápido, se desarrolló el uso práctico de *tiempo compartido* (*time sharing*), en el cual muchos usuarios aparentan utilizar la computadora simultáneamente. En realidad, en una computadora con tiempo compartido, solo una persona realmente utiliza la computadora en un momento dado. Sin embargo, la computadora se reparte su tiempo tan rápido entre los usuarios que ninguno de ellos nota diferencia alguna.

El desarrollo de los circuitos integrados redujo todavía más el tamaño de las computadoras, a un costo mucho más bajo. Tales circuitos integrados son dispositivos semiconductores en los cuales, mediante técnicas ópticas, muchos miles de transistores se fabrican en una oblea de silicio. Los circuitos integrados son componentes que no requieren alambrarse a mano, como era el caso de los bulbos o los transistores. Consecuentemente, el costo y mano de obra para producir una computadora digital descendieron, y la fabricación de computadoras cada vez más rápidas y baratas es hoy una realidad.

El primer circuito integrado tenía solo unos cuantos componentes como transistores y resistencias. Debido a esto, a este tipo de circuito se le conoce hoy en día como circuito SSI (*Small Scale of Integration*). Poco tiempo después, se produjeron otros circuitos con un número de 50 a 100 componentes, llamados MSI (*Medium Scale of Integration*). La fabricación de miles de componentes en un solo circuito integrado dio lugar a los sistemas LSI (*Large Scale of Integration*), lo cual representó un gran paso en el desarrollo de la computación electrónica. Finalmente, se han desarrollado a últimas fechas los circuitos integrados VLSI (*Very Large Scale of Integration*), donde decenas o cientos de miles de componentes en un solo circuito los hace lo

suficientemente poderosos como para contener un sistema de cómputo. De hecho, el *microprocesador* es un ejemplo de un circuito VLSI que representa una "computadora en un solo circuito integrado", realmente constituido por una Unidad de Control, una Unidad Lógico-Aritmética y algunos registros.

Capítulo 2

Sistemas numéricos

En el capítulo anterior se menciona que las computadoras digitales trabajan con un sistema numérico binario que utiliza los dígitos 0 y 1. En este capítulo se discuten las ideas básicas de tal sistema numérico. Además, se mencionan otros dos sistemas numéricos que realmente no se utilizan por las computadoras digitales, pero que son extremadamente convenientes para el uso de las personas que trabajan con ellas.

2.1. Ideas básicas de los sistemas numéricos

El sistema numérico más familiar para los seres humanos es el sistema decimal, que utiliza 10 símbolos o dígitos. Sin embargo, los sistema numéricos pueden formarse de cualquier número de símbolos, siempre y cuando tal número sea mayor que uno. El número de símbolos que un sistema numérico utiliza se conoce como *base* del sistema. En el presente capítulo, se consideran *(a)* el sistema *binario* o base 2, *(b)* el sistema *octal* o base 8 y *(c)* el sistema *hexadecimal* o base 16.

Para comenzar la discusión, considérese la siguiente tabla, que compara parte de los sistemas numéricos de interés. Nótese que el sistema hexadecimal requiere de 16 símbolos, por lo que por convención se utilizan las letras A, B, C, D, E y F para representar los dígitos de los valores decimales 10, 11, 12, 13, 14 y 15, respectivamente. La tabla muestra los valores numéricos de cada base entre 0 y 20 decimal.

Decimal	Binario	Octal	Hexadecimal
0	00000	0	0
1	00001	1	1
2	00010	2	2
3	00011	3	3
4	00100	4	4
5	00101	5	5
6	00110	6	6
7	00111	7	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

Para saber qué valor representa un número, es necesario primero conocer su base. Para especificar la base se hace uso de un subíndice. Por ejemplo, de la tabla anterior, se tiene que:

$$12_{10} = 1100_2 = 14_8 = C_{16}$$

Nótese que el subíndice *siempre* se escribe en base 10. Comúnmente, cuando se conoce la base que se utiliza, normalmente el subíndice puede omitirse.

Se discute a continuación algunas ideas básicas sobre los sistemas numéricos. Se comienza con la base 10, que es la más conocida, para tratar después las otras bases. También, se inicia con números enteros, ya que posteriormente se tratan los números fraccionarios.

Considérese el número 293_{10}. Es sencillo notar que este número decimal se encuentra compuesto por:

$$2 \text{ centenas} + 9 \text{ decenas} + 3 \text{ unidades}$$

Es posible expresar esto en forma más compacta utilizando las siguientes equivalencias:

$$\begin{aligned} 10^0 &= 1 \\ 10^1 &= 10 \\ 10^2 &= 10 \times 10 = 100 \\ 10^3 &= 10 \times 10 \times 10 = 1000 \\ 10^4 &= 10 \times 10 \times 10 \times 10 = 10000 \end{aligned}$$

Por tanto, se puede escribir:

$$293_{10} = (2 \times 10^2) + (9 \times 10^1) + (3 \times 10^0)$$

lo que significa que 293_{10} representa 2 centenas, 9 decenas y 3 unidades. De manera similar:

$$32864_{10} = (3 \times 10^4) + (2 \times 10^3) + (8 \times 10^2) + (6 \times 10^1) + (4 \times 10^0)$$

ó 3 decenas de millar, 2 millares, 8 centenas, 6 decenas y 4 unidades.

En cuanto al sistema numérico binario, y basándose en las mismas ideas excepto que la base es 2, se tiene que:

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 2 \times 2 = 4 \\ 2^3 &= 2 \times 2 \times 2 = 8 \\ 2^4 &= 2 \times 2 \times 2 \times 2 = 16 \\ 2^5 &= 2 \times 2 \times 2 \times 2 \times 2 = 32 \\ 2^6 &= 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64 \end{aligned}$$

Por lo tanto, se puede escribir que:

$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

Similarmente, se tiene que:

$$110101_2 = (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 53_{10}$$

En cuanto a los sistemas octal y hexadecimal, las ideas son muy similares. Utilizando las siguientes equivalencias:

$$\begin{aligned}
8^0 &= 1 \\
8^1 &= 8 \\
8^2 &= 8 \times 8 = 64 \\
8^3 &= 8 \times 8 \times 8 = 512 \\
8^4 &= 8 \times 8 \times 8 \times 8 = 4096 \\
8^5 &= 8 \times 8 \times 8 \times 8 \times 8 = 32768
\end{aligned}$$

$$\begin{aligned}
16^0 &= 1 \\
16^1 &= 16 \\
16^2 &= 16 \times 16 = 256 \\
16^3 &= 16 \times 16 \times 16 = 4096 \\
16^4 &= 16 \times 16 \times 16 \times 16 = 65536
\end{aligned}$$

Se tiene, por ejemplo, que:

$$\begin{aligned}
731_8 &= (7 \times 8^2) + (3 \times 8^1) + (1 \times 8^0) = 473_{10} \\
4132_8 &= (4 \times 8^3) + (1 \times 8^2) + (3 \times 8^1) + (2 \times 8^0) = 2138_{10} \\
1E2_{16} &= (1 \times 16^2) + (14 \times 16^1) + (2 \times 16^0) = 482_{10} \\
263A_{16} &= (2 \times 16^3) + (6 \times 16^2) + (3 \times 16^1) + (10 \times 16^0) = 9786_{10}
\end{aligned}$$

donde $E_{16} = 14_{10}$ y $A_{16} = 10_{10}$.

Esto termina con la discusión de la parte entera. A continuación, se discute cómo diferentes bases consideran la parte fraccional de un número. De nuevo, se inicia con un ejemplo en base decimal, ya que se trata del sistema numérico más familiar. La parte fraccional de un número decimal se indica mediante un *punto decimal*. Por ejemplo, el número $0{,}136_{10}$ tiene:

$$1 \text{ décima} + 3 \text{ centésimas} + 6 \text{ milésimas}$$

Las potencias en base 10 útiles aquí son:

$$\begin{aligned} 10^{-1} &= 1/10 = 0{,}1 \\ 10^{-2} &= 1/(10 \times 10) = 0{,}01 \\ 10^{-3} &= 1/(10 \times 10 \times 10) = 0{,}001 \\ 10^{-4} &= 1/(10 \times 10 \times 10 \times 10 = 0{,}0001 \\ 10^{-5} &= 1/(10 \times 10 \times 10 \times 10 \times 10) = 0{,}00001 \end{aligned}$$

De este modo, se tiene que:

$$0{,}136_{10} = (1 \times 10^{-1}) + (3 \times 10^{-2}) + (6 \times 10^{-3})$$

Un ejemplo que considera parte entera y fraccionaria es el siguiente:

$$13{,}3174_{10} = (1\times10^{1})+(3\times10^{0})+(3\times10^{-1})+(1\times10^{-2})+(7\times10^{-3})+(4\times10^{-4})$$

Considérese ahora números fraccionales en una base diferente a la base 10. El punto que separa la parte entera de la parte fraccionaria se llama punto decimal para la base 10. En el caso general, se llama *punto a la base*. En particular, en base 2 se llama *punto binario*, en base 8 se llama *punto octal*, y en base 16, *punto hexadecimal*.

Las fracciones binarias se presentan en un número binario, como por ejemplo $0{,}101_2$ tiene:

$$1 \text{ mitad} + 0 \text{ cuartos} + 1 \text{ octavo}$$

Para los números binarios fraccionales, entonces, se requiere de otro conjunto de potencias de 2:

$$\begin{aligned} 2^{-1} &= 1/2 = 0{,}5_{10} \\ 2^{-2} &= 1/(2 \times 2) = 0{,}25_{10} \\ 2^{-3} &= 1/(2 \times 2 \times 2) = 0{,}125_{10} \\ 2^{-4} &= 1/(2 \times 2 \times 2 \times 2 = 0{,}0625_{10} \\ 2^{-5} &= 1/(2 \times 2 \times 2 \times 2 \times 2) = 0{,}03125_{10} \end{aligned}$$

Y por lo tanto, se tiene que:

$$0{,}101_2 = (1 \times 2^{-1}) + (0 \times 10^{-2}) + (1 \times 10^{-3}) = 0{,}625_{10}$$

De forma similar:

$$11{,}101_2 = (1\times10^{1})+(1\times10^{0})+(1\times10^{-1})+(0\times10^{-2})+(1\times10^{-3}) = 3{,}625_{10}$$

Algo similar sucede con el sistema octal, para el cual son útiles las siguientes potencias de 8:

$$\begin{aligned} 8^{-1} &= 1/8 = 0{,}125_{10} \\ 8^{-2} &= 1/(8 \times 8) = 0{,}015625_{10} \\ 8^{-3} &= 1/(8 \times 8 \times 8) = 0{,}001953125_{10} \\ 8^{-4} &= 1/(8 \times 8 \times 8 \times 8) = 0{,}000244140625_{10} \end{aligned}$$

Así, por ejemplo:

$$0{,}213_8 = (2 \times 8^{-1}) + (1 \times 8^{-2}) + (3 \times 8^{-3}) = 0{,}271484735_{10}$$

Finalmente, las potencias de 16 útiles para el cálculo de fracciones hexadecimales son las siguientes:

$$
\begin{aligned}
16^{-1} &= 1/16 = 0{,}0{,}0625_{10} \\
16^{-2} &= 1/(16 \times 16) = 0{,}00390625_{10} \\
16^{-3} &= 1/(16 \times 16 \times 16) = 0{,}000244140625_{10} \\
16^{-4} &= 1/(16 \times 16 \times 16 \times 16) = 0{,}00001525878906_{10}
\end{aligned}
$$

De este modo, se tiene que:

$$1E3{,}0A1_{16} = 483{,}0393066406_{10}$$

2.2. Cambiando de una base a otra

Tras revisar algunos de los sistemas numéricos más importantes en la computación digital, se analiza ahora algunos de los procedimientos convenientes que pueden utilizarse para cambiar números de una base a otra. Por ejemplo, debe ser posible expresar un número binario en términos de su equivalente decimal o expresar tal número binario en términos de su equivalente octal.

Como se muestra en las igualdades anteriores, es relativamente sencillo convertir de un sistema númerico cualquiera a su equivalente en base 10. Por ahora, se presentan algunas otras conversiones, comenzando con la conversión de un número decimal a su equivalente binario. Supóngase que se tiene el número decimal 53_{10}, y se desea convertirlo en su equivalente binario. Para esto, se hace necesario obtener cuántos dígitos binarios se encuentran en la primera posición del número entero, es decir, el dígito que se multiplica por 2^0. Para hacer esto, se divide el número entre 2:

$$\frac{53}{2} = 26 + \frac{1}{2}$$

El residuo de la división (en este caso, 1) representa el dígito binario del número multiplicado por 2^0. Ahora, tomando el cociente, pero *sin considerar el residuo* (aquí, el valor 26), se divide de nuevo entre 2. Esto da como resultado el dígito que se multiplica, ahora por la segunda posición a la izquierda o el número multiplicado por 2^1:

$$\frac{26}{2} = 13 + \frac{0}{2}$$

Ya que no hay residuo, no hay un dígito que multiplique a 2^1 para la representación binaria de 53_{10}. Continuando con la división del cociente, se tiene que:

$$\frac{13}{2} = 6 + \frac{1}{2}$$

El residuo 1 indica que hay un dígito (el propio 1) que multiplica a 2^2 en la expresión binaria de 53_{10}. Al continuar, se hace:

$$\frac{6}{2} = 3 + \frac{0}{2}$$

Por tanto, no hay un dígito binario multiplicando a 2^3 para el equivalente de 53_{10}. Así, se llega a que:

$$\frac{3}{2} = 1 + \frac{1}{2}$$

Que indica que hay un dígito binario mutiplicando a 2^4. Finalmente, el procedimiento termina con la expresión:

$$\frac{1}{2} = 0 + \frac{1}{2}$$

Esto muestra que hay un dígito para 2^5 en la expresión binaria de 53_{10}. Aquí termina el procedimiento, y se obtiene por lo tanto que:

$$53_{10} = 110101_2$$

Si se compara esto con la expansión hecha anteriormente para el número binario 110101_2, se puede comprobar que es equivalente a 53_{10}. En general, se puede utilizar el procedimiento de la división para obtener el equivalente binario (octal o hexadecimal) de cualquier número decimal. Como ejemplo, exprésese 75_{10} en base 2:

$$\begin{aligned}
\frac{75}{2} &= 37 + \frac{1}{2} \\
\frac{37}{2} &= 18 + \frac{1}{2} \\
\frac{18}{2} &= 9 + \frac{0}{2} \\
\frac{9}{2} &= 4 + \frac{1}{2} \\
\frac{4}{2} &= 2 + \frac{0}{2} \\
\frac{2}{2} &= 1 + \frac{0}{2} \\
\frac{1}{2} &= 0 + \frac{1}{2}
\end{aligned}$$

El número binario se obtiene mediante listar todos los residuos en orden reverso, es decir, el primer residuo es el dígito más a la izquierda o *bit menos significativo*. Así, se tiene que:

$$75_{10} = 1001011_2$$

Hasta aquí, se ha considerado la conversión de un número entero. A contiuación, se examina cómo una fracción expresada en base 10 puede convertirse en binario. En este caso, el proceso involucra la multiplicación con la base en lugar de la división entre la base, como en el caso de los números enteros. Por ejemplo, considérese obtener el equivalente binario de $0{,}125_{10}$. Se comienza multiplicando el número por 2:

$$0{,}125 \times 2 = 0{,}25$$

La parte entera del resultado es 0, por lo que se sabe que el dígito binario que multiplica a 2^{-1} del equivalente binario es 0. Ahora, se multiplica de nuevo *solo la parte fraccional* del resultado por 2:

$$0{,}25 \times 2 = 0{,}5$$

De nuevo, la parte entera del resultado es 0, por lo que el dígito que multiplica a 2^{-2} también es 0. Multiplicando una vez mas la parte fraccional del resultado por 2, se tiene que:

$$0{,}5 \times 2 = 1{,}0$$

La parte entera del resultado es ahora 1, de tal modo que el siguiente digíto (que multiplica a 2^{-3}) es 1. Además, la parte fraccional del resultado es 0, por lo que el procedimiento puede terminarse aquí. Así se tiene que:

$$0{,}125_{10} = 0{,}001_2$$

Considérese otro ejemplo: exprésese $0{,}257_{10}$ en binario. Recuérdese que sólo la parte fraccional se multiplica cada vez por 2:

$$\begin{aligned}
0{,}257 \times 2 &= 0{,}514 \\
0{,}514 \times 2 &= 1{,}028 \\
0{,}028 \times 2 &= 0{,}056 \\
0{,}056 \times 2 &= 0{,}112 \\
0{,}112 \times 2 &= 0{,}224 \\
0{,}224 \times 2 &= 0{,}448 \\
0{,}448 \times 2 &= 0{,}896 \\
0{,}896 \times 2 &= 1{,}792 \\
0{,}792 \times 2 &= 1{,}584 \\
&\vdots
\end{aligned}$$

Por lo tanto:

$$0{,}257_{10} = 0{,}010000011\ldots_2$$

Nótese que este procedimiento se puede continuar, repitiéndose indefinidamente; esto significa que no hay una representación binaria exacta para este número. Esto es, hay un número infinito de términos a la derecha del punto binario. Así, la conversión de algunas fracciones decimales a su equivalente binario puede resultar en algunas inexactitudes ya que ni las personas ni las computadoras pueden trabajar con un número infinito de términos. Tal inexactitud se conoce como *error de redondeo* (*round-off error*). El uso de suficientes términos puede hacer que la inexactitud sea despreciable.

Ahora bien, cuando se convierte un número que consta tanto de parte entera como fraccional, la conversión de cada parte se realiza de manera

independiente y por separado. Por ejemplo, para expresar $53{,}125_{10}$ en binario, se convierten 53_{10} y $0{,}125_{10}$ como se ha mostrado anteriormente para obtener que:

$$53{,}125_{10} = 110101{,}001_2$$

Ahora bien, si se desea convertir un número decimal a su equivalente octal o hexadecimal, se utiliza el mismo procedimiento, excepto que en el caso octal la parte entera se divide entre 8 y la parte fraccionaria se multiplica por 8. Obviamente, en el caso hexadecimal, se utiliza 16. Por ejemplo, a continuación se convierte 31_{10} a base octal:

$$\begin{aligned} \frac{31}{8} &= 3 + \frac{7}{8} \\ \frac{3}{8} &= 0 + \frac{3}{8} \end{aligned}$$

Por lo tanto:

$$31_{10} = 37_8$$

En seguida, se convierte $0{,}125_{10}$ a su equivalente octal:

$$0{,}125 \times 8 = 1{,}000$$

Y por tanto:

$$0{,}125_{10} = 0{,}1_8$$

Algo similar se aplica a la conversion a base hexadecimal. Por ejemplo, 31_{10} se convierte a hexadecimal de la siguiente manera:

$$\begin{aligned} \frac{31}{16} &= 1 + \frac{15}{16} \\ \frac{1}{16} &= 0 + \frac{1}{16} \end{aligned}$$

Sin embargo, el número 15 se representa en hexadecimal mediante el símbolo F, por lo que:

$$31_{10} = 1F_{16}$$

Finalmente, se muestra cómo convertir $0{,}125_{10}$ a hexadecimal:

$$0{,}125 \times 16 = 2{,}000$$

Como no hay parte fraccional remanente, se tiene que:

$$0{,}125_{10} = 0{,}2_{16}$$

Finalmente, se considera la conversión entre bases binaria, octal y hexadecimal. Tal conversión es relativamente simple, y puede realizarse por inspección. Supóngase que se desea convertir el número octal 2637_8 a binario. Simplemente, la conversión se realiza mendiante escribir *cada dígito* del número octal como un número binario de tres dígitos. Cuando estos dígitos se escriben en el orden en que aparecen en el número octal, se obtiene el equivalente binario deseado. En el ejemplo actual:

$$\begin{aligned} 2_8 &= 010_2 \\ 6_8 &= 110_2 \\ 3_8 &= 011_2 \\ 7_8 &= 111_2 \end{aligned}$$

Entonces:

$$2637_8 = 10110011111_2$$

Este simple procedimiento también es válido para números con parte fraccional, siempre considerando dónde se encuentra el punto. Por ejemplo:

$$2637{,}126_8 = 10110011111{,}00101011_2$$

El procedimiento, por supuesto, puede usarse en forma inversa para convertir de binario a octal. Por ejemplo, supóngase que se tiene 1011_2 y se desea convertir a su equivalente octal. Primero, se añaden ceros a la izquierda del número binario a fin de que el número de dígitos en él sea un múltiplo de tres. En el ejemplo presente, esto hace que se tenga el número binario como 001011_2. En seguida, se divide el número binario en grupos de tres dígitos. Finalmente, se escribe el equivalente octal de cada uno de los grupos de tres dígitos. Así, se tiene que:

$$001011_2 = 13_8$$

El mismo método se utiliza para cualquier número binario con parte fraccional, siempre tomando en cuenta la posición del punto. Pero ahora se añaden ceros tanto a la izquierda como a la derecha del número, de tal forma que el número de dígitos hacia la izquierda y derecha del punto binario sea un múltiplo de tres. *Nótese que añadir ceros a la izquierda y derecha del número binario original no cambia su valor numérico.* Por ejemplo, conviértase $1011{,}10111_2$ a base octal. Añadiendo ceros, se tiene:

$$001011{,}101110_2 = 13{,}56_8$$

La conversión de hexadecimal a binario y de binario a hexadecimal son muy similares a las conversiones entre binario y octal, con la excepción de que ahora los grupos de dígitos binarios son de cuatro. Por ejemplo:

$$1EA{,}26B_{16} = 0001\,1110\,1010{,}0010\,0110\,1011_2$$

Nótese que las conversiones entre binario, octal y hexadecimal no están sujetas al error de redondeo.

En general, las computadoras trabajan sólo con números binarios. Sin embargo, resulta más fácil para las personas utilizar números octales o hexadecimales, ya que tienen un número menor de dígitos y la conversión de binario a octal o hexadecimal (y viceversa) es fácil de realizar utilizando un simple programa de computadora. Por otro lado, algunas computadoras se han construido de tal modo que pueden aceptar en su entrada y producir en su salida números octales y hexadecimales. Sin embargo, todo procesamiento de datos se realiza en binario. Además, en la mayoría de las aplicaciones, la programación y la entrada de datos se hace mediante números decimales. Los números octales y hexadecimales se usan cuando se requiere estudiar los números binarios dentro de la computadora. Esto es muy útil si se desea construir una computadora, y se desea probar que funciona apropiadamente. Finalmente, los números binarios y sus equivalentes octales y hexadecimales se usan también en algunos tipos de programas que se discuten más adelante.

2.3. Algo de aritmética binaria elemental

En esta sección se discuten algunas ideas sobre la adición utilizando números binarios. En un capítulo posterior se discute la aritmética binaria en mayor detalle, cuando se describe cómo la artimética se realiza por una computadora. Se discute, además, algunas restricciones resultantes cuando la aritmética se realiza por computadoras.

Ya que la base decimal es más familiar, se inicia la discusión sobre aritmética binaria elemental con un ejercicio de adición. Considérese el siguiente ejemplo:

$$\begin{array}{r} 12436 \\ +\,13253 \\ \hline 25689 \end{array}$$

Como se sabe, cada columna (unidades, decenas, centenas, unidades de millar, etc.) se suma para obtener el resultado deseado. En este ejemplo, la suma de cada columna no excede o es igual a la base. Así, no se tiene la necesidad de llevar un acarreo de una columna a la siguiente (consideradas de derecha a izquierda). Obsérvese lo que ocurre cuando existe un acarreo (es decir, el resultado excede o es igual a la base):

$$\begin{array}{r} 1578 \\ +\,2694 \\ \hline 4272 \end{array}$$

El resultado de la primer columna (de unidades) es 12, lo cual excede la base 10 en este caso; se escribe el 2, y los restantes 10 se acarrean a la siguiente columna (de decenas) mediante sumarle un 1. Tal procedimiento se repite para todas las columnas de los sumandos. Por ejemplo, es necesario añadir un 1 a la tercera columna (de centenas) para continuar.

Exactamente la misma idea básica se aplica para sumar cualquier par de números, independientemente de su base. Por ejemplo, considére se la siguiente adición octal:

$$\begin{array}{r} 1253_{8} \\ +\,4321_{8} \\ \hline 5574_{8} \end{array}$$

Es posible comprobar que la adición es correcta mediante convertir los números octales a decimales.

Ahora bien, considérese una adición octal con acarreos:

$$\begin{array}{r} 1\,4\,7\,6_{\,8} \\ +\,1\,6\,3\,4_{\,8} \\ \hline 3\,3\,3\,2_{\,8} \end{array}$$

En la tabla de la página 18 puede consultarse que $6_8 + 4_8 = 12_8 = 10_{10}$. Por tanto, se escribe el 2 en la columna de unidades y un 1 se acarrea a la siguiente columna (de 8^1). Ahora, se suman $1_8 + 7_8 + 3_8 = 13_8 = 11_{10}$. Se escribe el 3 en la segunda columna y se acarrea otro 1 a la tercera columna (de 8^2). Este procedimiento parece más complicado que la adición en base 10, pero esto se debe a la poca familiaridad con los números octales, y además, que se requiere conocer las tablas de la adición en base 8.

La adición binaria utiliza el mismo procedimiento básico. Un ejemplo que no requiere acarreo se muestra a continuación:

$$\begin{array}{r} 1\,0\,1\,1\,0_{\,2} \\ +\,0\,1\,0\,0\,1_{\,2} \\ \hline 1\,1\,1\,1\,1_{\,2} \end{array}$$

Otro ejemplo que lleva a cabo acarreo es el siguiente:

$$\begin{array}{r} 1\,0\,1\,1\,1_{\,2} \\ +\,0\,0\,0\,1\,1_{\,2} \\ \hline 1\,1\,0\,1\,0_{\,2} \end{array}$$

Considérese la primera columna. Se tiene que sumar $1_2 + 1_2 = 10_2$. El 0 se coloca en la primera columna del resultado y se acarrea el 1 a la segunda columna (de 2^1). En la segunda columna se suma el acarreo, teniendo $1_2 + 1_2 + 1_2 = 11_2$. El 1 se escribe como resultado en la segunda columna y se acarrea otro 1 a la tercera columna (de 2^2). Como puede verse, el procedimiento se repite por tantas columnas como las haya entre los dos números binarios. Nótese que las reglas de la adición binaria son exactamente las mismas que las reglas para la adición decimal.

En el siguiente ejemplo, se suman dos números binarios con parte fraccional, que igualmente siguen las reglas básicas de la adición. Nótese la alineación de los dos números respecto al punto binario:

$$\begin{array}{r} 1\,0\,1\,1\,0\,1.1\,0\,1\ _2 \\ +\,0\,0\,0\,1\,1\,0.0\,0\,1\ _2 \\ \hline 1\,1\,0\,0\,1\,1.1\,1\,0\ _2 \end{array}$$

Un problema que no sucede cuando realizamos adiciones manualmente, pero que *puede* ocurrir cuando se utiliza una computadora, se ilustra en el siguiente ejemplo:

$$\begin{array}{r} 1\,0\,1\,1\,0\,1\,1\,1\ _2 \\ +\,1\,0\,1\,1\,1\,0\,0\,1\ _2 \\ \hline 1\,0\,1\,1\,1\,0\,0\,0\,0\ _2 \end{array}$$

En este último ejemplo, los dos números que se suman tienen 8 dígitos cada uno. Un dígito binario recibe el nombre de *bit* (de *bi*nary digi*t*). Así, se dice que se han sumado dos números de 8 bits. Ahora bien, debido al acarrero de la columna de la extrema izquierda, el resultado de la suma tiene 9 bits. Esto no representa ningún problema cuando las personas suman números. Sin embargo, en las computadoras, los números deben almacenarse en dispositivos físicos llamados *registros*. Un registro puede almacenar hasta un cierto número de bits. Si el número binario que debe almacenarse tiene más bits que los que pueden almacenarse en el registro, el "exceso" de bits se pierde. Notoriamente, esto puede provocar un error muy substancial. Por ello, es importante analizar a detalle cómo tal error sucede.

Un registro que puede almacenar un número binario de 8 bits puede representarse gráficamente como se muestra en la figura 2.1 (el siguiente capítulo discute circuitos reales para la implementación de registros).

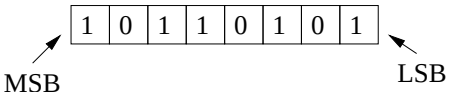

Figura 2.1: Una representación gráfica de un registro de 8 bits almacenando el número 10110101_2.

El bit a la extrema derecha se conoce con el nombre de "bit menos sig-

nificativo" (*least significant bit* o LSB) ya que tiene el valor numérico más pequeño de todos los dígitos que componen al número binario. En forma similar, el bit más a la extrema izquierda se le conoce como "bit más significativo" (*most significant bit* o MSB), debido a que tiene el mayor valor numérico dentro de número binario. Por el momento, supóngase que se trabaja con valores enteros (en el capítulo 5 se aplican las mismas ideas a números con parte fraccional).

Ahora bien, si por una operación aritmética como la adición, el resultado que se obtiene tiene más bits que los que pueden almacenarse en el registro, entonces *(a)* los bits que se retienen son los menos significativos; y *(b)* los bits más significativos se *pierden*. Por lo tanto, en el último ejemplo de adición, si el resultado se coloca en un registro de 8 bits, la respuesta sería:

$$01110000_2 = 112_{10}$$

en lugar de la respuesta correcta, que sería:

$$101110000_2 = 368_{10}$$

Como puede observarse, esto representa un error substancial e importante, conocido con el nombre de "sobrecapacidad" u *overflow*. Los constructores de computadoras, así como los usuarios de las mismas, deben entender claramente el almacenamiento de números binarios en registros para evitar este tipo de errores. En los capítulos 4 y 5 se discuten la teoría y aritmética del almacenamiento de números binarios en mayor detalle, así como las formas en que el *overflow* puede, en ocasiones, utilizarse como ayuda en el cómputo.

Capítulo 3

Elementos básicos de una computadora

La computadora digital de la actualidad se compone de un conjunto de circuitos, los cuales representan sus bloques básicos de construcción. Por tanto, entender tales circuitos básicos es un paso esencial para comprender cómo funcionan las computadoras. En el presente capítulo se supone en general qué señales digitales (formadas por conjuntos de ceros y unos) se aplican a los circuitos que se discuten. En siguientes capítulos se ve de dónde provienen tales señales.

Otro objetivo de este capítulo es introducir una forma fácil de referirse y describir los circuitos digitales. Esto no hace la discusión más breve, pero la idea es asistir al lector durante el aprendizaje de cómo conectar en conjunto los circuitos de computadora.

3.1. Notación lógica

En una computadora digital, todos los valores se representan por ceros (0's) o unos (1's). Se puede decir que un valor, en cualquier momento, es una señal (una variable física cuya magnitud cambia en el tiempo), y que tal señal es digital cuando sólo puede tener los valores de 0 ó 1. A tales valores se les conoce con el nombre de *valores lógicos*. Tal nombre proviene de una rama de las matemáticas conocida como *lógica matemática*. En tal rama, el objetivo es estudiar aquellas situaciones o hechos que son *falsos* o *verdaderos*. Respecto a las computadoras digitales, éstas se construyen para funcionar con 0's y 1's, es decir, son una implementación física de un

sistema matemático de dos valores, lo que hace que la lógica matemática tenga una gran aplicación e importancia en el diseño de computadoras. Sin embargo, para ello no es necesario considerar todos los detalles de la lógica matemática. Basta considerar algunas ideas simples que ayudan a entender de forma fácil la circuitería de una computadora digital.

Inicialmente, se requiere definir qué es una *variable binaria*. Esto no es otra cosa que una variable que puede tener un valor 0 ó 1, es decir, en un momento dado, tal variable puede "contener" como valor un 0 o un 1. Las variables binarias se utilizan para representar los valores de las señales digitales dentro de una computadora. Nótese la razón por lo que esto es conveniente. En la Figura 3.1 se muestra un simple interruptor. En una computadora, tal interruptor se construye utilizando transistores u otros dispositivos semiconductores. Sin embargo, para los propósitos actuales y por simplicidad, se muestra aquí como un simple interruptor. Nótese que en la figura, la letra A representa una variable binaria: cuando $A = 0$, el interruptor se encuentra abierto, y cuando $A = 1$, el interruptor se cierra. El símbolo a la izquierda del diagrama es una bateria con voltaje V, y el símbolo a la derecha del diagrama es una resistencia eléctrica R. Es conveniente pensar en ella como un pequeño foco.

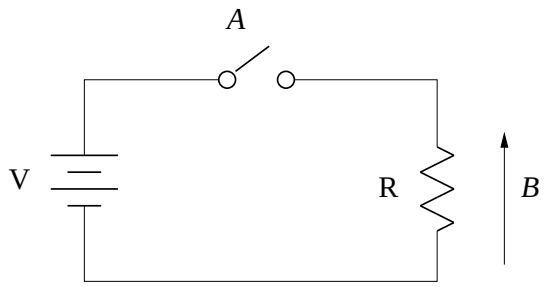

Figura 3.1: Un circuito lógico simple.

Comúnmente, al circuito de la Figura 3.1 se le conoce como *circuito lógico*, donde B representa la caída de voltaje en la resistencia. Cuando el interruptor está abierto, $B = 0$. Cuando el interruptor se cierra, el voltaje de la bateria se presenta en la resistencia (el foco se enciende). Se considera que esto corresponde a $B = 1$. Por lo tanto, cuando $A = 1$, entonces $B = 1$; y cuando $A = 0$, entonces $B = 0$. Se puede escribir esto mediante la ecuación:

$$A = B$$

Otra forma de describir este comportamiento del circuito lógico es mediante una *tabla de verdad*. En tal tabla, se listan los valores que va tomando la salida B para todos los posible valores de entrada de A. Para el circuito lógico de la Figura 3.1 se tiene que:

A	B
0	0
1	1

B se conoce como *variable dependiente*, ya que su valor depende del valor de A. Complementariamente, A se conoce como *variable independiente*, pues su valor no depende de ninguna otra variable.

Considérese ahora el circuito de la Figura 3.2, que es un poco más complicado que el anterior. Recuérdese que los interruptores están cerrados cuando su valor es 1, y se encuentran abiertos cuando su valor es 0.

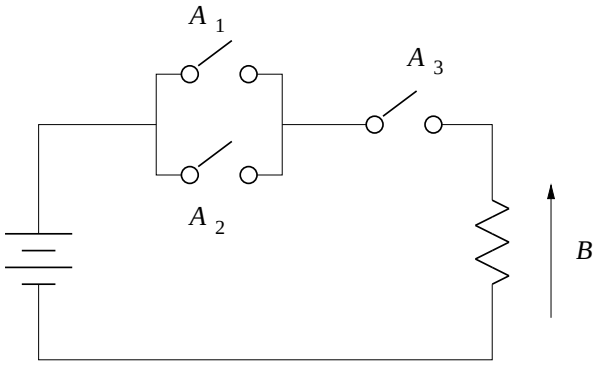

Figura 3.2: Otro circuito lógico.

Para este circuito, B toma el valor de 1 si A_1 y A_3 son ambos 1, ó si A_2 y A_3 son ambos 1, ó si A_1, A_2 y A_3 son todos 1. La tabla de verdad es ahora una forma conveniente de mostrar esta información. La columna de salida B muestra los valores que tal variable puede presentar para todas las combinaciones posibles de las variables independientes A_1, A_2 y A_3.

A_1	A_2	A_3	B
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	0
1	1	1	1

Para finalizar, es importante mencionar que en esta sección se consideran tan solo algunas formas de representar circuitos lógicos. En realidad, los circuitos que se han discutido se conocen como *circuitos combinacionales* o *lógica combinacional*. En la siguiente sección se continúa la discusión de algunos circuitos combinacionales que forman los bloques básicos de construcción de computadoras.

3.2. Compuertas

Los circuitos lógicos digitales se encuentran disponibles en forma de *circuitos integrados* (o *chips*), de tal modo que pueden utilizarse para construir cualquier tipo de circuitos lógicos. En las siguientes secciones se muestra cómo ciertos circuitos básicos, genéricamente llamados *compuertas lógicas* (*logic gates*) o simplemente *compuertas* (*gates*), se usan para construir en la práctica varios dispositivos lógicos. Tales compuertas realizan algún tipo de operación lógica. Por tanto, se presenta la operación y el circuito que la representa y realiza.

3.2.1. La compuerta AND

Una de las más comunes e importantes operaciones lógicas se ilustra en la Figura 3.3, la cual muestra dos interruptores conectados mediante un circuito en serie. Nótese que $B = 1$ sólo en el caso en que tanto el interruptor A_1 como el interruptor A_2 tengan ambos un valor de 1. Es por esto que a este circuito se le conoce con el nombre de *circuito AND*.

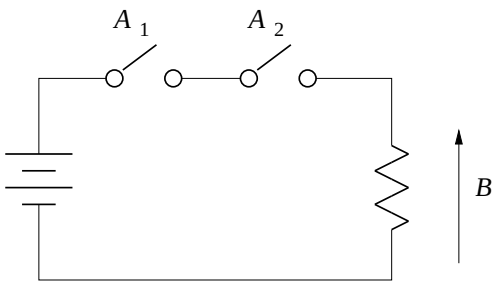

Figura 3.3: Un circuito AND con interruptores.

La tabla de verdad de la operación lógica AND se muestra como sigue:

A_1	A_2	B
0	0	0
0	1	0
1	0	0
1	1	1

Con la finalidad de obtener una representación más breve, se utilizan algunos símbolos especiales entre las variables para indicar operaciones lógicas. En el caso de la operación AND, comúnmente se utiliza un punto ($\cdot$) entre las variables. De tal modo, la operación de la Figura 3.3 puede representarse mediante la siguiente ecuación:

$$B = A_1 \cdot A_2$$

Muy frecuentemente, el punto se omite. Es por ello que la siguiente expresión es equivalente:

$$B = A_1 A_2$$

Actualmente, existen algunas compuertas semiconductoras que han sido construidas para realizar la operación lógica AND; si la compuerta realiza la operación AND, entonces se le conoce como *compuerta AND*. Más aun, en lugar de utilizar interruptores para representar compuertas, se utilizan algunos símbolos o diagramas lógicos, que son más pequeños y fáciles de dibujar. El símbolo para la compuerta AND se muestra en la Figura 3.4.

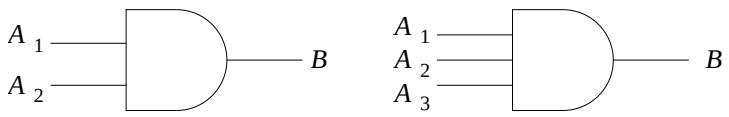

Figura 3.4: Símbolos para la compuerta AND de dos y tres entradas.

Nótese que inicialmente se muestra una compuerta AND con dos entradas. En realidad, las compuertas AND pueden construirse con muchas más entradas (o compuertas AND pueden conectarse entre sí se modo que se obtenga un circuito lógico cuya salida sea equivalente a una compuerta AND de varias entradas). La Figura 3.4 muestra también una compuerta AND de tres entradas. La variable de salida B tiene como valor 1 sólo si las variables de entrada A_1, A_2 y A_3 tienen todas valor de 1.

3.2.2. La compuerta OR

La operación OR, que consiste en dos interruptores conectados en paralelo, se muestra en la Figura 3.5. Aquí, B tiene valor de 1 si A_1 es 1, ó si A_2 es 1, ó si ambos A_1, A_2 son 1.

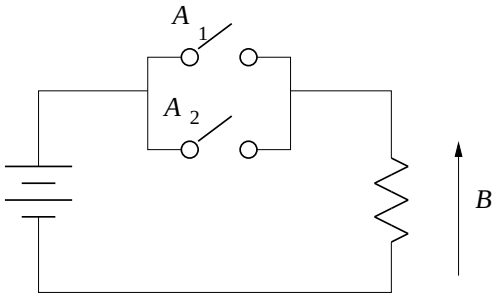

Figura 3.5: Un circuito de interruptores para la operación lógica OR.

La tabla de verdad de la operación lógica OR es como sigue:

A_1	A_2	B
0	0	0
0	1	1
1	0	1
1	1	1

El símbolo usado para designar la operación lógica OR en forma de ecuación es +. Por tanto, para el circuito de la Figura 3.5, se tiene que:

$$B = A_1 + A_2$$

Nótese que cuando las ecuaciones lógicas se escriben, los símbolos que utilizan son los mismos que se usan en aritmética ordinaria. Sin embargo *su significado no es el mismo.*

La Figura 3.6 muestra los símbolos para una compuerta OR de dos y cuatro entradas.

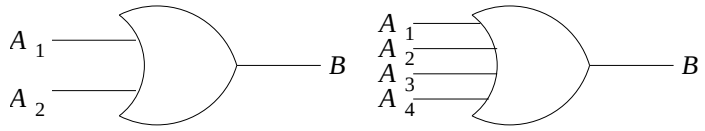

Figura 3.6: Símbolos para la compuerta OR.

Para la compuerta OR de cuatro entradas, se tiene la siguiente ecuación:

$$B = A_1 + A_2 + A_3 + A_4$$

En este caso, B tiene el valor de 1 si A_1 ó A_2 ó A_3 ó A_4, en cualquier combinación, es 1.

Un ejemplo de circuitos construidos con compuertas se muestra en la Figura 3.7.

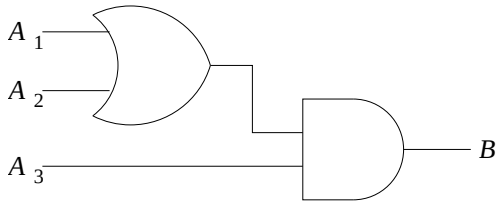

Figura 3.7: Un circuito con compuertas para el circuito de la Figura 3.2.

Nótese que la variable de salida B tiene valor 1 si A_1 ó A_2 tienen valor 1, y además si A_3 tiene valor 1. De hecho, el circuito de la Figura 3.7 representa mediante un circuito con compuertas la función del circuito con interruptores de la Figura 3.2.

La ecuación lógica que representa a ambos circuitos es:

$$B = (A_1 + A_2)A_3$$

El paréntesis en esta expresión se interpreta de la siguiente forma: todos los términos o variables dentro del paréntesis se tratan como una sola variable en relación con los términos o variables fuera del paréntesis. Por tanto, B tiene valor 1 si A_1 ó A_2 da como resultado 1, y este resultado y A_3 da como resultado 1.

3.2.3. La compuerta NOT

Considérese ahora la operación lógica llamada *negación* o *complemento*. Si la variable A tiene como valor 1, su negación es 0; si A tiene valor 0, su negación es 1. Esto significa que cuanto se obtiene la negación o complemento de una variable, ésta cambia su valor de 0 a 1 o viceversa. El símbolo que se utiliza para la negación es una comilla ($'$). Por lo tanto, si B es la negación de A, se escribe la ecuación:

$$B = A'$$

La compuerta que realiza la negación de una varibale se conoce como compuerta NOT. Su diagrama lógico se presenta en la Figura 3.8.

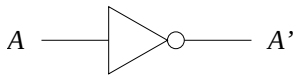

Figura 3.8: El diagrama lógico de la compuerta NOT.

Nótese que las compuertas NOT tienen tan solo *una* entrada. El pequeño círculo en el diagrama de la compuerta NOT se utiliza frecuentemente para representar la operación de negación. Por ejemplo, para la Figura 3.9, se tiene la ecuación:

$$B = A_1 A_2{}'$$

Efectivamente, el pequeño círculo representa una compuerta NOT que toma como entrada la variable A_2. Sin embargo, el símbolo del pequeño círculo nunca se utiliza solo, sino en conjunto con otras compuertas, como se muestra a continuación.

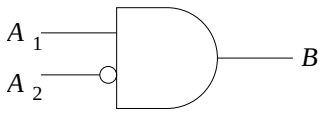

Figura 3.9: Un circuito lógico que realiza la operación $B = A_1 A_2{}'$.

3.2.4. La compuerta NOR

La operación lógica NOR consiste en realizar una operación OR y negar el resultado. Se define mediante la ecuación:

$$B = (A_1 + A_2)'$$

La tabla de verdad para una operación NOR es:

A_1	A_2	B
0	0	1
0	1	0
1	0	0
1	1	0

Nótese que para esta operación, la variable de salida B es tan solo la negación de la operación OR de las variables de entrada.

El diagrama lógico de la compuerta NOR se muestra en la Figura 3.10.

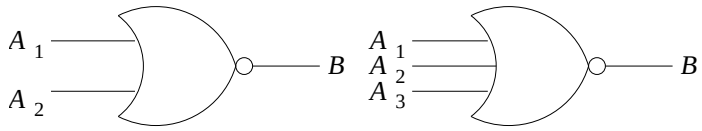

Figura 3.10: El diagrama lógico de la compuerta NOR.

También se muestra en la Figura 3.10 una compuerta NOR de tres entradas, cuya ecuación lógica es:

$$B = (A_1 + A_2 + A_3)'$$

Para la compuerta NOR, la variable de salida B tiene un valor de 0 cuando A_1, A_2, A_3, o cualquiera de sus combinaciones entre sí tiene valor

1. De hecho, las compuertas NOR, así como las compuertas NAND que se discuten a continuación, puede construirse de una manera muy sencilla como semiconductores en forma de circuitos integrados, siendo su uso en general y en la práctica muy diseminado. Además, toda función lógica puede construirse a partir de compuertas NOR y NAND, como se discute más adelante.

3.2.5. La compuerta NAND

La operación lógica NAND consiste en realizar una operación AND y negar el resultado. Se define mediante la ecuación siguiente:

$$B = (A_1 A_2)'$$

La tabla de verdad de la operación NAND es como sigue:

A_1	A_2	B
0	0	1
0	1	1
1	0	1
1	1	0

Nótese que el valor de la variable B es tan solo la negación de la operación AND entre las variables de entrada A_1 y A_2. El diagrama lógico para una compuerta NAND se muestra en la Figura 3.11

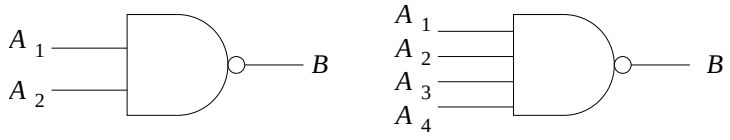

Figura 3.11: El diagrama lógico de la compuerta NAND.

Como se muestra en la Figura 3.11 y es el caso de la mayoría de las compuertas, la compuerta NAND puede contar con más de dos entradas. La compuerta NAND de cuatro entradas que se muestra tiene la ecuación lógica:

$$B = (A_1 A_2 A_3 A_4)'$$

Nótese que B tiene valor 0 sólo si A_1, A_2, A_3 y A_4 tienen todas valor 1.

3.2.6. La compuerta XOR

La operación lógica OR exclusiva, que se conoce con el nombre de XOR, es similar a la operación OR excepto que, para el caso de dos entradas, la salida B tiene valor 0 cuando *ambas* variables de entrada A_1 y A_2 tienen valor 1. El símbolo dado a la operación XOR es $\oplus$. Por lo tanto, la ecuación para la operación XOR se escribe como:

$$B = A_1 \oplus A_2$$

La tabla de verdad de la operación XOR es como sigue:

A_1	A_2	B
0	0	0
0	1	1
1	0	1
1	1	0

El diagrama lógico de la compuerta XOR se muestra en la Figura 3.12.

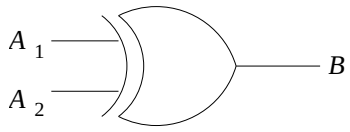

Figura 3.12: El diagrama lógico de la compuerta XOR.

Cuando una compuerta XOR tiene más de dos entradas, el resultado a la salida es algo más complicado. Supóngase que se tiene la siguiente ecuación:

$$B = A_1 \oplus A_2 \oplus A_3$$

Es posible re-escribirla de la siguiente manera:

$$B = (A_1 \oplus A_2) \oplus A_3$$

De este modo, es posible obtener la operación XOR entre A_1 y A_2, a su resultado, hacer de nuevo una operación XOR con A_3. De hecho, la agrupación de las dos primeras variables puede considerar cualquier combinación de entre dos variables de las tres existentes. El resultado de la operación XOR con tres entradas se muestra en la siguiente tabla de verdad:

A_1	A_2	A_3	$A_1 \oplus A_2$	B
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

Nótese que si $A_1 = 1$, $A_2 = 1$, y $A_3 = 1$, entonces $A_1 \oplus A_2 \oplus A_3 = 1$, y no 0, como podría haberse esperado.

Hasta ahora, se han mostrado una serie de compuertas que operan únicamente entre variables de entrada y salida. En realidad, las compuertas de los circuitos digitales integrados tienen otras variables, que físicamente son terminales o conexiones. De tal modo, las señales lógicas consisten de voltajes que se aplican (en el caso de las entradas) o se miden (en el caso de las salidas) entre pares de terminales. Una de tales terminales es llamada en circuitos eléctricos la *tierra* (*ground*, o GND), la cual es común para todas las entradas y salidas. Obsérvese que en los diagramas lógicos no se muestra tal conexión, ya que tales diagramas se presentan de una manera únicamente convencional.

Sin embargo, a fin de que las compuertas (o cualquier circuito digital integrado) funcione apropiadamente, debe conectarse a un voltaje directo, como el que proporciona una bateria o una fuente de voltaje. Las conexiones a la fuente de voltaje también se requieren como parte del circuito integrado, pero tampoco se muestran en los diagramas lógicos. Nótese que el voltaje directo que debe aplicarse debe tener una magnitud determinada (para la mayoría de los circuitos digitales integrados, tal valor es de 5 V). Si no es así, el circuito podría no funcionar apropiadamente si el voltaje de alimentación es muy pequeño, o podría quemarse si es muy grande.

3.3. Interconexión de compuertas para obtener otras compuertas

Todas las compuertas que se han mencionado hasta ahora son utilizadas cuando se diseñan e implementan circuitos lógicos para una computadora digital. Sin embargo, comúnmente en la práctica no se requiere que todos

los tipos de compuertas sean físicamente construidos, ya que la operación de algunas compuertas puede obtenerse a partir de interconectar otras compuertas más pequeñas y sencillas de implementar. Por ejemplo, considérese las interconexiones que se muestran en la Figura 3.13.

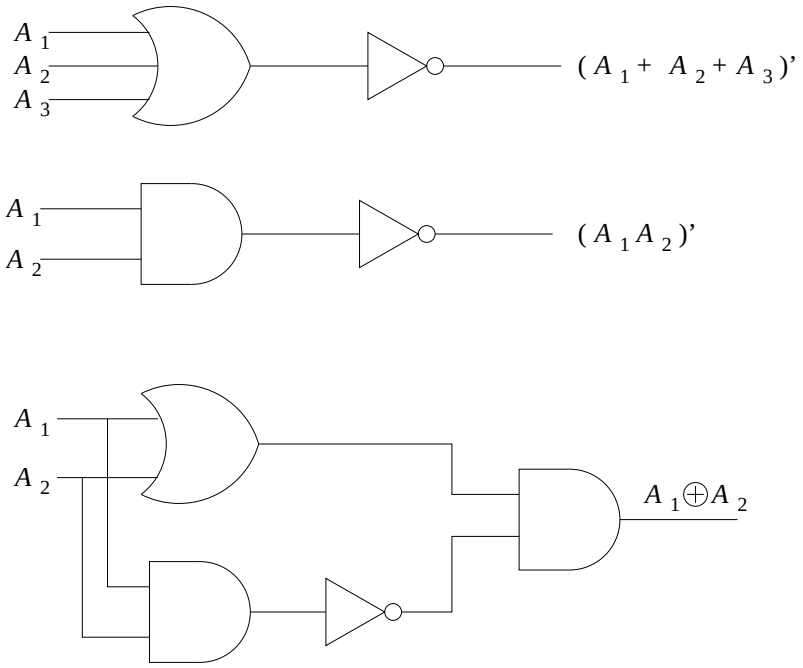

Figura 3.13: Interconexiones entre compuertas AND, OR y NOT para obtener compuertas NOR, NAND y XOR.

Nótese que las operaciones NOR y NAND son tan solo negaciones de las operaciones OR y AND, respectivamente. Por tanto, tales operaciones lógicas se obtienen mediante aplicar una compuerta NOT a la salida de las compuertas OR o AND, según sea el caso.

Sin embargo, la implementación de la compuerta XOR es algo más complicada. De hecho, es necesario hacer un cierto análisis para demostrar que la interconexión $(A_1 + A_2)(A_1A_2)'$ es equivalente a la operación XOR. La forma más sencilla es escribir la tabla de verdad para esta interconexión:

A_1	A_2	$A_1 + A_2$	A_1A_2	$(A_1A_2)'$	$(A_1 + A_2)(A_1A_2)'$	$A_1 \oplus A_2$
0	0	0	0	1	0	0
0	1	1	0	1	1	1
1	0	1	0	1	1	1
1	1	1	1	0	0	0

Comparando las dos últimas columnas de la tabla, es claro que ambas representan la misma operación. Por lo tanto, $(A_1 + A_2)(A_1A_2)' = A_1 \oplus A_2$, por lo que el tercer diagrama lógico de la Figura 3.13 es equivalente a una operación XOR.

En realidad, las compuertas que son más fáciles de implementar utilizando tecnología de semicondutores son las compuertas NOR y NAND. De hecho, todas las operaciones lógicas pueden implementarse mediante una interconexión que utiliza únicamente compuertas NOR o únicamente compuertas NAND.

La Figura 3.14 muestra las interconexiones de compuertas NOR para obtener compuertas NOT, OR y AND.

Una compuerta NOT se obtiene mediante conectar las dos entradas de una compuerta NOR. Así, si la variable de entrada A_1 tiene valor 1, la salida de la compuerta NOR es 0; si la variable de entrada A_1 tiene valor 0, la salida de la compuerta NOR es 1. Por tanto, la salida es efectivamente la negación de la entrada, lo que equivale a una compuerta NOT.

La segunda interconexión de compuertas NOR de la Figura 3.14 implementa la operación de una compuerta OR. Si se realiza la negación dos veces, se obtiene la variable original, es decir:

$$(A')' = A$$

De este modo, la salida de la compuerta NOR se niega con otra compuerta NOR conectada como compuerta NOT. Así, negando la salida de la NOR, se obtiene una compuerta OR.

La tercera interconexión de la Figura 3.14 representa una compuerta AND. Dado que se trata de una interconexión más complicada, se recurre a una tabla de verdad para comprobar su operación:

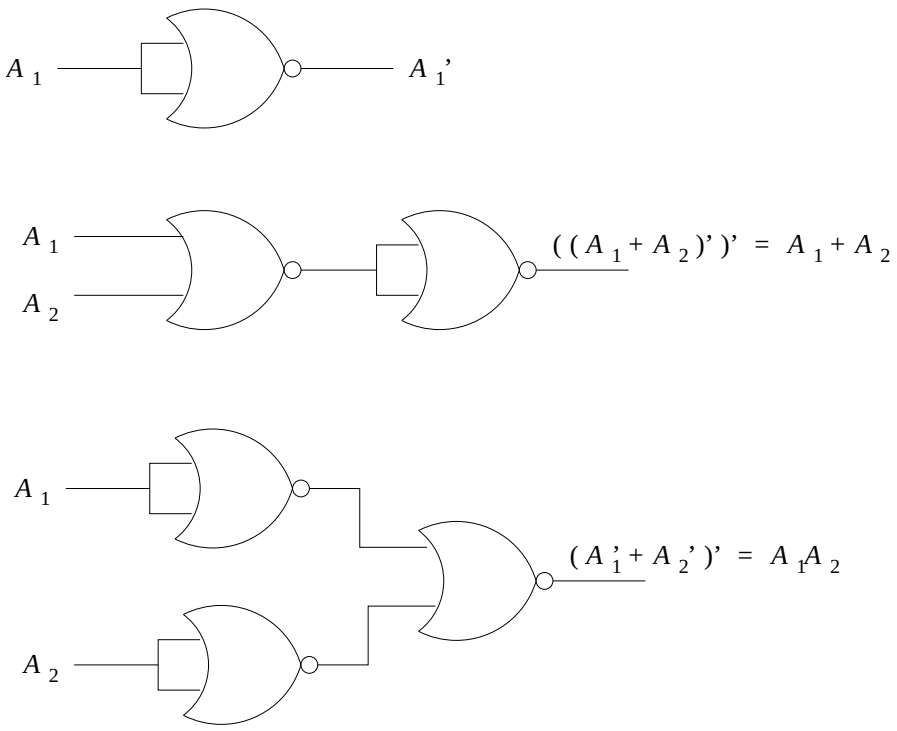

Figura 3.14: Interconexiones entre compuertas NOR para obtener compuertas NOT, OR y AND.

A_1	A_2	$A_1{}'$	$A_2{}'$	$A_1{}' + A_2{}'$	$(A_1{}' + A_2{}')'$	A_1A_2
0	0	1	1	1	0	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	0	1	1

Cuando se comparan las dos últimas columnas, es notorio que:

$$(A_1' + A_2')' = A_1A_2$$

por lo que tal interconexión representa una operación AND.

De forma similar, se muestra a continuación cómo las compuertas NOT, AND y OR pueden obtenerse mediante una interconexión de compuertas NAND, como se muestra en la Figura 3.15.

La primera inteconexión de la Figura 3.15 muestra la implementación de una compuerta NOT mediante una compuerta NAND cuyas entradas han sido conectadas juntas. Por tanto, cuando la variable de entrada A_1 tiene valor 1, la salida de la compuerta NAND es 0. Similarmente, si la variable de entrada A_1 tiene valor 0, la salida de la compuerta NAND es 1. Por tanto, se obtiene una operación NOT.

La segunda interconexión de la Figura 3.15 representa una operación AND. Esto es notorio debido a que la operación NAND es solamente la negación de la operación AND. De tal modo, al obtener la negación de la negación, se obtiene la operación deseada AND.

Finalmente, se puede mostrar que la tercera interconexión de la Figura 3.15 es una operación OR mediante la siguiente tabla de verdad:

A_1	A_2	$A_1{}'$	$A_2{}'$	$A_1{}'A_2{}'$	$(A_1{}'A_2{}')'$	$A_1 + A_2$
0	0	1	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	0	1	1

Al comparar las últimas dos columnas, es notorio que:

$$(A_1'A_2')' = A_1 + A_2$$

lo que indica que tal interconexión representa una operación OR.

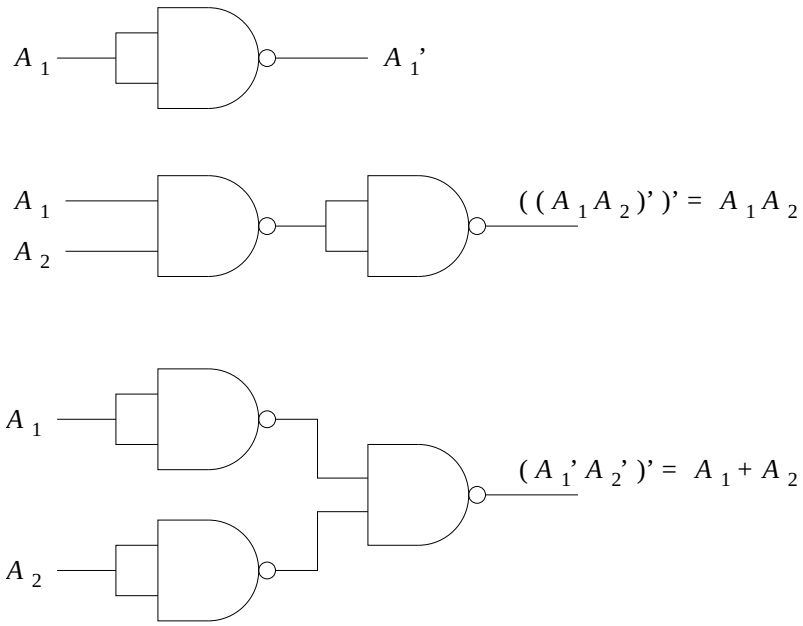

Figura 3.15: Interconexiones entre compuertas NAND para obtener compuertas NOT, AND y OR.

3.4. El sumador

Tras una breve introducción a los principales tipos de compuertas lógicas, en esta sección se muestra cómo las compuertas lógicas pueden interconectarse de modo que se produzca un circuito digital que realice una suma binaria. Supóngase que se desea sumar dos números de un solo bit, a_1 y b_1. La suma se expresa de la siguiente forma:

$$\begin{array}{r} a_1 \\ + \quad b_1 \\ \hline c_1 \; s_1 \end{array}$$

donde s_1 y c_1 son a la vez números de un solo bit, siendo s_1 el dígito *suma* y c_1 el dígito *acarreo*. La tabla de verdad de esta operación se muestra a continuación:

a_1	b_1	s_1	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Nótese que los valores binarios de s_1 y c_1 se obtienen siguiendo las reglas de la suma binaria. Considerando s_1 y c_1 separadamente, se construye a continuación un circuito lógico que cumpla con lo expresado en la tabla de verdad. Tal circuito lógico se conoce con el nombre de *sumador medio* (*half-adder*), por razones que se discuten más adelante.

Considérese la variable c_1. El circuito debe ser tal que produzca un valor 1 como salida sólo cuando a_1 y b_1 son ambos 1, es decir:

$$c_1 = a_1 b_1$$

Es sencillo notar que que c_1 se obtiene de aplicar la operación AND entre a_1 y b_1, por lo que se utiliza una compuerta AND que cumple con tal funcionalidad.

Ahora bien, considérese la variable s_1. Tal variable tiene valor de 1 en dos renglones de la tabla de verdad, de tal modo que hay dos conjuntos de entradas que producen un valor de 1 para s_1. Un conjunto de tales entradas

sucede cuando $a_1 = 0$ y $b_1 = 1$. El otro conjunto sucede cuando $a_1 = 1$ y $b_1 = 0$. De este modo, se tiene la siguiente expresión para s_1:

$$s_1 = a_1{}'b_1 + a_1b_1{}'$$

De acuerdo con esta ecuación, se desea que s_1 tenga valor 1:

- cuando $a_1 = 0$ y $b_1 = 1$

  ó

- cuando $a_1 = 1$ y $b_0 = 1$

La Figura 3.16 muestra la interconexión de compuertas para obtener las variables de salida s_1 y c_1 a partir de las variables de entrada a_1 y b_1.

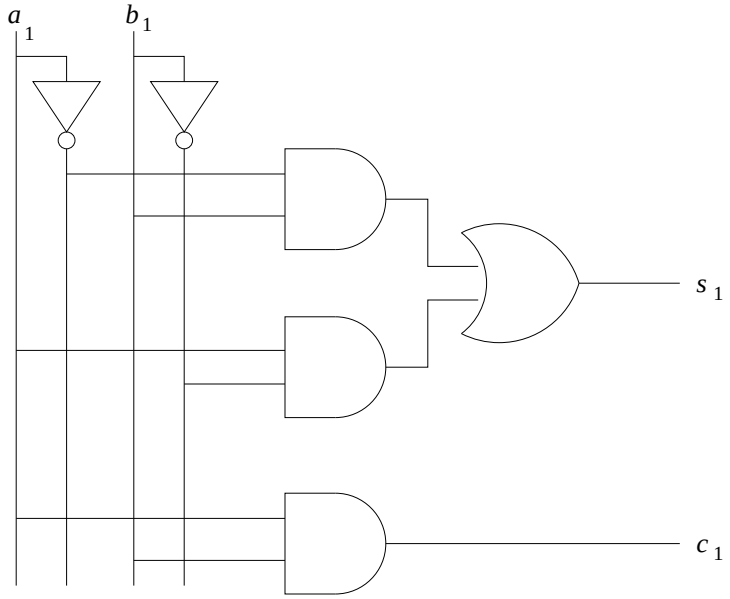

Figura 3.16: Interconexiones de compuertas para obtener un sumador medio.

El razonamiento detrás de esta interconexión se basa en considerar que la variable de salida s_1 tiene valor 1, primero, si $a_1 = 0$ y $b_1 = 1$. Pensando en términos positivos, se sabe que si $a_1 = 0$, implica que $a_1{}' = 1$. Ahora bien, $a_1{}'b_1$ tiene valor de 1 cuando $a_1{}'$ AND b_1 tiene valor 1. De hecho, esto es similar que decir $a_1 = 0$ y $b_1 = 1$, que es el punto inicial de este análisis.

De forma similar, s_1 tiene valor 1 si $a_1 = 1$ y $b_1 = 0$. De nuevo, pensando en términos positivos, se sabe que si $b_1 = 0$, implica que ${b_1}' = 1$. Ahora bien, $a_1{b_1}'$ tiene valor de 1 cuando a_1 AND ${b_1}'$ tiene valor 1.

La expresión para s_1 consiste de una operación OR aplicada entre ${a_1}'b_1$ y $a_1{b_1}'$. Este razonamiento da como resultado el circuito de la Figura 3.16.

El sumador medio puede utilizarse para sumar únicamente dos bits. Sin embargo, considérese ahora que se desea sumar dos valores binarios de más de un bit. Parecería que cada par de bits dentro de los valores binarios pudieran ser sumados utilizando sumadores medios entre ellos. Este no es el caso, ya que para sumar dos valores binarios de más de un bit siguiendo las reglas de la adición, es necesario considerar el acarreo que resulta de la suma de los dos bits anteriores. Por lo tanto, es necesario que cada sumador de dos bits considere tales valores y además el acarreo que se produce de la suma de los bits en la columna anterior. Un sumador que toma en cuenta tal acarrero se conoce como *sumador completo* (*full-adder*). De este modo, para sumar dos valores binarios de más de un bit, se utilizan sumadores completos para cada par de bits. Por ejemplo, si se desea sumar dos números de 8 bits cada uno, son necesarios 8 sumadores completos.

Considérese la operación de un sumador completo. Supóngase que tal sumador adiciona los j-ésimos bits de dos valores binarios. Se tiene, entonces, que:

$$\begin{array}{rr} & c_{j-1} \\ + & a_j \\ & b_j \\ \hline c_j & s_j \end{array}$$

Nótese que c_{j-1} es el acarreo de la columna anterior $j - 1$, que es la columna inmediatamente a la derecha de la columna de bits que se suma aquí. De nuevo, utilizando las reglas de la adición binaria, se tiene la siguiente tabla de verdad para el sumador completo:

a_j	b_j	c_{j-1}	s_j	c_j
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Mediante el razonamiento utilizado para obtener el circuito del sumador medio, se obtiene el circuito de la Figura 3.17.

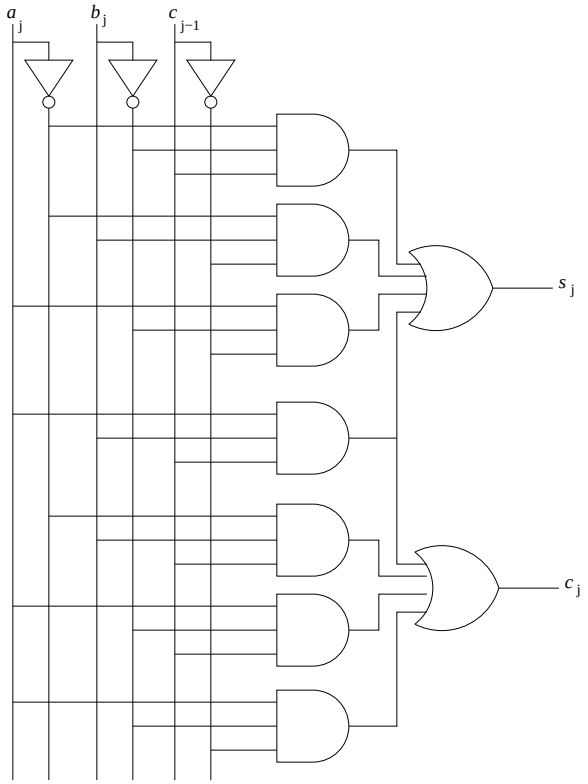

Figura 3.17: Interconexiones de compuertas para obtener un sumador completo.

Cada 1 en la columna de s_j resulta en una compuerta AND en el circuito lógico para s_j, Similarmente, cada 1 en la columna de c_j resulta en una compuerta AND en el circuito lógico para c_j. De este modo, se pueden escribir las ecuaciones para ambas variables de salida del sumador completo:

$$\begin{aligned} s_j &= a_j{}'b_j{}'c_{j-1} + a_j{}'b_jc_{j-1}{}' + a_jb_j{}'c_{j-1}{}' + a_jb_jc_{j-1} \\ c_j &= a_j{}'b_jc_{j-1} + a_jb_j{}'c_{j-1} + a_jb_jc_{j-1}{}' + a_jb_jc_{j-1} \end{aligned}$$

En la Figura 3.17, s_j se implementa mediante cuatro compuertas AND, cada una con tres entradas, cuyas salidas se conectan a la vez a una compuerta OR de cuatro entradas. Similarmente, c_j requiere de cuatro compuertas AND conectadas a una compuerta OR. Sin embargo, nótese que en la figura sólo se utilizan siete compuertas AND, dado que s_j y c_j pueden compartir la salida de una compuerta AND. La intención aquí es reducir el número de compuertas que se conectan en el circuito.

3.5. El multiplexor

En esta sección, se discute otro circuito lógico útil llamado *multiplexor*, que se trata de un interruptor controlado digitalmente. La Figura 3.18 muestra el diagrama de bloque de un multiplexor con cuatro entradas: a_0, a_1, a_2 y a_3, y una salida b.

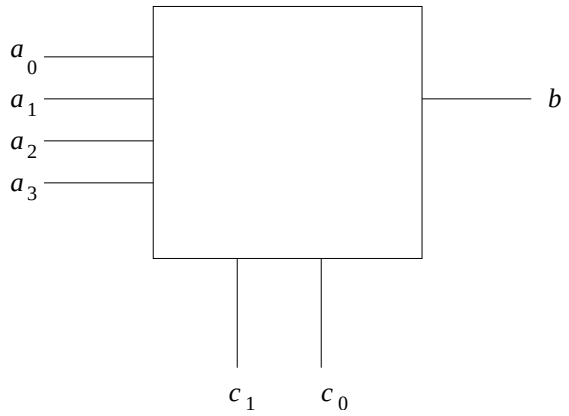

Figura 3.18: Diagrama de bloque de un multiplexor con cuatro entradas.

La función del multiplexor es conectar una de las variables de entrada a la variable de salida. La variable de entrada se selecciona mediante un

valor digital que se coloca en las terminales c_1 y c_0. La selección se hace de la siguiente forma: c_1 y c_0 se consideran bits independientes, pero en conjunción, se pueden ver como un valor binario de dos bit, siendo c_1 el bit más significativo y c_0 el bit menos significativo. Por ejemplo, si $c_1 = 1$ y $c_0 = 0$, entonces el valor de la selección es $10_2 = 2_{10}$. Si $c_1 = 1$ y $c_0 = 1$, entonces el valor de la selección es $11_2 = 3_{10}$. Ahora bien, el valor base 10 del valor binario c_1c_0 indica el subíndice de la entrada que se conecta a b. Por ejemplo, si $c_1 = 1$ y $c_0 = 1$ ($11_2 = 3_{10}$), entonces:

$$b = a_3$$

Es decir, si $a_3 = 1$, entonces b toma el valor de 1; si $a_3 = 0$, entonces b toma el valor de 0.

Puede obtenerse una expresión lógica que describe la operación del multiplexor de cuatro entradas:

$$b = c_1{}'c_0{}'a_0 + c_1{}'c_0a_1 + c_1c_0{}'a_2 + c_1c_0a_3$$

Esta expresión muestra la operación OR entre cuatro operaciones AND. Debido a la operación OR, si cualquiera de las AND da como resultado 1, entonces b toma el valor 1. Supóngase que $c_1 = 0$ y $c_0 = 0$. Esto da como resultado los las siguientes operaciones en cada AND:

$$\begin{aligned} c_1{}'c_0{}'a_0 &= 1 \cdot 1 \cdot a_0 = a_0 \\ c_1{}'c_0a_1 &= 1 \cdot 0 \cdot a_1 = 0 \\ c_1c_0{}'a_2 &= 0 \cdot 1 \cdot a_2 = 0 \\ c_1c_0a_3 &= 0 \cdot 0 \cdot a_3 = 0 \end{aligned}$$

De tal modo, para $c_1 = 0$ y $c_0 = 0$, se tiene que:

$$b = a_0$$

Nótese que en este caso la salida b depende del valor de a_0: si $a_0 = 1$, entonces $b = 1$; si $a_0 = 0$, entonces $b = 0$.

A partir de la ecuación que describe la operación del multiplexor de cuatro entradas, se puede construir un multiplexor utilizando compuertas lógicas, como se muestra en la Figura 3.19.

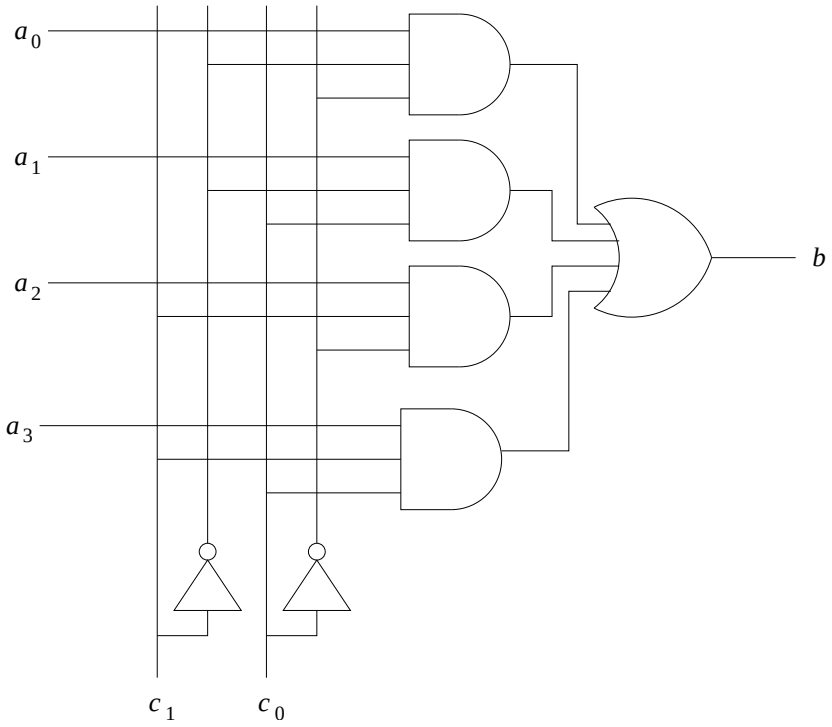

Figura 3.19: Un multiplexor con cuatro entradas.

Los multiplexores se utilizan en circuitos digitales del mismo modo que los interruptores se utilizan en circuitos eléctricos ordinarios. Dado que pueden servir para la conexión de la salida con varias probables entradas, se utilizan comúnmente para la selección de información binaria.

3.6. Flip-flops

Hasta este punto, los circuitos digitales que se han discutido son conocidos bajo la denominación de *circuitos lógicos combinacionales*, en los cuales el valor de la variable de salida, en cualquier momento, depende sólo de los valores de las variables de entrada disponibles *en ese momento*. De este modo, estos circuitos no tienen *memoria*, es decir, el valor de las variables de salida no dependen de los valores en el pasado de las variables de entrada. El valor de la variable salida, entonces, se dice que es una *combinación lógica* únicamente de los valores actuales de las variables de entrada.

En esta sección se inicia la descripción de circuitos digitales que sí cuentan con una capacidad de memoria. Tales circuitos se conocen con el nombre de *circuitos lógicos secuenciales*. Particularmente, en esta sección se discuten algunos circuitos secuenciales básicos llamados *flip-flops*, los cuales representan importantes bloques básicos de construcción de muchos otros circuitos secuenciales de mayor tamaño.

Un *flip-flop* o *multivibrador biestable* es un circuito cuya variable de salida permanece con un valor de 0 ó 1 hasta que uno o más valores binarios se apliquen a sus terminales de entrada, en cuyo caso el valor de la variable de salida cambia *en el siguiente tiempo*. Por ejemplo, si la salida tiene valor 0, permanece en tal valor hasta que los valores binarios adecuados se apliquen a sus entradas, causando su cambio a un valor de 1. En forma similar, si la salida tiene valor 1, permanece en tal valor hasta que los valores binarios adecuados se apliquen a sus entradas, causando su cambio a un valor de 0. Ya que la salida del flip-flop no cambia hasta que los valores binarios adecuados se apliquen a sus entradas, este dispositivo es capaz de "recordar" el valor de un solo dígito binario (*bit*).

La Figura 3.20 muestra un diagrama de bloques para un flip-flop. Nótese que este dispositivo tiene dos salidas, etiquetadas Q y Q'. Como se ha indicado anteriormente, Q' representa la negación o complemento de Q. Al trabajar con computadoras, es útil y conveniente contar con valores binarios no sólo de las variables, sino también de su complemento, lo que elimina

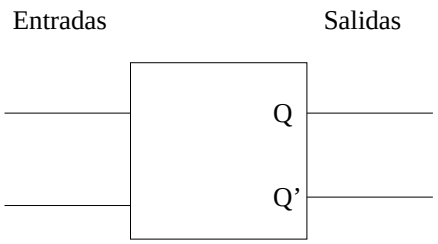

Figura 3.20: Diagrama de bloques de un flip-flop.

la necesidad de muchas compuertas NOT. De tal modo, la mayoría de los flip-flops se construyen para tener disponibles las dos salidas.

Comúnmente, el valor de Q es llamado el *estado* del flip-flop. Si $Q = 1$, entonces el estado del flip-flop es 1. Similarmente, si $Q = 0$, el estado del flip-flop es 0. A continuación, en las siguientes secciones se mencionan algunos de los flip-flops más útiles y conocidos.

3.6.1. El flip-flop RS

La Figura 3.21 muestra el diagrama de bloques de un flip-flop RS. En este flip-flop, si las variables de entrada tienen los valores $R = 0$ y $S = 0$, entonces la variable de salida Q no cambiará en el siguiente tiempo. Por ejemplo, si $Q = 1$ cuando se reciben las entradas $R = 0$ y $S = 0$, entonces Q permanece con valor 1 para el siguiente tiempo (y obviamente, $Q' = 0$ también permanece con tal valor para el siguiente tiempo).

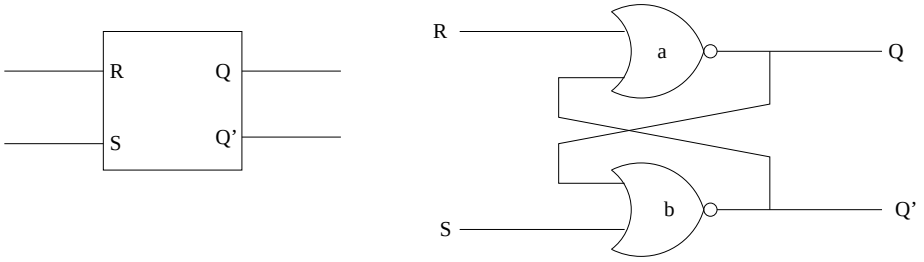

Figura 3.21: El flip-flop RS: diagrama de bloques e implementación con compuertas NOR.

Ahora bien, si $R = 0$ y $S = 1$, entonces la salida Q (el estado) del flip-

flop RS toma un valor de 1 ($Q = 1$, $Q' = 0$). Sin importar cuál era el estado anterior del flipflop RS, después de que se aplican los valores de $R = 0$ y $S = 1$ a las entradas correspondientes, el valor de su salida es $Q = 1$.

Similarmente, si los valores a las entradas del flip-flop RS se hacen $R = 1$ y $S = 0$, entonces el estado del flip-flop se vuelve 0. Sin importar el valor previo de su salida, tras aplicar $R = 1$ y $S = 0$, el valor de su salida es de $Q = 0$.

Finalmente, no es permitido que los valores a las entradas de un flip-flop RS sean $R = 1$ y $S = 1$. En general, si se intenta aplicar tales entradas a un flip-flop RS, se obtienen resultados erráricos. Por ejemplo, supóngase que se tienen dos flip-flops RS construidos por dos fabricantes diferentes. Ambos flip-flops tienen un comportamiento similar para todas las entradas permitidas. Sin embargo, si se aplica $R = 1$ y $S = 1$ a estos flip-flops, pueden actuar en formas diferentes. De hecho, las designaciones de las entradas R y S tienen como origen las palabras *reset* y *set*. Así, las entradas $R = 0$ y $S = 1$ indican al flip-flop tomar el estado $Q = 1$, mientras que las entradas $R = 1$ y $S = 0$ le indican tomar el estado $Q = 0$. La pregunta es ¿qué significaría el hecho de que ambas entradas tomaran el valor de 1 al mismo tiempo?

Por otro lado, hay muchas formas de construir flip-flops. Un flip-flop RS construido con compuertas NOR se muestra en la Figura 3.21. Considérese su operación: supóngase que $Q = 0$ y $Q' = 1$, y que $R = 0$ y $S = 0$. Una entrada a la compuerta NOR **a** tiene valor 1. Por lo tanto, Q permanece con valor 0. Además, ambas entradas a la compuerta NOR **b** son 0; por lo tanto, Q' permanece con valor 1. En resumen, la salida (estado) del flip-flop no cambia.

Ahora, supóngase que las entradas toman valores $R = 0$ y $S = 1$. Después de este cambio, una entrada de la compuerta **b** toma el valor de 1. Por lo tanto, su salida cambia a 0, es decir, Q' cambia a 0. Ahora bien, ambas entradas de la compuerta NOR **a** tienen valor 0, por lo que Q toma el valor 1, cambiando el estado del flip-flop como se había dispuesto.

En el caso en que las entrada tomaran valores $R = 1$ y $S = 0$ con $Q = 1$, una de las entradas a la compuerta NOR **a** toma el valor de 1, por lo que Q toma el valor 0. Después de que este cambio ocurre, las dos entradas a la compuerta NOR **b** tienen valor 0, por lo que Q' se vuelve 1. Esto implica que se han operado los cambios adecuados.

En forma similar, es posible mostrarse que cada uno de los cambios

apropiados de estado suceden para cada conjunto o combinación de valores de las variables de entrada permisibles. Nótese que los cambios operados en las salidas Q y Q' no ocurren instantáneamente, o al mismo tiempo. Por ejemplo, el cambio del estado en que $Q = 0$ ($Q' = 1$) cuando se reciben las entradas $R = 0$ y $S = 1$, implica sólo que al final de la operación $Q = 1$ ($Q' = 0$). Sin embargo, primero cambia Q', tomando el valor de 0, y solo entonces Q cambia su valor a 1. En general, estos cambios ocurren muy rápido, pero siempre requieren un tiempo para realizarse.

En una computadora digital hay un gran número de circuitos diferentes, y éstos no tienden a responder al mismo tiempo o con la misma velocidad. Pero si la computadora debe funcionar apropiadamente, entonces debe mantenerse cuidadosamente un orden secuencial correcto de respuestas de los circuitos. Si se ignora este orden, podría intentarse utilizar la salida de un circuito antes de que éste complete su cambio de estado. Existen varias técnicas utilizadas para mantener la operación ordenada de una computadora, y para asegurar que todos los circuitos responden en los momentos apropiados.

Una técnica muy importante para mantener la sincronización de los circuitos lógicos de una computadora es llamado *temporización por reloj* (o simplemente *clocking*). Un circuito oscilador electrónico, llamado *reloj*, genera un tren de pulsos parecido al que se muestra en la Figura 3.22.

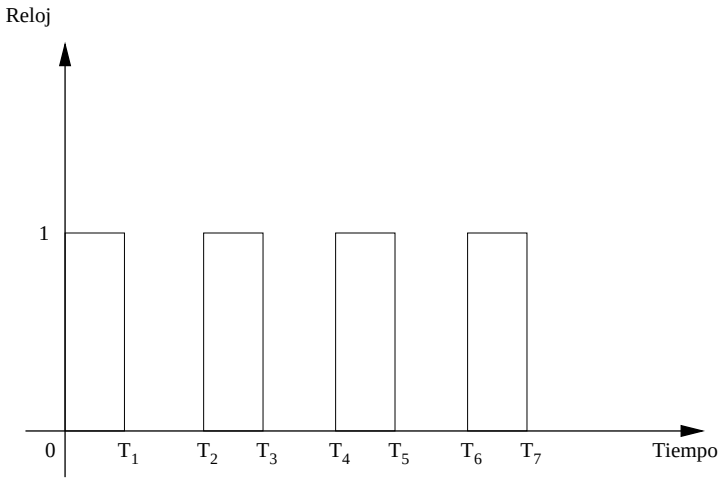

Figura 3.22: Un tren de pulsos de reloj.

Cuando el pulso está presente, el nivel de la señal corresponde a un valor de 1 lógico; cuando el pulso está ausente, corresponde a un valor 0 lógico. Esto es importante, ya que la mayoría de los flip-flops se construyen de tal manera que cambian su estado solo durante el tiempo en que el pulso está presente, como en los tiempo entre 0 y T_1. Por lo tanto, para proveer de un control basado en un reloj, la construcción del flip-flop debe modificarse.

El diagrama de bloques y el circuito que implementa un flip-flop RS con señal de reloj se muestra en la Figura 3.23.

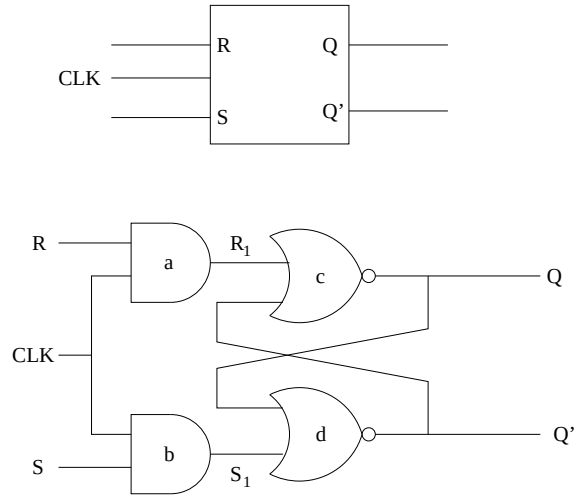

Figura 3.23: Un flip-flop RS con señal de reloj.

Las compuertas NOR **c** y **d** conforman un flip-flop RS ordinario, cuyas entradas son R_1 y S_1. Ahora bien, considérense las compuertas AND **a** y **b**. Cuando el pulso de reloj está ausente, la entrada CLK tiene valor 0. Por lo tanto, las salidas de las compuertas AND **a** y **b** tienen valor de 0, y $R_1 = 0$ y $S_1 = 0$. Así, cuando el pulso del reloj está ausente, el flip-flop no puede cambiar su estado.

Ahora, supóngase que el pulso de reloj se encuentra presente. Su nivel es tal que actúa como un valor de 1, por lo que la entrada CLK a las compuertas AND **a** y **b** tiene valor 1. Considérese la compuerta **a**. Si $CLK = 1$ (es decir, el pulso de reloj está presente) y $R = 1$, entonces $R_1 = 1$. Si $CLK = 1$ y $R = 0$, entonces $R_1 = 0$. Por lo tanto, para los tiempos en que

el pulso de reloj está presente, se tiene que:

$$\begin{aligned} R_1 &= R \\ S_1 &= S \end{aligned}$$

Entonces, si las entradas se aplican durante la presencia de un pulso de reloj, el flip-flop cambia su estado, y se comporta como un flip-flop RS sin reloj. Cuando el pulso de reloj está ausente, el flip-flop no cambia su estado, sin importar los valores de las variables de entrada R y S.

En las siguientes secciones se describen algunos otros tipos particulares de flip-flops, los cuales se consideran todos con señal de reloj.

3.6.2. El flip-flop D

El flip-flop D se diseña con una sola entrada. Su diagrama de bloques se muestra en la Figura 3.24.

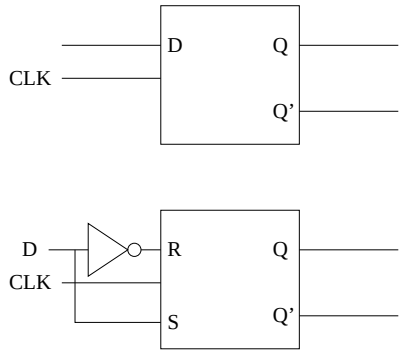

Figura 3.24: Un flip-flop D y su implementación mediante flip-flop RS.

Si durante el pulso de reloj $D = 1$, entonces el estado del flip-flop toma el valor de 1 ($Q = 1$, $Q' = 0$). En forma similar, si durante el pulso de reloj $D = 0$, entonces el estado del flip-flop toma el valor de 0 ($Q = 0$, $Q' = 1$). Pareceía que este tipo de circuitos no son muy útiles, pero como se ve más adelante, hay varios circuitos en que el flip-flop D resulta de mucha utilidad práctica.

Un flip-flop D se construye utilizando una compuerta NOT y un flip-flop RS conectados como se muestra en la Figura 3.24. Nótese que cuando $D = 0$,

entonces $R = 1$ y $S = 0$. Esto causa que el estado del flip-flop RS tome el valor de 0 ($Q = 0$). En forma similar, cuando $D = 1$, entonces $R = 0$ y $S = 1$, lo que causa que el estado del flip-flop RS tome el valor de 1 ($Q = 1$).

3.6.3. El flip-flop JK

En ocasiones, es conveniente contar con un flip-flop que funcione como un flip-flop RS, pero que pueda aceptar como valores de entrada $R = 1$ y $S = 1$. Tal flip-flop se conoce con el nombre de flip-flop JK. En este flip-flop, la variable de entrada J tiene la misma función que S y la variable de entrada K tiene la misma función que R. Cuando se aplican las entradas $J = 1$ y $K = 1$ durante el pulso de reloj, el flip-flop cambia su estado al estado complementario. Esto es, si $Q = 1$ y se aplica $J = 1$ y $K = 1$, entonces el flip-flop cambia al estado $Q = 0$. Similarmente, si $Q = 0$ y se aplica $J = 1$ y $K = 1$, entonces el flip-flop cambia al estado $Q = 1$.

El diagrama de bloques de un flip-flop JK se muestra en la Figura 3.25, así como su implementación utilizando un flip-flop RS y dos compuertas AND.

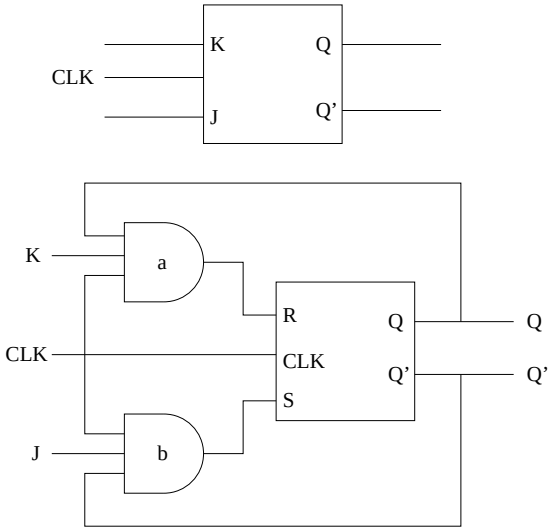

Figura 3.25: Un flip-flop JK y su implementación mediante flip-flop RS y compuertas AND.

El flip-flop JK funciona de la siguiente manera: supóngase que $Q = 0$,

$J = 0$ y $K = 0$. Las salidas de las compuertas AND **a** y **b** toman un valor de 0. Por tanto, $R = 0$ y $S = 0$, y el estado permanece sin cambio ($Q = 0$).

Ahora, supóngase que durante el pulso de reloj se tiene que $Q = 0$, $J = 1$ y $K = 0$. Esto implica que todas las entradas de la compuerta AND **b** tienen valor 1, y por tanto, $S = 1$ para el flip-flop RS. Por otro lado, dos entradas a la compuerta AND **a** tienen valor 0, y por lo tanto, $R = 0$ para el flip-flop RS. Sin embargo, la combinación de entradas $R = 0$ y $S = 1$ causa que el flip-flop RS cambie al estado $Q = 1$ (con $Q' = 0$). Nótese que una vez que el cambio de estado ha ocurrido, al menos una entrada de cada compuerta AND es 0, de tal modo que el estado no continúa cambiando.

En seguida, supóngase que $Q = 1$ ($Q' = 0$) y que los valores de las entradas durante el pulso son $J = 0$ y $K = 1$. Todas las entradas a la compuerta AND **a** tienen valor 1, por lo que $R = 1$. Dos entradas a la compuerta AND **b** tienen valor 0, por lo que $S = 0$. La combinación $R = 1$ y $S = 0$ hace que el flip-flop RS cambie de estado a 0 ($Q = 0$ y $Q' = 1$). Continuando con las combinaciones, se demuestra que el flip-flop JK funciona correctamente para todas las combinaciones de entrada y salida.

Considérese la operación del flip-flop JK cuando sus entradas son $J = 1$ y $K = 1$. Supóngase que el estado es $Q = 1$. Durante el pulso de reloj, todas las entradas para la compuerta AND **a** tienen valor 1, por lo que $R = 1$. Por otro lado, ya que $Q' = 0$, al menos una entrada a la compuerta AND **b** tiene valor 0, y por esto, $S = 0$. Tal combinación de valores de entrada al flip-flop RS provocan que éste cambie de estado a $Q = 0$.

En forma complementaria, supóngase que el estado del flip-flop JK es $Q = 0$, y se tiene la combinación $J = 1$ y $K = 1$ durante el pulso de reloj. Al menos una entrada a la compuerta AND **a** tienen valor 0, por lo que $R = 0$, mientras que todas las entradas a la compuerta AND **b** tienen valor 1, por lo que $S = 1$. Tal combinación de valores de entrada al flip-flop RS provocan que éste cambie de estado a $Q = 1$.

Aun cuando el circuito de la Figura 3.25 funciona propiamente como flip-flop JK para las combinaciones de entrada y salida, un problema puede surgir cuando $J = 1$ y $K = 1$. Supóngase que el flip-flop puede cambiar su estado en un tiempo mucho menor que el pulso de reloj. Cuando se tiene a las entradas $J = 1$ y $K = 1$, el flip-flop JK simplemente cambia su estado. Pero como el pulso de reloj puede todavía estar presente, el flip-flop JK puede cambiar de nuevo su estado. Esto en la mayoría de los casos es indeseable.

Es por esto que más adelante se discuten circuitos que eliminan este tipo de comportamiento errático.

3.6.4. El flip-flop T

Otro flip-flop con una sola entrada es el flip-flop T, que se muestra en la Figura 3.26.

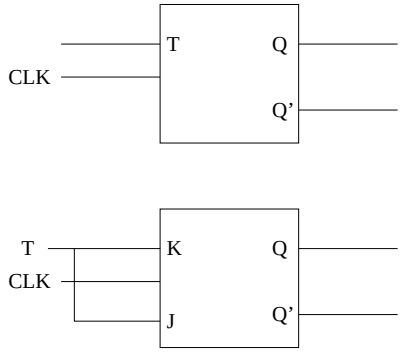

Figura 3.26: Un flip-flop T y su implementación mediante flip-flop JK.

Si $T = 1$ durante el pulso de reloj, el flip-flop cambia de estado. Si $T = 0$, el flip-flop permanece en su estado original. La implementación del flip-flop T se muestra también en la Figura 3.26. Esta implementación consiste de un flip-flop JK cuyas entradas han sido conectadas juntas. Nótese que cuando $T = 0$, $J = 0$ y $K = 0$, y el estado del flip-flop no cambia. Cuando $T = 1$, entonces $J = 1$ y $K = 1$, por lo que el flip-flop cambia a su estado complementario.

3.6.5. Entradas de inicialización

Los flip-flops en forma de circuitos integrados frecuentemente cuentan con otras entradas de inicialización llamadas *preset* y *clear*. Cuando se aplica un valor lógico 1 a la entrada de preset, el flip-flop toma el estado $Q = 1$ ($Q' = 0$). Cuando se aplica un valor lógico de 0 a la entrada clear, el estado del flip-flop se vuelve $Q = 0$ ($Q' = 1$). Generalmente, las entradas preset y clear funcionan independientemente de la entrada para el pulso de reloj, y se utilizan para inicializar el estado de los flip-flops antes de realizar un cómputo dado.

Algunas entradas preset y clear se operan en forma diferente a las que se han descrito, en el sentido de que los niveles de voltaje que corresponden a un valor 1 lógico deben aplicarse contínuamente a las entradas de preset o clear. Si se desea aplicar un preset, la entrada correspondiente debe entonces tomar un valor 0 lógico. En forma similar, si se desea aplicar un clear, la entrada correpondiente debe tomar un valor de 0 lógico.

3.7. Señales de reloj

En la sección anterior se menciona un problema que puede generarse al aplicar las entradas $J = 1$ y $K = 1$ a un flip-flop JK. Si el tiempo que requieren los componentes del flip-flop para cambiar fuese más rápido que el periodo de tiempo de un pulso de reloj, la salida podría cambiar varias veces durante el propio pulso de reloj. Problemas similares puede surgir en otros circuitos de la computadora. Por ejemplo, supóngase que la salida de uno de los flip-flops fuera la entrada de otro, y que una señal se aplica al primer flip-flop de modo que modifique su estado; su salida cambia, por lo que la entrada al segundo flip-flop cambia.

Dependiendo de la velocidad de operación, y cuándo se aplique la señal de entrada al flip-flop, el cambio en la entrada del segundo flip-flop puede ocurrir en varios momentos. Por ejemplo, puede cambiar antes del final del pulso de reloj, o puede ocurrir cuando ya ha pasado el pulso; o posiblemente puede cambiar parcialmente al final del pulso y parcialmente en el siguiente. Si el cambio en la entrada del segundo flip-flop ocurre mucho antes que el final del pulso, entonces el segundo flip-flop cambia su estado inmediatamente. Si el cambio en la entrada del segundo flip-flop ocurre después del pulso de reloj, el segundo flip-flop no cambia de estado durante el pulso, sino muy cerca de su final, por lo que el segundo flip-flop podría no experimentar el cambio requerido de estado. El resultado de todo esto es una operación errática; dependiendo del momento, el flip-flop puede o no cambiar su estado en cierto tiempo.

Tal operación errática no es permisible dentro de una computadora digital, donde la precisión en las operaciones es preponderante. Se pueden evitar, sin embargo, los problemas debidos a las diferentes velocidades de los circuitos si la operación de los flip-flops pudiera modificarse apropiadamente. Supóngase que, como anteriormente se describe, el flip-flop puede cambiar su estado en respuesta a señales de entrada aplicadas *durante* el pulso de reloj. Pero la salida del flip-flop no cambia hasta *después* de que el pulso de

reloj ha pasado. En tal circunstancia, las dificultades se evitan. Considérese el ejemplo previo, donde la salida de un flip-flop se conecta a la entrada de otro flip-flop. Durante el pulso de reloj, la salida del primer flip-flop no cambia; por lo tanto, la entrada al segundo flip-flop no cambia tampoco, y el estado del segundo flip-flop permanece sin cambio. Después de que el pulso de reloj ha ocurrido, por ejemplo, entre los tiempos T_1 y T_2 de la Figura 3.22, la salida del primer flip-flop, y por tanto la entrada al segundo flip-flop, efectivamente cambia. Sin embargo, el estado del segundo flip-flop no puede cambiar durante este tiempo, ya que la señal de reloj tiene como valor 0. Al *siguiente* pulso de reloj, el segundo flip-flop cambia su estado. Así, cualquier ambigüedad en la operación ha sido removida (Nótese que el primer flip-flop no puede cambiar su estado hasta *después* del siguiente pulso de reloj, por lo que no hay ambigüedad en la entrada al siguiente flip-flop).

Hay varios tipos de flip-flops que responden de esta manera. Uno de ellos se conoce con el nombre de *flip-flop de disparo por flanco*. Otro tipo es el *flip-flop maestro-esclavo*. Considérese la operación de un flip-flop RS maestro-esclavo. De hecho, todos los flip-flops pueden ordenarse en esta configuración. La configuración de tal flip-flop, que en realidad consiste de dos flip-flops RS, se muestra en la Figura 3.27

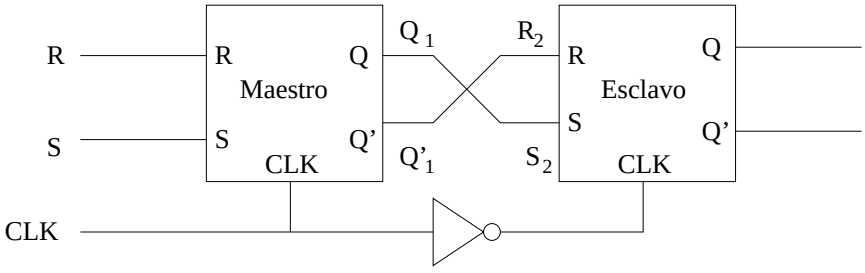

Figura 3.27: Un flip-flop RS maestro-esclavo.

La salida del primer flip-flop (el maestro) es la entrada al segundo flip-flop (el esclavo). Por lo pronto, ignórese la señal de reloj. El flip-flop esclavo siempre toma el valor del estado del flip-flop maestro. Nótese que $Q_1 = S_2$ y que ${Q_1}' = R_2$. Por lo tanto, si el estado del flip-flop maestro es $Q_1 = 1$, entonces se tiene que $R_2 = 0$ y $S_2 = 1$, de tal modo que el estado del flip-flop esclavo se vuelve $Q = 1$. Una situación similar sucede si el estado del maestro es $Q = 0$.

Ahora, considérese el efecto de la conección de los relojes. La terminal de reloj del flip-flop maestro se conecta directamente a la señal de reloj. Sin embargo, la terminal de reloj del flip-flop esclavo es la negación de la señal de reloj. Durante el pulso de reloj, el maestro puede cambiar su estado, pero el esclavo no puede hacerlo, ya que a su entrada de reloj se encuentra un 0 lógico. Por lo tanto, su salida no cambia. Después del pulso de reloj, el estado del maestro no puede cambiar, ya que a su entrada de reloj se presenta un valor de 0 lógico. Sin embargo, ahora el esclavo puede cambiar, ya que tiene un valor 1 lógico en su entrada de reloj. Así, como se desea, la salida sólo cambia cuando el pulso de reloj no se encuentra presente. En general, cuando se labora con circuitos temporizados, se asume que todos ellos disparan por flanco o trabajan en forma maestro-esclavo.

3.8. Registros

En las secciones anteriores se discuten los bloques básicos de construcción de los sistemas de cómputo. Sin embargo, tales bloques básicos resultan ser elementos demasiado simples y pequeños como para describir un sistema de cómputo completo. Es por eso que en esta sección se tratan elementos de mayor tamaño, compuestos por los bloques básicos, pero que permiten describir una computadora digital.

En esta sección se discute un circuito llamado *registro* (*register*), que se utiliza para almacenar un número binario. En realidad, se trata de una simple memoria. En el siguiente capítulo se desarrolla más ampliamente el tema de memorias con mayor capacidad. La información se almacena tanto en registros como en todas las memorias como una secuencia de ceros y unos. En general, esta información se le provee al registro de una de dos maneras: mediante una *carga serial* o *secuencial*, en la que un bit se introduce al registro por cada pulso de reloj (utlizando esta carga, si se desea almacenar un número binario de 8 bits, se requieren 8 pulsos de reloj para ello) y mediante una *carga paralela*, en la que todos los bits se introducen simultáneamente durante un solo pulso de reloj.

Los mismos procedimientos para introducir bits a un registro pueden utilizarse para descargar la información binaria del registro. Por ejemplo, supóngase que la información debe descargarse de un registro con una salida serial. Esto implica que cada bit sale por cada pulso de reloj. En una descarga paralela, todos los bits deben proveerse durante el mismo pulso de reloj.

Aun cuando evidentemente la operación paralela es más rápida que la operación serial, ésta tiene la ventaja de frecuentemente resultar en ahorros en el costo de equipo. Por ejemplo, marcar un teléfono es una operación serial. Se marca primero un número, luego el siguiente, y así hasta haber marcado todo un número telefónico. En una operación paralela, debería haber suficientes teclados para marcar cada dígito del número telefónico y, si se cuenta con suficientes manos, el número podría marcarse en un solo momento. Similarmente, dentro de las computadoras digitales, hay circuitos sumadores seriales y circuitos sumadores paralelos. En éstos últimos, se tiene un sumador completo por cada bit (véase la Sección 3.4), mientras que en un sumador serial, se tiene solo un circuito que realiza la suma de cada bit en turno.

3.8.1. El registro de corrimiento

Un registro de corrimiento es una serie o arreglo de flip-flops cuya operación se ilustra en la Figura 3.28.

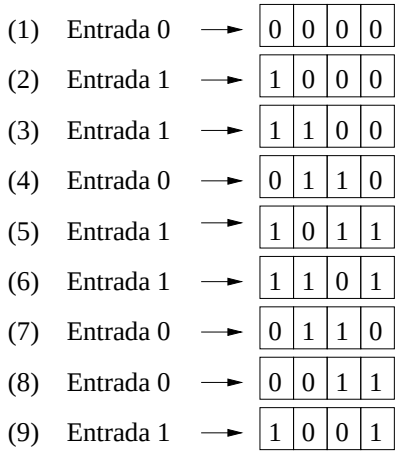

Figura 3.28: La información binaria contenida en un registro de corrimiento durante una serie de pulsos de reloj sucesivos.

Primero se discute su operación y en seguida se considera su implementación. Supóngase inicialmente que el registro de corrimiento es capaz de almacenar 4 bits, y que cada uno de ellos inicialmente tiene un valor de 0. Si se introduce un bit con valor 0 (por la izquierda del registro), después de un pulso de reloj, el valor de todos los bits almacenados es 0 (Figura 3.28, paso

1). Ahora, supóngase que la siguiente entrada es un bit de valor 1. Después de un pulso de reloj, el registro almacena un valor binario 1000 (Figura 3.28, paso 2); es decir, el bit más significativo recibe el valor de 1, y los demás bits reciben a su vez un valor 0 procedente de la posición a su izquierda. Si se introduce en seguida otro bit con valor 1, en el siguiente pulso de reloj el contenido del registro tiene el valor binario 1100, es decir, cada bit cambia su valor, tomando el valor que recibe por su izquierda (Figura 3.28, paso 3). Esto da la impresión de que el valor binario "corre" hacia la derecha. De hecho, el contenido del registro se mueve a la derecha, un bit cada pulso de reloj, mientras que el bit que se introduce por la izquierda ocupa la posición del bit más significativo.

La operación del registro de corrimiento continúa conforme se mueven los bits a la derecha, y se introduce un bit a la izquierda. De hecho, la Figura 3.28 representa un corrimiento serial de 9 bits, que en su orden de entrada, son 011011001.

Con cada pulso de reloj la información binaria se corre un bit a la derecha. Cada vez que esto ocurre, el bit menos significativo se pierde. Se supone que tal información puede ser utilizada por otro dispositivo digital de la computadora.

Partiendo de la descripción anterior, a continuación se describe la construcción de un registro de corrimiento. La Figura 3.29 muestra dos implementaciones sencillas para el registro de corrimiento, una implementada con flip-flops JK y la otra con flip-flops D. Nótese que estas implementaciones solo permiten un corrimiento de la información binaria hacia la derecha.

Considérese la implementación con flip-flops JK, suponiendo que el estado de inicio de todos los flip-flops es 0. Supóngase que la primera entrada es un 0. Entonces $J_1 = 0$ y $K_1 = 1$. Por lo tanto, el estado del flip-flop más a la izquierda permance siendo $Q_1 = 0$ después del primer pulso de reloj. (Nótese que se supone el uso de flip-flops maestro-esclavo o de disparo por flanco.) Ya que $Q_1 = 0$ y $Q'_1 = 1$, entonces $J_2 = 0$ y $K_2 = 1$. Así, el estado del segundo flip-flop permanece en el estado $Q_2 = 0$, y de forma similar, el estado del tercer flip-flop permanece siendo $Q_3 = 0$.

Ahora bien, supóngase que la siguiente entrada al primer flip-flop JK es un 1. En tal caso, se tiene que $J_1 = 1$ y $K_1 = 0$. Entonces, el estado del primer flip-flop debe cambiar a $Q_1 = 1$. Sin embargo, Q_1 no cambia de valor hasta después de que el pulso de reloj ha pasado. Durante el pulso de reloj

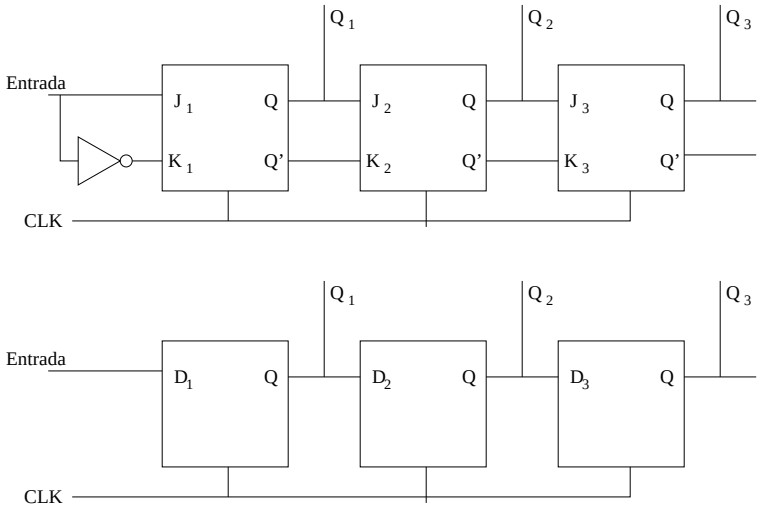

Figura 3.29: Dos registros de corrimiento de tres bits.

$Q_1 = 0$, lo que implica que los otros dos flip-flops no cambian de estado.

Cuando ocurre el siguiente pulso de reloj, entonces el estado del primer flip-flop cambia a $Q_1 = 1$ y $Q'_1 = 0$, lo que forza que el estado del segundo flip-flop cambie a $Q_2 = 1$. Sin embargo, tal salida no ocurre hasta después del pulso de reloj (antes de esto $Q_2 = 0$). Por tanto, el estado del tercer flip-flop no cambia. Por supuesto, el estado del primer flip-flop durante este tiempo se establece a partir de la entrada.

Procediendo de esta manera, la información binaria "viaja" hacia la derecha con cada pulso de reloj, y las entradas sucesivas establecen el estado del flip-flop más a la izquierda.

Por otro lado, si Q_3 se toma como salida, entonces un solo bit sale del registro con cada pulso de reloj. De hecho, este es un ejemplo de salida serial. Pero si las tres terminales Q_1, Q_2 y Q_3 se utilizan simultáneamente como salidas, se tiene una salida en paralelo. Por lo tanto, el registro de corrimiento basado en flip-flops JK de la Figura 3.29 puede utilizarse con entrada serial y para salida tanto serial como paralela (o hasta ambas). Si la entrada es serial y la salida paralela, se tiene un *convertidor de serie a paralelo.*

La operación del registro de corrimiento basado en flip-flops D de la

Figura 3.29 es esencialmente igual, excepto por supuesto, en el tipo de flip-flops utilizados. Recuérdese que el estado de un flip-flop D después del pulso de reloj corresponde al valor binario de su entrada. Si se desea incrementar el número de bits en un registro, solo se necesita añadir flip-flops a la derecha.

En una computadora real, los registros de corrimiento son frecuentemente más versátiles. A continuación, se discute el diseño de un registro de corrimiento que puede ser controlado para realizar varias funciones. Durante su operación normal, se proveen entradas por la izquierda y se van recorriendo hacia la derecha con cada pulso de reloj, un operación ordinaria que ya se ha descrito. Sin embargo, se desea también que sea capaz de aceptar entradas por la derecha, mientras que el registro recorre sus bits a la izquierda. En este caso, funciona como un registro de corrimiento ordinario, excepto que funciona de derecha a izquierda. Además en ocasiones no se desea que el registro recorra sus bits con cada pulso de reloj; es decir, se desea que el registro sea capaz de retener toda la información binaria almacenada en él, sin cambios. También es deseable que se pueda "limpiar" al registro, es decir, poner todos sus estados con valor de 0. Finalmente, se desea que los datos puedan ingresar en paralelo al registro.

Los modos de operación de los flip-flops del registro a diseñar se controlan mediante una serie de señales. Por simplicidad, supóngase que hay cuatro de tales señales: c_0, c_1, c_2 y c_3. Si se desea un corrimiento ordinario a la derecha, los valores de las señales de control serán $c_0 = 1$, $c_1 = c_2 = c_3 = 0$. Si se desea un corrimiento a la izquierda, entonces $c_0 = 0$, $c_1 = 1$, $c_2 = c_3 = 0$. Para entrada paralela, $c_0 = c_1 = 0$, $c_2 = 1$, $c_3 = 0$. Si se desea limpiar el registro, entonces $c_0 = c_1 = c_2 = 0$, $c_3 = 1$. Si no se desea realizar ningún cambio en la información almacenada en el registro, entonces $c_0 = c_1 = c_2 = c_3 = 0$.

Un circuito para este registro controlado se muestra en la Figura 3.30. Básicamente, consiste en el conjunto de flip-flops D mostrado en la Figura 3.29. Se discute a continuación cómo las señales de control producen la operación deseada. Para simplificar la explicación, se redibuja el diagrama para una sola etapa, en la Figura 3.31.

Obsérvese la Figura 3.31. Supóngase que todas las señales c_0, c_1, c_2 y c_3 tienen valor 0. Entonces, todas las entradas a la compuerta OR **e** tienen valor 0, por lo que una de las entradas a la compuerta AND **f** tiene valor 0. Nótese que la otra entrada tiene conectada la señal de reloj, y que la salida de **f** se conecta a las terminales de reloj de cada uno de los flip-flops D. De

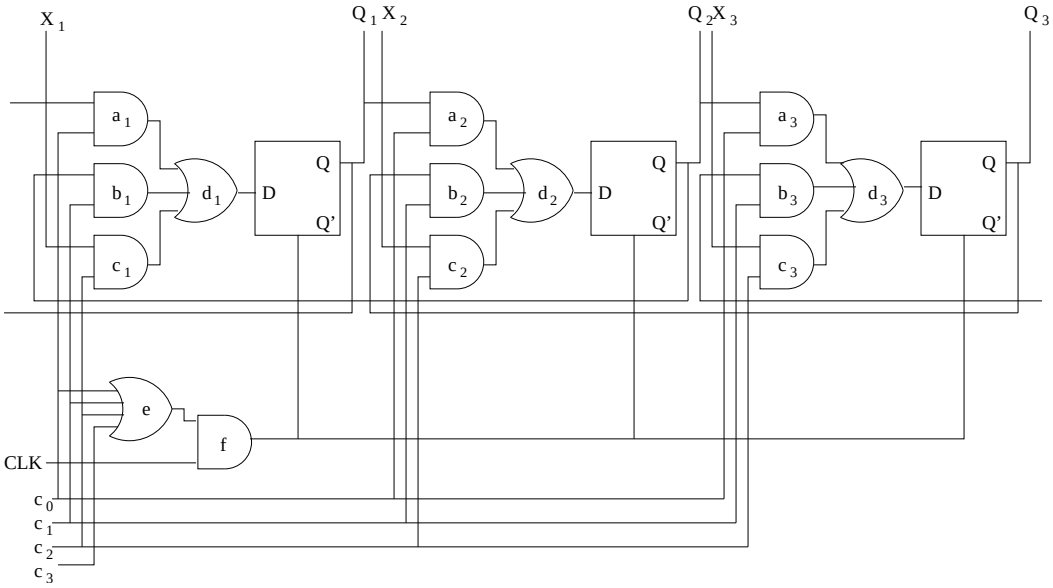

Figura 3.30: Un registro.

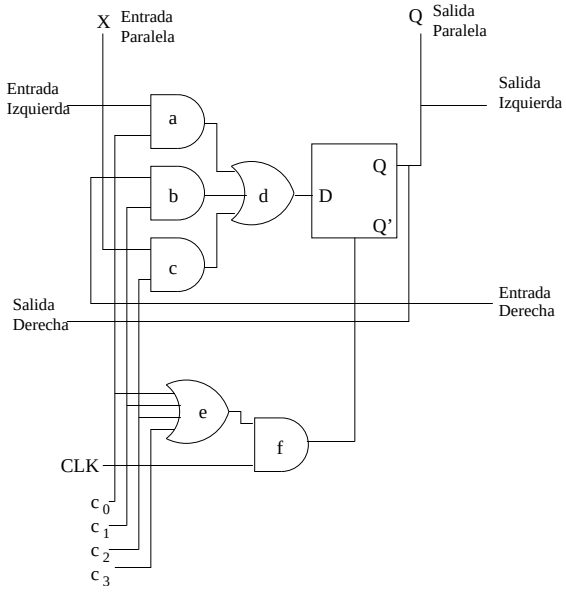

Figura 3.31: Una etapa del registro.

tal modo, para este conjunto de señales de control, ninguna señal de reloj alcanza a los flip-flops, y por tanto, el contenido del registro no cambia con los pulsos de reloj. El contenido del registro se preserva. Si cualquiera de las señales c_0, c_1, c_2 o c_3 tienen un valor de 1, entonces el pulso de reloj se aplica a las terminales correspondientes de los flip-flops, a fin de generar cambios en sus estados.

Ahora, supóngase que $c_0 = 1$ y que $c_1 = c_2 = c_3 = 0$. Entonces, la salida de la compuertas AND **b** y **c** tendrán valor de 0, mientras que la salida de la compuerta AND **a** tomará el valor de la entrada izquierda. Así, esta entrada se convierte en la entrada para el flip-flop D. Viendo el diagrama completo del registro en la Figura 3.30, para la condición $c_0 = 1$ y $c_1 = c_2 = c_3 = 0$, todo el circuito funciona como un registro de corrimiento hacia la derecha basado en flip-flops D que se muestra en la Figura 3.29, ya que cada flip-flop recibe su entrada de la etapa anterior inmediatamente a su izquierda.

Ahora, supóngase que las señales de control sean $c_0 = 0$, $c_1 = 1$ y $c_2 = c_3 = 0$. Entonces, en la Figura 3.31, una entrada de las compuertas AND **a** y **c** tiene valor 0, por lo que las salidas de tales compuertas toma el valor 0. Solamente la compuerta AND **b** tiene un valor de 1 a su entrada, por lo que su salida tomará el valor que haya en la entrada derecha. De tal modo, la entrada al flip-flop D se conecta efectivamente a la entrada derecha. Viendo el diagrama completo de la Figura 3.30, si $c_0 = 0$, $c_1 = 1$ y $c_2 = c_3 = 0$, entonces el circuito actúa como un registro de corrimiento basado en flip-flops D, sólo que ahora el bit de entrada proviene de la etapa anterior a la derecha, y el corrimiento se realiza hacia la izquierda.

Supóngase ahora que las señales de control sean $c_0 = c_1 = 0$ y $c_2 = 1$ y $c_3 = 0$. Usando el mismo razonamiento, nótese en la Figura 3.31 que la entrada al flip-flop se conecta ahora a la entrada paralela **X** mediante la compuerta AND **c**. En la Figura 3.30, puede verse que el circuito no funciona ahora como un registro de corrimiento, ya que cada flip-flop D se conecta efectivamente a su propia entrada paralela **X**, y después de que el pulso de reloj ocurre, almacena tal entrada como su estado. De este modo, se tiene un registro con carga en paralelo.

Finalmente, considérese la secuencia de control $c_0 = c_1 = c_2 = 0$ y $c_3 = 1$. Esto implica que una de las dos entradas a cada compuerta AND tiene valor de cero (Figura 3.31). Por lo tanto, la entrada a cada flip-flop D tiene valor de 0. Ya que $c_3 = 1$, en cualquier momento que el pulso de reloj ocurra, la señal de reloj se aplicará en las entradas de reloj de cada

flip-flop D. Al aplicarse la señal de reloj y teniendo una entrada $D = 0$, el estado siguiente de cada flip-flop será 0. Obsérvese la Figura 3.30. Si $c_3 = 1$ y $c_0 = c_1 = c_2 = 0$, entonces al siguiente pulso de reloj el estado de todos los flip-flops D tendrá el valor de 0. El registro se ha limpiado, como se deseaba en un principio.

Habiendo discutido todos los valores permisibles de las señales de control y mostrado que el registro tiene un funcionamiento como se planteó en un principio, se discuten a continuación las señales de salida del registro. Considérese la salida etiquetada *Salida Derecha* en la Figura 3.31. Se trata de la salida del flip-flop más a la derecha del registro en la Figura 3.30. Si se recorre hacia la derecha el contenido del registro, y se observa los valores de esta salida, el registro funciona como un registro de corrimiento simple.

En forma similar, si se desea que el registro funcione como un registro de corrimiento pero ahora a la izquierda, entonces la salida a observar es la salida del flip-flop más a la izquierda del registro. Nótese que en la Figura 3.31, la salida etiquetada *Salida Izquierda* se conecta a la salida de tal flip-flop en la Figura 3.30.

En la Figura 3.30, las salidas de todos los flip-flops se conectan a terminales etiquetadas Q_1, Q_2 y Q_3. De este modo, una salida paralela del registro puede obtenerse en cualquier momento. Como se ha discutido antes, también hay entradas paralelas X_1, X_2 y X_3, por lo que este registro puede funcionar sencillamente como un registro de carga y descarga en paralelo. Nótese que las entradas pueden cargarse en paralelo, y mediante manipular las señales de control, proveer de una salida serial. Por tanto, este registro puede utilizarse como un conversor paralelo a serial. De igual forma, puede utilizarse como conversor serial a paralelo.

3.9. Contadores

Un circuito digital de gran utilidad se conoce con el nombre de *contador*. Tal circuito se utiliza para contar el número de pulsos que se le aplican. Los contadores se utilizan en las computadoras para poder seguir el número de operaciones que se han realizado. También se les utiliza en muchas aplicaciones no computacionales. Por ejemplo, supóngase que toda persona que entra al metro pasa por un torniquete. Cada vez que alguien entra, el torniquete produce un pulso mediante un interruptor. El contador se utliza entonces para determinar el número de personas que han pasado por el torniquete.

Un contador real produce un número binario equivalente a su cuenta. A continuación, se discute un circuito digital capaz de contar hasta 8 pulsos de reloj. Después de llegar a la cuenta de 7, el siguiente evento a contar produce que este contador reinicie con un valor de 0, y comience de nuevo la cuenta. Es por ello que tal contador se conoce con el nombre de *contador módulo 8*. Esta forma de operación es común en muchos contadores. Por ejemplo, el odómetro de un automóvil es un tipo de contador, que va de 00000 a 99999 kilómetros. Después de que se ha alcanzado tal límite, el odómetro reinicia la cuenta en 00000.

Un circuito para un contador módulo 8 utilizando flip-flops JK se muestra en la Figura 3.32.

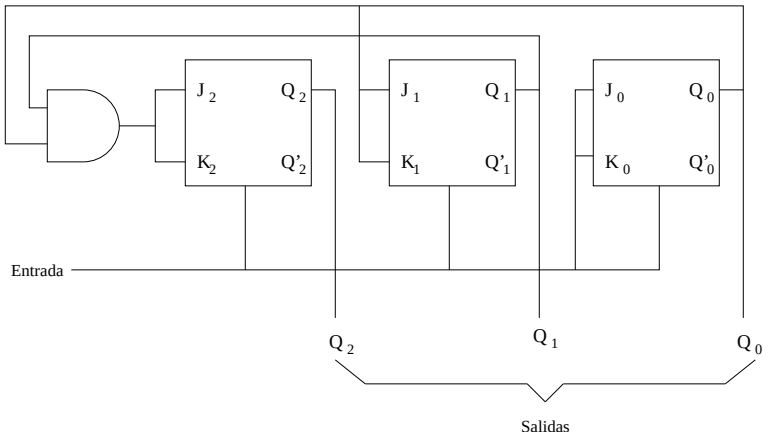

Figura 3.32: Contador módulo 8 usando flip-flops JK.

Supóngase que el estado de todos los flip-flops inicia en 0. Nótese que los pulsos de entrada se aplican en las terminales de reloj de todos los flip-flops, así como a las entradas J_0 y K_0. Cuando se aplica el primer puslo, $J_0 = K_0 = 1$ durante el pulso. Por lo tanto, Q_0 toma el valor de 1 después del pulso. Se supone de nuevo aquí el uso de flip-flops maestro-esclavo o disparados por flanco. Por tanto, Q_0 no pasa de 0 a 1 hasta que el pulso ha pasado. Así, durante el pulso, J_1, K_1, J_2 y K_2 permanecen con valor 0, por lo que Q_1 y Q_2 también permanecen en 0. De este modo, la salida después del primer pulso de entrada es $Q_2 = 0$, $Q_1 = 0$ y $Q_0 = 1$, ó $001_2 = 1_{10}$, como se supone debe ser. Esto indica que se ha contado un pulso.

Cuando un segundo pulso se aplica en la entrada y $Q_0 = 1$, las entradas al segundo flip-flop toman los valores $J_1 = K_1 = 1$. Por tanto, Q_1 cambia de estado hasta que el pulso haya pasado. Además, durante ese tiempo, Q_0 cambia de 1 a 0 ya que $J_0 = K_0 = 1$ durante tal pulso. Nótese que durante el segundo pulso, $J_2 = K_2 = 0$, ya que la entrada de Q_1 a la compuerta AND es 0 hasta después que el pulso haya pasado. De modo que después del segundo pulso, se tiene que $Q_2 = 0$, $Q_1 = 1$ y $Q_0 = 0$, ó $010_2 = 2_{10}$.

Cuando un tercer pulso se aplica a la entrada, Q_0 cambia de nuevo su estado de 0 a 1 después del pulso. Sin embargo, Q_1 no cambia, ya que durante el pulso, $Q_0 = 0$. También, la entrada conectada a Q_0 a la compuerta AND durante el pulso tiene valor 0, por lo que $J_2 = K_2 = 0$, lo que implica que $Q_2 = 0$. Así, después del tercer pulso, se tiene que $Q_2 = 0$, $Q_1 = 1$ y $Q_0 = 1$, ó $011_2 = 3_{10}$.

Ahora, supóngase que llega un cuarto pulso a la entrada. Durante el tiempo que ese pulso se encuentra presente, $Q_0 = 1$ y $Q_1 = 1$, por lo que $J_2 = K_2 = 1$, y entonces $Q_2 = 1$. Ya que $Q_0 = 1$ durante el pulso, entonces $J_1 = K_1 = 1$ durante el pulso. Por lo tanto, el segundo flip-flop cambia su estado después del pulso. Así, después del pulso $Q_1 = 0$. Por tanto, al final se tiene que $Q_2 = 1$, $Q_1 = 0$ y $Q_0 = 0$, ó $100_2 = 4_{10}$.

Continuando así, se puede detallar que el contador llegará al valor de $111_2 = 7_{10}$, para recomenzar con el valor de $000_2 = 0_{10}$ después. Nótese que los contadores de este tipo pueden ensamblarse de modo que pueden contar hasta cualquier potencia de 2. Sin embargo, también es posible diseñar contadores que cuenten hasta el valor que se desee.

3.10. Detectores de secuencia y generadores de secuencia

En esta sección se describen algunos circuitos conocidos como *detectores de secuencias*. Estos circuitos dan como salida 1 si se les introduce una secuencia específica de ceros y unos. De otra manera, su salida es 0.

Los detectores de secuencia no son únicamente utilizados en computadoras digitales. Pueden usarse, por ejemplo, en cerrojos de combinación digital. Supóngase que se cuenta con un control de radio para abrir una puerta. Debe abrir sólamente a una señal de radio, y a ninguna otra. El radiotransmisor, usando un detector de secuencia, transmite la secuencia apropiada.

Considérese un circuito detector de secuencia que da como salida 1 si la secuencia 010 se le introduce (Figura 3.33). Ya que se trata de flip-flops maestro-esclavo, sólo responden a señales aplicadas durante el pulso de reloj, y solo cambian sus estados después del pulso de reloj. Se supone que una señal de entrada se aplica al detector de secuencia durante cada pulso de reloj. Si el nivel de la señal de entrada durante el pulso tiene valor 0, entonces se introduce un 0. De forma similar, si el nivel de la señal de entrada tiene valor 1 durante el pulso de reloj, se introduce un 1.

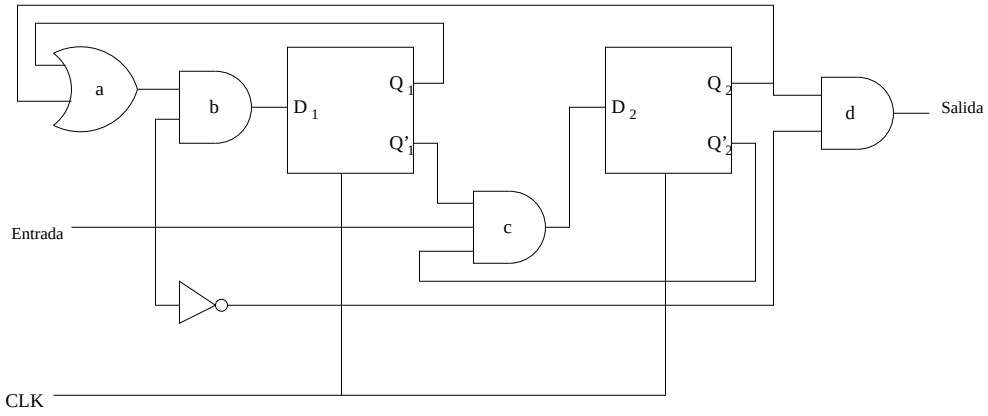

Figura 3.33: Detector de la secuencia 010.

Ahora supóngase que un 0 se ha introducido. Entonces, al menos una entrada de la compuerta AND **b** debe tener valor 0. Por lo tanto, D_1 tiene valor 0, y por lo tanto, después del pulso de reloj, $Q_1 = 0$, $Q_1' = 1$. Además, ya que la entrada fue 0, al menos una entrada a la compuerta AND **c** es también 0, por lo que $D_2 = 0$, y en el siguiente pulso de reloj $Q_2 = 0$. Ya que una entrada a la compuerta AND **d** tiene valor 0, entonces su salida es 0.

Supóngase que en el siguiente pulso de reloj, la entrada tiene valor 1. Ahora, las tres entradas de la compuerta AND **c** tienen valor 1, y por lo tanto, D_2 toma el valor de 1. Para el siguiente pulso de reloj, $Q_2 = 1$ y $Q_2' = 0$. Así, una entrada a la compuerta AND **d** tiene valor 1. Sin embargo, la otra entrada tiene valor 0, ya que es el complemento de la entrada. Por tanto, la salida continúa siendo 0.

Supóngase ahora que durante el siguiente pulso de reloj, se introduce un 0. Q_2 no puede cambiar hasta que el pulso haya pasado. Entonces, durante el pulso de reloj, la entrada inferior de la compuerta AND **d** tiene valor 1, ya que es el complemento de la entrada. Por tanto, la salida toma el valor de 1. Si se procede de esta forma, con todas las posibles secuencias, se puede demostrar que sólo la secuencia 010 provoca una salida 1, y ninguna otra. De tal modo, este circuito funciona efectivamente como un detector de secuencia.

Ocasionalmente, se requiere generar secuencias de pulsos. Esto puede lograrse mediante un dispositivo digital conocido como *generador de secuencias*. Por ejemplo, tal dispositivo puede usarse en un radiotransmisor para abrir una puerta. Los generadores de secuencias consisten de interconexiones de flip-flops y compuertas. El contador de la Figura 3.32 es un generador de secuencias. Si Q_0 se toma como salida, entonces se produce la secuencia 01010... Si Q_1 es la salida, la secuencia es 00110011... Si Q_2 es la salida, la secuencia es 000011110000... Ahora bien, si se utiliza una interconexión mediante compuertas de estas tres señales, se puede obtener la secuencia deseada de cero y unos.

Capítulo 4

Memorias

Todas las computadoras digitales tienen una memoria que se usa para almacenar tanto programas como datos. En este capítulo se discuten diferentes formas de memoria que se utilizan para almacenar diversas cantidades de datos. En la sección 3.8 ya se menciona un tipo muy sencillo de memoria: el registro.

La memoria que almacena un programa en ejecución o los datos para tal programa se conoce como *memoria principal* o en ocasiones *memoria de trabajo*. Además, hay otro tipo de memoria conocida como *secundaria*, *auxiliar* o *periférica*, que permite almacenar programas y datos para su uso futuro. En general, este tipo de memoria consiste de cintas magnéticas y discos. Cuando se desea ejecutar un programa almacenado en memoria secundaria, es necesario "leerlo" de la cinta o el disco, y "cargarlo" en la memoria principal.

Los registros, como un tipo de memoria que se ha discutido anteriormente, almacenan datos binarios. Sin embargo, hay otras formas de memoria creadas a partir de elementos semiconductores, como por ejemplo, la *memoria de sólo lectura* (*read-only memory* o ROM), que almacena permanentemente sus datos y no pueden ser (fácilmente) cambiados.

Actualmente, las memorias semiconductoras se fabrican mediante tecnología de circuitos integrados, y son generalmente rápidas y pequeñas. Otros tipos de memorias utilizan materiales magnéticos y ópticos, y son más lentas que las memorias semiconductoras. Sin embargo, las memorias magnéticas y ópticas tienen ventajas sobre las memorias semiconductoras: *(a)* tienen una mayor capacidad de almacenamiento de información binaria, y *(b)* en

el evento de una falla de corriente, no pierden la información que almacenan. A esta capacidad se le conoce como *no volatilidad*, en contraste con las memorias semiconductoras, que al fallar la fuente de alimentación eléctrica sí pierden sus contenidos. Son memorias *volátiles*.

Las memorias se clasifican por la forma en que se les escribe o lee información binaria. De hecho, los contenidos binarios almacenados en una memoria se organizan en grupos de bits llamados *palabras*. Cada palabra se almacena en una localidad de memoria llamada *dirección*. Cuando cada palabra se almacenan una tras otra, se dice que el almacenamiento es de tipo *secuencial*. En tal caso, para poder acceder (leer o escribir) la *n*-ésima palabra, es necesario pasar por todas las *n-1* palabras anteriores. En general, las memorias con acceso secuencial tienden a ser muy lentas.

En contraste, la forma más rápida de acceso (lectura o escritura) de memoria es el *acceso aleatorio* (*random access*), que es característico de la memoria RAM (*random access memory*). En este tipo de memoria, cualquier localidad de memoria puede direccionarse sin considerar ninguna secuencia u orden específico. Ya que la lectura y escritura se hace velozmente, la RAM se utiliza siempre cuando la velocidad de acceso a los datos es importante.

Este capítulo describe brevemente todos estos tipos de memoria. La discusión se inicia con las memorias semiconductoras, particularmente con las memorias de acceso aleatorio (RAM).

4.1. Memorias de acceso aleatorio – RAM

Las memorias de acceso aleatorio (RAM), construidas mediante circuitos semiconductores, son utilizadas como la memoria principal de la mayoría de las computadoras digitales. Las memorias se organizan en palabras. Cada palabra consta de un número fijo de bits, que pueden tomar los valores binarios de 0 y 1. Por ejemplo, una palabra de memoria puede ser de 8 bits de longitud. Una palabra puede representar un número dentro de un cálculo, o puede representar una instrucción en un programa. De hecho, una operación típica que involucra a la memoria es la suma de dos valores numéricos contenidos en dos palabras con diferentes direcciones en memoria. El resultado puede a la vez almacenarse en una tercera dirección.

La localidad de una palabra en una memoria se conoce con el nombre de dirección. Cuando se almacenan datos en cada bit de una palabra, se dice que los datos almacenados se *escriben* en memoria. Similarmente, cuando

los datos almacenados en cada bit de una palabra se extraen de la memoria, se dice que se *leen* de la memoria. La palabra que ha sido escrita o leida de la memoria, para tal efecto, se dice que ha sido *direccionada*, es decir, se encuentra su localidad en la memoria ya sea para modificarse (escribirse) o inspeccionarse (leerse).

Aun cuando hay variaciones al respecto, resulta conveniente suponer que cada bit de una palabra se almacena en un flip-flop, es decir, que hay un flip-flop por separado cuyo estado determina el almacenamiento de cada bit en la memoria. En computadoras pequeñas, las palabras comúnmente contienen 8, 16 ó 32 bits. En grandes computadoras, las palabras contienen 16, 32, 64 o más bits. En general, se intenta que la memoria sea capaz de almacenar un número grande de palabras.

De hecho, el almacenamiento de un solo bit involucra más que un simple flip-flop. Debe haber previsiones indicando si la información binaria debe ser escrita o leida. En general, existe una terminal del circuito de memoria etiquetado *read/write*. Cuando se coloca un valor de 1 en tal terminal, la memoria almacena el dato binario (se escribe en memoria); cuando se coloca un valor de 0 en la terminal, la memoria provee el dato binario (se lee la memoria). También debe tomarse en consideración el direccionamiento de cada palabra; es decir, que al leer o escribir una palabra en memoria, debe proveérsele una dirección de memoria.

Una *celda binaria* es la unidad básica de memoria, y se utiliza para almacenar un solo bit. Además de contener el valor binario, también considera señales binarias para su lectura y escritura, así como para su direccionamiento. La Figura 4.1 muestra el diagrama de bloque de una celda binaria. Nótese que cuenta con cinco terminales: una entrada (para la escritura del bit), una salida (para la lectura del bit), una terminal *read/write*, una terminal de direccionamiento y una entrada de reloj.

La Figura 4.1 también muestra el circuito digital de una celda de memoria usando un flip-flop RS. Considérese su operación. Para que el circuito funcione, debe tenerse un valor 1 en la terminal A (de direccionamiento). De no ser así, entonces al menos una entrada de las compuertas AND **c**, **d** y **e** tiene un valor de 0, y por lo tanto, $R = 0$ y $S = 0$, por lo que el flip-flop no puede cambiar su estado: no puede escribirse en la celda binaria. Además, la salida tiene valor 0. Por lo tanto, el contenido de la celda de memoria tampoco puede leerse.

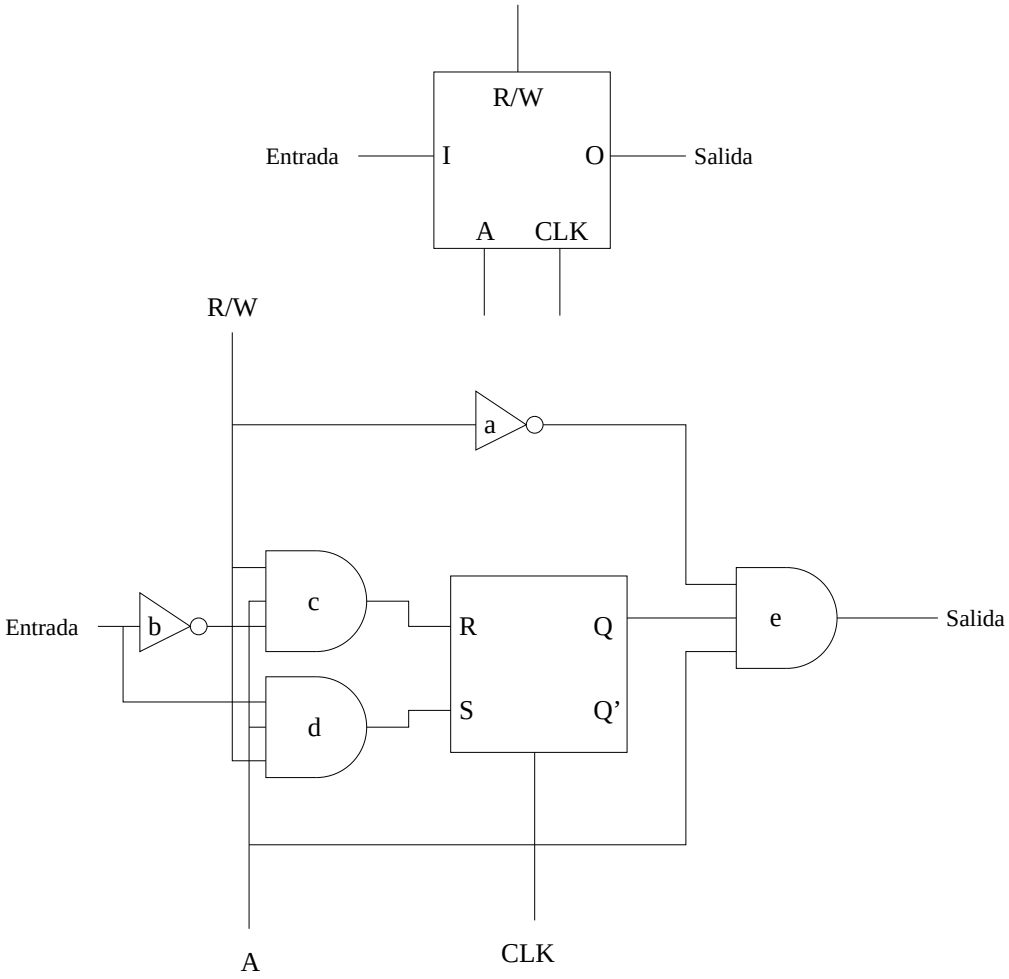

Figura 4.1: Celda binaria en diagrama de bloque e implementación usando flip-flop RS.

Ahora bien, supóngase que la celda ha sido direccionada, es decir, hay un valor de 1 en su terminal A. También supóngase que se desea escribir un bit en ella, de modo que se coloca un valor de 1 en su terminal R/W. De este modo, dos de las entradas a las compuertas AND **c** y **d** tienen valor 1. Supóngase que se desea almacenar (escribir) un 1. Entonces, se coloca tal valor en la terminal de entrada. De este modo, las tres entradas de la compuerta AND **d** tienen valor 1, lo que hace que $S = 1$. Sin embargo, al menos una entrada a la compuerta AND **c** tiene valor de 0, por lo que $R = 0$. Dado que $R = 0$ y $S = 1$, el flip-flop cambia cuando ocurra el pulso de reloj, poniéndose en estado $Q = 1$. De este modo, se ha almacenado un 1 en la celda.

De manera similar, si se tienen valores de 1 en las terminales A y R/W, y la entrada tiene valor 0, entonces $R = 1$ y $S = 0$, por lo que se almacena un 0 en la celda de memoria.

Supóngase ahora que se desea leer el bit almacenado en la celda de memoria. Hay un valor de 1 en la terminal A, pero un valor 0 en la terminal R/W, por lo que al menos una entrada de las compuertas AND **c** y **d** tiene valor 0, y entonces $R = 0$ y $S = 0$, por lo que la información almacenada no cambia. Sin embargo, dos entradas a la compuerta AND **e** tienen valor de 1, por lo que la salida tendrá el valor Q, que corresponde al bit almacenado en la celda binaria.

Una memoria real consiste de muchas celdas binarias organizadas en palabras. La Figura 4.2 muestra una memoria compuesta de cuatro palabras de cuatro bits cada una (en general, el número de palabras y el número de bits por palabra no necesariamente son iguales). Nótese la notación usada. Se han añadido subíndices a la terminal A, de dirección, de cada celda binaria. Estos subíndices indican el número de palabra y la posición del bit en la palabra que la celda ocupa en la memoria. Por ejemplo, los subíndices 1,0 indican la palabra número 1, bit número 0. Del mismo modo, 2,1 indica palabra 2, bit 1.

Cada palabra se direcciona utilizando una terminal de dirección. Nótese que todas las terminales A de las celdas de memoria pertenecientes a la misma palabra están conectadas juntas. Así, si un valor 1 se coloca en la terminal de dirección 0 y se colocan valores de 0 en las demás terminales, entonces se direccionan todas las celdas de la palabra 0. Ninguna otra de las celdas de memoria se direccionan. De manera similar, si se coloca un valor 1 en la terminal de dirección 2 con valores 0 colocados en las demás terminales

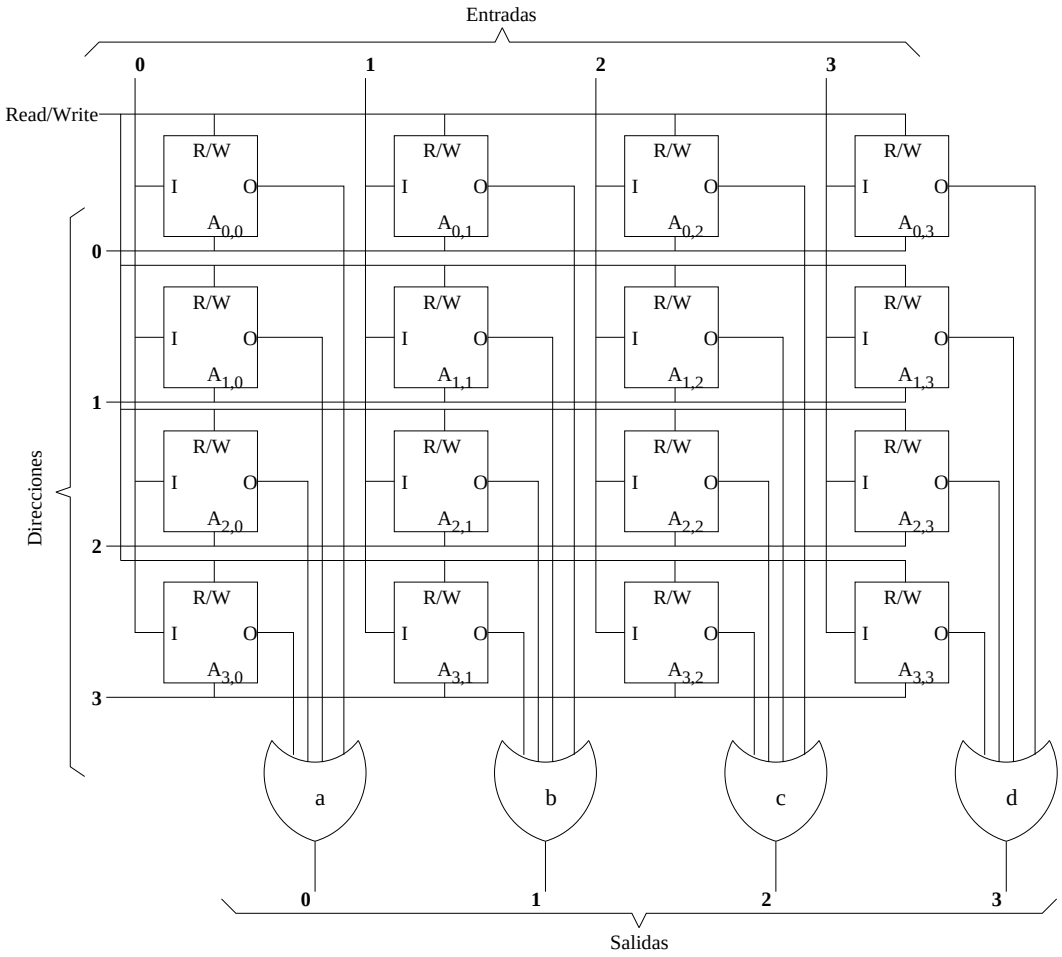

Figura 4.2: Una memoria de cuatro palabras de cuatro bits. La señal de reloj se omite por claridad.

de dirección, entonces todas las celdas de la palabra 2 se direccionan y ninguna otra celda en la memoria. Recuérdese que cuando una celda de memoria no se direcciona, no pierde su información almacenada.

Nótese que el número de terminales de entrada es igual al número de terminales de salida, y al número de bits en una palabra. Ya que se utilizan palabras de cuatro bits, hay cuatro terminales de entrada. Las terminales *read/write* de todas las celdas estan conectadas. Si se desea almacenar una palabra (de cuatro bits) en una localidad de memoria cuya dirección es 2, entonces debe colocarse un 1 en la terminal *read/write* y en la terminal de dirección número 2 (se debe colocar un valor de 0 a todas las demás terminales de dirección). Después del siguiente pulso de reloj, el estado de todos los flip-flops en la palabra 2 será igual a los valores colocados en sus respectivas terminales de entrada. Por lo tanto, la palabra de cuatro bits de entrada ha sido almacenada en la dirección 2. Nótese que todas las terminales de reloj deben conectarse a un reloj maestro. Tal conexión se omite en el diagrama de la Figura 4.2, para evitar sobrecargar el diagrama.

En seguida se describe cómo se lee una palabra de una localidad de memoria. Hay una terminal de salida por cada bit de todas las palabras. Así, para esta memoria, hay cuatro terminales de salida. La salida de una celda de memoria será 0 a menos que haya tanto un valor de 1 en su terminal de dirección, como un 0 en la terminal *read/write*. Supóngase que se coloca un valor de 1 en la terminal de dirección 3 y 0 en todas las demás terminales de dirección. Además, un 0 se coloca en la terminal *read/write*. Obsérvese la compuera OR **a**. Tiene cuatro entradas, cada una de las cuales corresponde al bit 0 de cada palabra en la memoria. Sin embargo, las salidas del bit 0 de las celdas de memoria 0, 1 y 2 tienen un valor de 0, ya que estas palabras no están direccionadas. Por lo tanto, la salida de la compuerta OR **a** es igual al valor almacenado en el bit 0 de la palabra 3. Así, cuando se coloca un valor 0 en la terminal *read/write*, las terminales de salida tendrán los valores de los bits almacenados en la palabra que haya sido direccionada.

Este tipo de lectura se conoce con el nombre de *no-destructiva* ya que no remueve o borra la información almacenada en la memoria. La palabra permanece almacenada hasta que se sobreescribe. Una palabra almacenada permanece sin cambios a menos que se coloque un valor de 1 en la terminal *read/write*, y que tal palabra sea direccionada. En tal caso, los datos binarios colocados en las terminales de entrada reemplazan a la palabra almacenada.

En la memoria de la Figura 4.2 se tiene una terminal de dirección separada por cada palabra en la memoria. Si esta memoria fuera fabricada en un circuito integrado, debería tener una terminal por cada dirección. Ya que las memorias normalmente contienen una cantidad relativamente grande de palabras, se requeriría de igual número de terminales para su direccionamiento. Esto es, en la realidad, indeseable. El circuito integrado tendría que fabricarse demasiado grande para albergar todas las terminales necesarias, introduciendo a la vez otros problemas relacionados, como aquéllos debidos a la conectividad y al costo de fabricación. Es por ello que se hace necesario hallar una forma en que se pueda realizar el direccionamiento de una gran cantidad de localidades de memoria, pero sin incluir un gran número de terminales.

Para reducir el número de terminales *externas*, la dirección puede darse como un número binario. Por ejemplo, en el caso de haber tres terminales de dirección, éstas pueden usarse para indicar hasta ocho direcciones: 000, 001, 010, 011, 100, 101, 110 y 111. En forma general, si hay N terminales de dirección, entonces se pueden direccionar hasta 2^N localidades de memoria. Para la memoria de la Figura 4.2, se requieren entonces dos terminales de dirección. Aunque esto no parece mucho, para cantidades grandes de palabras en memoria, el ahorro es más dramático. Por ejemplo, con 16 terminales para direccionamiento, se puede direccionar hasta 65,536 palabras (o simplemente, 64k palabras).

Dentro del circuito integrado, cada palabra debe tener una terminal de dirección separada como se indica en la Figura 4.2, de tal modo que es necesario contar con una lógica integrada en el circuito tal que convierta las direcciones de entrada en binario a señales individuales para las terminales de dirección. En la Figura 4.3 se muestra un circuito que controla cuatro terminales internas a partir de dos entradas de dirección binarias.

La dirección binaria se introduce por la terminales a_0 y a_1. La forma del número binario es a_1a_0. Si $a_1 = 1$ y $a_0 = 0$, entonces la dirección binaria es 10_2. Nótese que estas entradas controlan el estado de los flip-flops mediante conectarse a las entradas D_0 y D_1. Después de un pulso de reloj, se tiene que $Q_0 = a_0$ y $Q_1 = a_1$. Tales valores se pasan a las compuertas AND. Por ejemplo, la compuerta AND **a** sólo puede tener una salida con valor 1 si $Q_0 = Q_1 = 0$, es decir, si $a_0 = a_1 = 0$ durante el pulso anterior. En el caso de este ejemplo, se tiene que la compuerta AND **c** es la única que tiene una salida 1 para $Q_1 = 1$ y $Q_0 = 0$, lo que coloca un valor de 1 en la terminal interna de dirección 2 ($10_2 = 2_{10}$). Todas las demás terminales de dirección

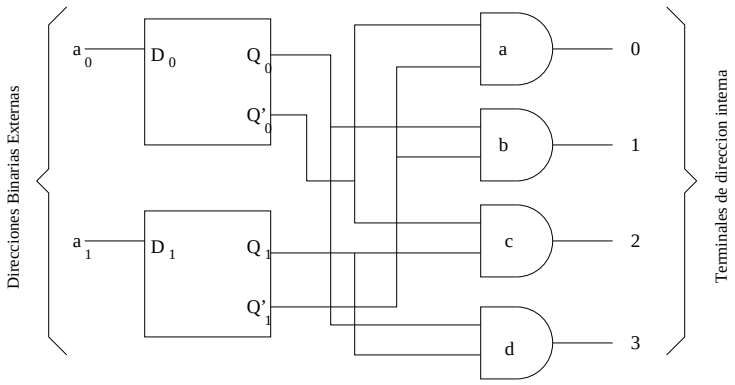

Figura 4.3: Un decodificador de dirección. Las señales de reloj se omiten.

interna tienen un valor 0. Por tanto, el circuito de la Figura 4.3 se conoce como *decodificador de dirección*.

De hecho, el decodificador de la Figura 4.3 no requiere necesariamente contener flip-flops. Esto se utilizan con el sólo propósito de almacenar la dirección, dado que en ocasiones, la dirección binaria puede estar presente por un periodo corto de tiempo, aun cuando se requiera contar con la dirección por un tiempo mayor. En tal caso, los flip-flops resultan útiles.

4.2. Memorias de solo lectura – ROM

La memoria RAM descrita en la sección anterior se diseña para poder escribir y leer información binaria de ella. Sin embargo, en muchas ocasiones no es necesario cambiar la información binaria almacenada una vez que ésta se introduce a la memoria. Es decir, la información binaria se escribe una vez, y no se cambia ni reescribe. Para este tipo de comportamiento de la memoria, resulta complicado el uso de la RAM. Es por ello que para aplicaciones en las que el contenido de la memoria no cambia durante su funcionamiento, se prefiere el uso de memoria ROM (*read-only memory*). La memoria ROM puede utilizarse para almacenar información binaria que se utiliza repetitivamente, como tablas de datos o instrucciones comúnmente utilizadas. La memoria ROM es no-volátil: la información que se almacena en ella no desaparece cuando se le deja de suministrar energía eléctrica.

Hay varios tipos de memorias ROM. En la ROM ordinaria, la información binaria se almacena durante la fabricación del circuito. El usuario provee al fabricante de la información binaria que requiere, y la ROM se fabrica bajo tal especificación. Una vez escrita, la información binaria no puede perderse ni modificarse. Está "grabada en piedra".

La ROM programable (*programmable ROM* ó PROM) es otra forma de ROM en la cual, después de su fabricación, el usuario puede escribir información binaria una vez, y a partir de ello, la información binaria queda permanentemente almacenada, y resulta muy difícil cambiarla de nuevo.

La ROM programable y borrable (*erasable PROM* ó EPROM) es un tercer tipo de ROM, que tiene la posibilidad de ser modificada después de su fabricación, y borrarse al ser afectada por una corriente eléctrica (*electricably erasable PROM* ó EEPROM) o por su exposición a rayos ultravioleta (*ultraviolet erasable PROM* ó UVEPROM). Sin embargo, los procesos de escritura y borrado son extremadamente lentos comparados con la velocidad de lectura y escritura en memoria RAM. Por lo tanto, todos los tipos de ROM se utilizan casi siempre como almacenamiento permanente de información binaria, excepto en los raros casos en que se requiere su reprogramación.

4.2.1. Estructura de la ROM básica

Para describir la operación de una memoria ROM, se utiliza la estructura de conexiones que se muestra en la Figura 4.4. Las terminales de salida se conocen como *bits de salida*. Por el momento, considérese que las terminales de palabra son entradas, y que sólo se permite colocar una valor de 1 en solo una de las terminales de entrada a la vez. El valor lógico de las demás terminales se mantiene en 0. En respuesta de un valor 1 en una terminal de palabra, una palabra se produce en las terminales de salida.

Para describir su operación, supóngase que la ROM en la Figura 4.4 debe realizar la siguiente función: como tiene 10 terminales de palabra, cada una corresponde a un dígito decimal. Cuando un valor de 1 se coloca en una terminal de palabra, su valor binario aparece en las terminales de salida. Por ejemplo, si se coloca un 1 en la terminal de palabra 1, entonces la salida es $a_3 = 0$, $a_2 = 0$ $a_1 = 0$ $a_0 = 1$ ($0001_2 = 1_{10}$). De forma similar, si se coloca un 1 en la terminal de palabra 6 se tiene que $a_3 = 0$, $a_2 = 1$ $a_1 = 1$ $a_0 = 0$ ($0110_2 = 6_{10}$).

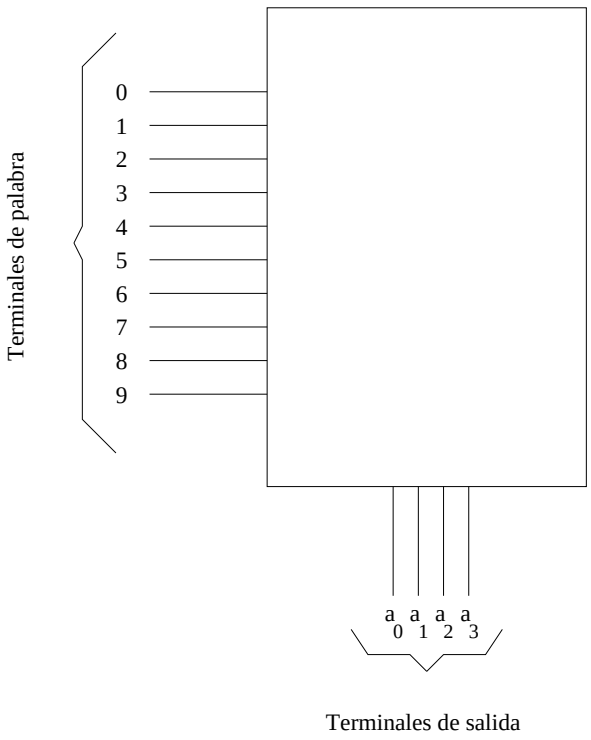

Figura 4.4: Conexiones de una ROM.

Los circuitos en el interior de la ROM conectan las terminales de salida con las terminales de palabra apropiadas. Estas conexiones no se hacen de forma ordinaria con alambre, sino más bien se utilizan circuitos semiconductores que previenen de la conexión directa entre circuitos de entrada y salida.

Una entrada real de una ROM consiste de un decodificador de dirección como el descrito en la Figura 4.3. Ahora, cada terminal de dirección del decodificador se conecta a la terminal de dirección correspondiente de la ROM. De tal forma, la ROM se utiliza de la misma manera que una RAM ordinaria, es decir, se le provee de una dirección a la ROM y ésta produce una palabra como salida (Nótese que puede considerarse que la ROM es una forma de RAM).

4.2.2. ROM programable – PROM

Cuando una PROM sale de la fábrica, viene con todas las conexiones "cerradas" entre sus entradas y sus salidas. Si se coloca un valor de 1 en cualquier terminal de entrada, se genera un 1 en todas las terminales de salida, es decir, la PROM no contiene información, y todos sus bits se encuentran con valor 1. Para poder programarla, la PROM cuenta con conexiones fundibles. Una conexión fundible actúa como un fusible común. Cuando se pasa suficiente corriente eléctrica por él, se derrite, abriendo el circuito. El usuario de la PROM, entonces, debe fundir aquellas conexiones donde la conexión entre entrada y salida no se desea. Esto se logra mediante pasar una corriente eléctrica relativamente grande entre las terminales de entrada y salida donde no se desea la conexión, lo que representa incorporar 0's de modo que la PROM se va llenando con la información binaria requerida por el usuario.

Como ejemplo, considérese que la ROM de la Figura 4.4 fuera una PROM. Durante su programación, todas las conexiones fundibles entre la terminal de dirección 0 y las salidas tendrían que romperse, de tal modo que no hubiera salida en las terminales fundidas cuando a la entrada se coloca un 1 en la terminal de dirección 0. De forma similar, la conexión entre la terminal de dirección 3 y las salidas a_3 y a_2 deben romperse, ya que al aplicar un valor de 1 a la terminal de dirección 3 se espera que a la salida $a_3 = a_2 = 0$ y $a_1 = a_0 = 1$ ($0011_2 = 3_{10}$). Procediendo de esta forma, toda la PROM puede programarse. Una vez programada, no puede volverse a programar. Se pueden romper algunas conexiones adicionales, pero aquéllas que han sido rotas no puede restaurarse.

4.2.3. PROMs borrables – EEPROM y UVEPROM

La conexión entre las salidas y entradas en una PROM borrable es un circuito semiconductor que inicialmente no provee de tal conexión, sino que se modifica su estado por efecto de una corriente eléctrica intensa (o en ocasiones un voltaje alto) aplicado entre las terminales de entrada y salida. Estas terminales queda "conectadas" en una forma (casi) permanente.

Las PROM borrables se programan mediante atrapar una carga en áreas de aislamiento eléctrico. Cuando se aplica la corriente intensa, la región de aislamiento se rompe temporalmente. En tal circunstancia, una carga se atrapa en la región. Cuando se quita la corriente, la capa aislante se reestablece, pero ahora cuenta con la carga eléctrica atrapada. La presencia de tal carga se utiliza para encender los dispositivos electrónicos que conectan las terminales de entrada y salida de la PROM.

Una PROM de este tipo puede borrarse mediante su exposición a luz ultravioleta (UVEPROM). Esto causa que la capa aislante se rompa temporalmente, liberando la carga atrapada. Los niveles de luz ultravioleta necesarios para borrar una memoria son tan altos que la exposición a una luz ordinaria no es capaz de borrarla. No se requiere introducir corrientes intensas cuando se borra este tipo de memorias. Así, la PROM puede volverse a programar con el procedimiento normal. Nótese que borrar y programar son dos operaciones muy lentas, comparadas con la escritura y lectura de una RAM. Es por esto que las PROM no son frecuentemente reprogramadas.

Existen algunos circuitos especiales controlados por microprocesador que se utilizan para programar PROMs. En los más sofisticados, la información binaria puede almacenarse primero en una RAM, y luego escribirse en la ROM para programarla.

4.3. Paralelización de dispositivos de memoria

El número de palabras, o el número de bits por palabra, que se almacenan en una memoria semiconductora dada es muy comúnmente insuficiente para almacenar todos los datos e instrucciones binarias de un programa dado. Sin embargo, es posible incrementar la capacidad de la memoria semiconductora mediante componer una memoria semiconductora "más grande" a partir de acumular memorias pequeñas. De hecho, se tienen disponibles en el mercado algunos dispositivos de memoria que permiten incrementar la cantidad de memoria de una computadora. Al incremento de memoria para utilizarse

como una sola memoria semiconductora en un sistema de cómputo se le llama *paralelización de dispositivos de memoria*. Esta técnica es muy utilizada por quienes cuentan con computadoras pequeñas para poder acceder a memorias de trabajo mayores. En tales casos, los dispositivos de memoria consisten de tarjetas impresas que cuentan con varias memorias y otros circuitos para su conexión. Alternativamente, el tamaño de la memoria puede incrementarse mediante añadirle memorias en forma de circuito integrado. A esto se le conoce como la *paralelización de circuitos integrados de memoria*.

4.3.1. La terminal *Chip Select* (CS)

Muchos dispositivos de memoria diseñados para paralelizarse frecuentemente tienen una terminal externa de direccionamiento llamada *chip select* (selección de circuito o CS). Esta terminal se utiliza para apagar el decodificador de dirección interno del dispositivo de memoria. La terminal CS funciona de la siguiente manera: si se le coloca un valor de 1, entonces el decodificador de dirección funciona como se muestra en la Figura 4.3, es decir, utiliza direcciones binarias externas para colocar un valor de 1 en la terminales de dirección internas. Si se coloca un valor de 0 en la terminal CS, entonces el decodificador de dirección no funciona. No importa qué valores se encuentren en las terminales externas de dirección, el decodificador no tendrá como salida ningún valor de 1.

El decodificador de dirección de la Figura 4.3 puede modificarse como se muestra en la Figura 4.5 para incluir una terminal CS. En tal caso, se añade una terminal extra de entrada a cada compuerta AND **a**, **b**, **c** y **d**, las cuales se conectan a la vez a la entrada CS. Cuando el valor de entrada en CS es 1, entonces el circuito de la Figura 4.5 funciona esencialmente como el circuito de la Figura 4.3. Por otro lado, cuando el valor en la terminal CS es 0, las salidas de todas las compuertas AND toman el valor de 0, por lo que ninguna palabra de memoria se direcciona.

4.3.2. La terminal *Output Enable* (OE)

Los dispositivos de memoria diseñados para la paralelización frecuentemente tienen una terminal de control de direcciones llamada *output enable* (habilitación de salida u OE). Cuando se coloca un valor de 1 en esta terminal, la memoria funciona de manera normal. Por otro lado, si se coloca un valor de 0, entonces la salida se desconecta efectivamente de las terminales de salida del dispositivo de memoria. Esto se conoce con el nombre de circuito tri-estado. En la Figura 4.2 se muestra cómo la salida de varias

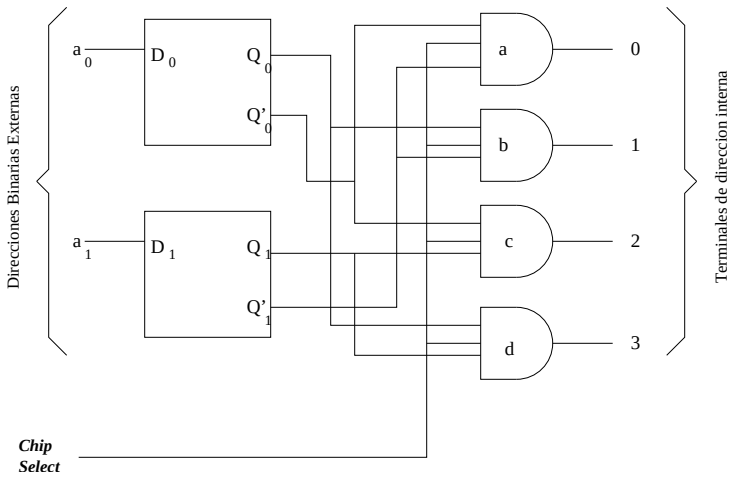

Figura 4.5: Modificación de un decodificador de dirección para contar con una terminal *chip select*.

celdas binarias se conectan mediante compuertas OR a las terminales de salida de la memoria. El uso de OE elimina la necesidad de tales compuertas OR cuando las salidas de la memoria se paralelizan. Algunos dispositivos de memoria combinan las dos terminales, CS y OE, en una sola terminal *chip enable* (habilitación de circuito o CE).

En seguida, se describe cómo dispositivos de memoria pueden paralelizarse para incrementar el número de palabras en la memoria de una computadora. En la Figura 4.6 se muestra la paralelización de tres dispositivos de memoria. En realidad, se supone que cada dispositivo de memoria cuenta con una estructura similar a la memoria de la Figura 4.2, conjuntamente con un decodificador de dirección con una terminal CS (Figura 4.5). Se supone que cada dispositivo de memoria tiene una terminal OE que se ha combinado con CS para contar con una sola terminal CE.

Cada dispositivo de memoria contiene cuatro palabras de cuatro bits cada una. Nótese que las terminales de dirección a_0 y a_1 de todos los dispositivos de memoria se han conectado entre sí. De este modo, el mismo valor binario aplicado a estas dos terminales direcciona una palabra dentro de cada uno de los tres dispositivos de memoria. Sin embargo, nótese que las terminales CE de cada dispositivo no se encuentran conectadas entre sí. De tal modo, solo aquel dispositivo de memoria cuyo CE tenga un valor de

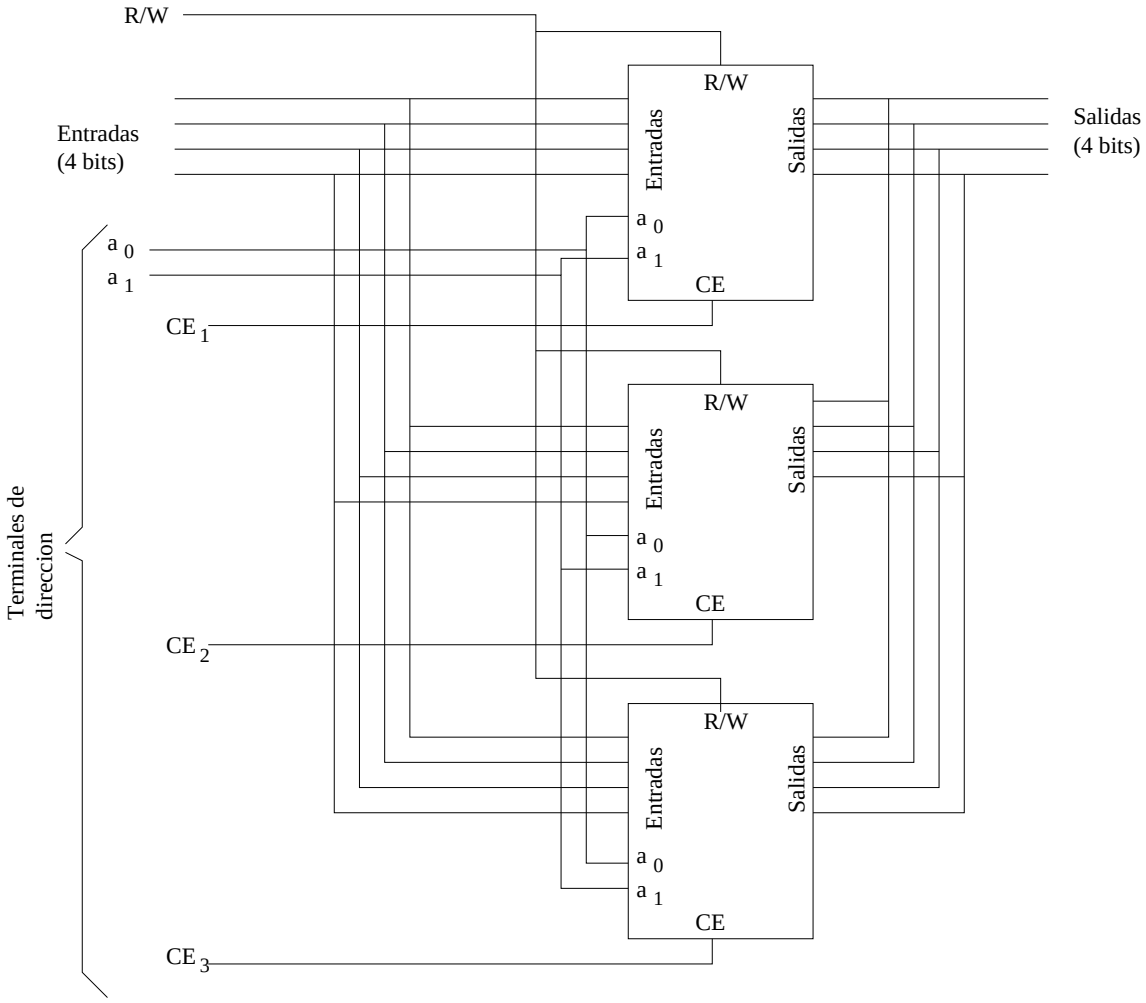

Figura 4.6: Una memoria de doce palabras construida mediante la paralelización de tres dispositivos de memoria de cuatro palabras.

1 será direccionado. Por ejemplo, si se desea direccionar la palabra 3 del dispositivo de memoria 2, entonces se requieren los siguientes valores:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1 \\ CE_1 &= 0 \\ CE_2 &= 1 \\ CE_3 &= 0 \end{aligned}$$

Las terminales de entrada correspondientes se conectan juntas. Por ejemplo, todas las terminales de entradas para el bit 0 se conectan juntas. De forma similar todas las terminales para los bits 1, 2, etc., así como las terminales de entrada para la selección de lectura o escritura (*Read/Write*). Ahora bien, supóngase que se desea escribir en la palabra 3 del dispositivo de memoria 2. Los valores dados anteriormente se deben colocar en las terminales de dirección, mientras que un valor de 1 debe colocarse en la terminal *Read/Write*. Solo entonces, la palabra 3 de la memoria 2 puede direccionarse. Así, los valores de las terminales de entrada a la memoria se pueden almacenar en la localidad deseada.

En el caso de las terminales de salida, tambiíen es necesario conectar las terminales juntas. Por ejemplo, todos los bits 0 de las terminales de salida se conectan. Recuérdese que cuando el valor de la terminal CE de una memoria es 0, las terminales de salida se desconectan. Por tanto, si se desea leer una palabra almacenada en la dirección 3 de la memoria 2, se aplican los valores anteriores para el direccionamiento y se coloca un valor de 0 a la terminal *Read/Write*. Ya que CE_1 y CE_3 tienen valor 0, las salidas de sus respectivas memorias se encuentran desconectadas. Igualmente, dado que CE_2 tiene valor 1, las salidas de esta memoria se conectan a las terminales de salida. De este modo, la palabra deseada aparece en las terminales de salida.

En la Figura 4.6 se muestra la paralelización de tres memorias de cuatro palabras cada una. En general, las memorias que se paralelizan contienen muchas más palabras, y esas palabras pueden contener más bits. Nótese que entonces hay una buena cantidad de variantes. Por ejemplo, la operación de las terminales de control como CE también puede variar de fabricante a fabricante. Así, los detalles reales para la conexión de memorias a fin de formar una memoria más grande y compleja puede variar respecto a los detalles que se han discutido hasta aquí, por lo que los detalles específicos de

las memorias deben obtenerse de los manuales que proveen los fabricantes. Aun cuando no se ha mencionado, las memorias pueden paralelizarse para formar palabras con un mayor número de bits.

4.4. Cintas y discos

Todos los tipos de memorias que se han discutido hasta este punto han sido memorias de acceso aleatorio. En tal tipo de memorias, cualquier dirección de memoria puede leerse o escribirse sin relación con ninguna otra dirección. Como se ha discutido anteriormente, esta es una memoria muy rápida. En esta sección, se mencionan otros tipos de memoria cuyo tiempo de respuesta es mayor, es decir, son más lentas para responder, pero que pueden proveer de mucho más espacio de almacenamiento a un costo mucho más bajo.

En realidad, estas formas más lentas de memoria pueden almacenar enormes cantidades de información binaria. Por ejemplo, pueden utilizarse para almacenar todos los registros de los cuentahabientes de un banco. Normalmente, esta cantidad de información es muy grande para almacenarse en la memoria principal, pero fácilmente puede guardarse en algunas cintas o discos. Cuando un programa que usa estos datos debe ejecutarse, sólo se requiere que los datos se transfieran de la cinta o disco a la memoria principal para ser procesados. El resultado puede ser de nuevo recolectado y almacenado en la cinta o disco.

Dado que se trata de información binaria, los programas también pueden almacenarse en cinta o disco. Cuando deben ejecutarse, se copian a la memoria principal. Las técnicas de almacenamiento masivo (y lento) que se discuten en esta sección resultan muy útiles para preservar datos y programas. También es común que este tipo de memoria se utilice como memoria principal, lo que hace la operación mucho más lenta, pero provee de una gran cantidad de espacio de memoria barata.

En la mayoría de los casos, las técnicas de almacenamiento masivas se basan en el uso de películas ferromagnéticas que almacenan datos. Su operación difiere al funcionamiento de la memoria principal en que aquí se efectúa realmente un movimiento, en el que la película ferromagnética se mueve respecto a un dispositivo o cabeza de lectura o escritura.

La Figura 4.7 muestra un dispositivo electromagnético que permite la grabación sobre superficies ferromagnéticas. En todos estos tipos de dispo-

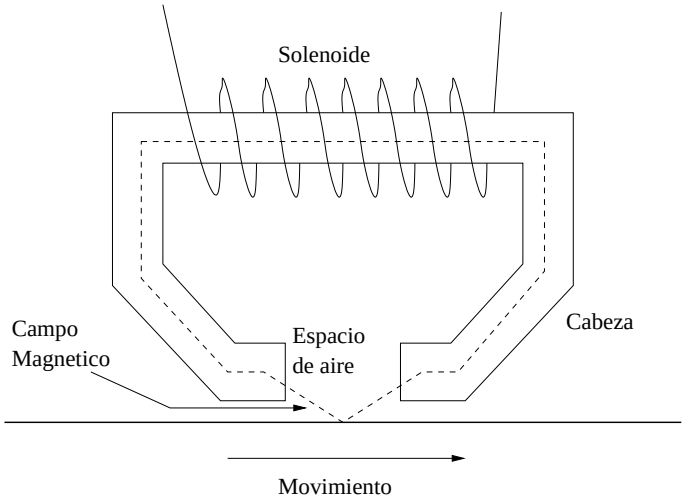

Figura 4.7: Diagrama de grabado en una película ferromagnética.

sitivos, la película de material ferromagnético se monta sobre un mecanismo de soporte como una cinta o un disco, que se encarga de mover el material bajo la acción de una cabeza. Esta se fabrica de material magnético, y comúnmente cuenta con un solenoide conductor que se enreda en uno de sus extremos. En el otro extremo, se tiene un espacio de aire colocado muy cercanamente al material ferromagnético. Nótese que el diagrama de la Figura 4.7 es en realidad mucho más grande que una cabeza magnética verdadera. El espacio de aire, así como la distancia entre la cabeza y la película, son muy pequeños. Cuando una corriente eléctrica se pasa a través del solenoide, se genera un campo magnético en la cabeza. En el punto del espacio de aire, el campo toca la superficie ferromagnética de la película, magnetizándola. Dado que la película se mueve bajo la acción del campo magnético, los pulsos de corriente que se apliquen al solenoide se reflejan como regiones cuyas moléculas tienen un tipo de arreglo magnético y regiones sin tal arreglo. La información, entonces, se ha escrito sobre la película. Nótese que se puede cambiar la dirección de la corriente, lo que invierte la polaridad de cada región magnética.

Ahora supóngase que no se aplica una corriente al solenoide, sino que solamente se mueve la película ferromagnética bajo la cabeza. Cuando una región magnetizada pasa bajo la cabeza, se genera un campo magnético en la cabeza. Cuando una región no-magnetizada pasa bajo la cabeza, no

se genera ningún campo. Por tanto, conforme la película se mueve bajo la cabeza, ésta cambia su característica magnética, lo que genera un voltaje en las terminales del solenoide. De este modo, la información almacenada en el medio ferromagnético se lee como un cambio de voltaje en las terminales del solenoide.

4.4.1. Cintas

Una forma de almacenamiento magnético involucra que la película ferromagnética se coloque sobre una cinta de plástico delgado. Esta cinta se enrrolla sobre rieles, lo que permite que pueda manipularse y almacenarse fácilmente. Frecuentemente, se utilizan varias cabezas en paralelo y perpendicularmente al movimiento de la cinta, lo que la divide en pistas o *tracks* a lo ancho de la cinta. Cada pista se utiliza para almacenar diferente información. Algunas pistas proveen a la computadora sobre información acerca de la localización o posición de datos en la cinta. Frecuentemente, las cintas tienen información grabada en su inicio que se conoce como *directorio* (*directory*). Este sirve como un índice, informando a la computadora dónde en la cinta se almacenan ciertos datos. En general, las cintas almacenan diferentes conjuntos de datos llamados *archivos* (*files*). Por ejemplo, un archivo puede contener los registros de las cuentas de los clientes de alguna tienda; otro archivo puede contener información sobre inventarios, etc. Si se desea leer un archivo en particular, es necesario saber su posición en la cinta. Supóngase, por ejemplo, que una pista puede almacenar hasta 100,000 bits. Entonces, un archivo puede comenzar en una posición particular, en el bit 17,325. La pista que se utiliza para determinar posiciones puede contener un valor de 1 por cada bit. Así, mediante contar el número de valores 1 a partir del inicio de la cinta, la computadora puede determinar la posición de un punto en la cinta que contiene los datos requeridos, y colocarlo bajo la cabeza lectora.

Las cintas se escriben y leen mediante una *unidad de cinta* (*tape drive*). Esta contiene componentes para la grabación y elementos mecánicos para el movimiento de la cinta. Las cintas puede removerse fácilmente o insertarse en la unidad de cinta, de modo que una cinta diferente puede usarse cada vez. Por tanto, la cantidad de información que puede almacenarse en cintas es casi ilimitado.

Una sola cinta puede almacenar gran cantidad de información. Por ejemplo, una cinta con 6 cabezas para almacenamiento de datos puede almacenar hasta 100 millones de bits. Por otro lado, la cantidad de tiempo que se necesita para leer o escribir un dato en memoria se conoce como *tiempo de acceso*

de esa memoria. La principal desventaja de las cintas como almacenamientos masivos es precisamente su gran tiempo de acceso. Una cinta debe ser re-embobinada completamente antes de que se pueda al menos buscar un dato. Algunas unidades de cinta pueden mover las cintas a gran velocidad (algo asi como 50 ó 400 cm/s). Sin embargo, el tiempo de acceso sigue siendo muy grande comparado con otros sistemas de almacenamiento. De hecho, las cintas se conocen como *sistemas de acceso secuencial*, ya que los datos se almacenan en una secuencia sobre la cinta, y es necesario mover la cinta secuencialmente por toda su longitud entre la posición actual y la posición deseada para obtener cualquier dato particular.

4.4.2. Discos

En este tipo de almacenamiento en memoria, la película ferromagnética se coloca sobre un disco. La información se almacena en pistas sobre la superficie del disco. Nótese que las pistas ahora son círculos concéntricos, y no en forma de espiral como los discos fonográficos. Se hace girar el disco sobre su centro a una alta velocidad. Para leer o escribir del disco, la cabeza se coloca sobre la pista deseada. Por tanto, para leer diferente información, la cabeza debe moverse sobre el radio del disco.

El almacenamiento en discos es mucho más rápido que en cintas, ya que se requiere pasar secuencialmente por las pistas en lugar de pasar por todos los datos almacenados. Así, en promedio, un disco requiere hacer media revolución antes de que la información deseada se encuentre bajo la cabeza, haciendo el tiempo de acceso mucho menor. El tiempo de acceso a disco se hace tan rápido como pueda la cabeza lectora moverse de una pista a otra, lo que representa un operación muy precisa y que toma algún tiempo. Para acelerar la operación, frecuentemente se colocan varias cabezas sobre un mismo disco. De este modo, la distancia en promedio que la cabeza debe moverse es menor, por lo que se disminuye el tiempo de acceso. El tiempo de acceso de un disco típico se encuentra en el orden de los milisegundos. Esto, por supuesto, es un promedio, ya que el tiempo de acceso depende del movimiento de las cabezas de una pista a otra. Este tiempo es mucho más veloz que una cinta, pero también resulta muy lento al compararlo con el tiempo de acceso de una RAM. El almacenamiento en disco no es secuencial, ya que no se lleva una secuencia de los datos del disco como se lleva en una cinta. Es por ello que a los discos se les conoce como almacenamiento de acceso directo.

Actualmente, existen dos tipos de discos para computadoras digitales: el *disco duro* (*hard disk*) y el *disco flexible* (*floppy disk*). El primero representa la forma original de los discos para almacenamiento masivo, y comúnmente se encapsulan en contenedores metálicos cerrados. Tienen capacidad de varios millones de bits, que año con año va en aumento.

Los discos flexibles son en realidad delgadas superficies de material ferromagnético almacenadas en un "sobre" de plástico. Esta es la razón precisa de su popular nombre. El sobre provee de ranuras al exterior que permiten la conexión del eje que hace rotar al disco, así como a la cabeza lectora. De hecho, el disco gira dentro del sobre. En comparación con los discos duros, los discos flexibles rotan a una velocidad mucho menor, y solo se cuenta con un juego de cabeza lectora y escritora. Por tanto, los discos flexibles tienen un tiempo de acceso mucho mayor que los discos duros. Además, tienen una capacidad limitada a apenas poco más de unos millones de bits. Aun cuando pueden leerse y escribirse varias veces, de hecho se gastan físicamente por el uso. Sin embargo, la principal ventaja de los discos flexibles es su bajo costo.

4.5. Códigos

La información almacenada en cualquier tipo de memoria digital es un número binario, una secuencia de unos y ceros. De hecho, todas las operaciones básicas de la computadora están diseñadas en términos binarios. Frecuentemente, es común que se desee almacenar información o trabajar con datos que contienen letras, números y otros caracteres útiles. Por ejemplo, una lista de nombres puede almacenarse para posteriormente ordenarse alfabéticamente. Este tipo de información se conoce con el nombre de *datos alfanuméricos*. Para almacenar datos alfanuméricos como secuencias de unos y ceros, se utilizan códigos. Un código es una cierta secuencia de números binarios que representan letras y símbolos. La computadora debe ser programada para procesar estos datos apropiadamente. Por ejemplo, un monitor funciona de modo que cuando en el teclado se escriben ciertas letras o números, estos aparezcan en la pantalla del monitor. Cuando el programa requiere que ciertas palabras se escriban, la computadora debe enviar las secuencias correctas de unos y ceros al monitor.

Los códigos de mayor uso en computadoras digitales son el código ASCII (*American Standard Code for Information Interchange*) y el código EBCDIC (*Extended Binary Coded Decimal Interchange Code*). La siguiente tabla

muestra los valores binarios o códigos de algunos de los datos alfanuméricos más comúnmente utilizados para representar letras, números y el espacio en blanco.

	ASCII	EBCDIC		ASCII	EBCDIC
A	100 0001	1100 0001	0	011 0000	1111 0000
B	100 0010	1100 0010	1	011 0001	1111 0001
C	100 0011	1100 0011	2	011 0010	1111 0010
D	100 0100	1100 0100	3	011 0011	1111 0011
E	100 0101	1100 0101	4	011 0100	1111 0100
F	100 0110	1100 0110	5	011 0101	1111 0101
G	100 0111	1100 0111	6	011 0110	1111 0110
H	100 1000	1100 1000	7	011 0111	1111 0111
I	100 1001	1100 1001	8	011 1000	1111 1000
J	100 1010	1101 0001	9	011 1001	1111 1001
K	100 1011	1101 0010	blanco	000 0000	0100 0000
L	100 1100	1101 0011			
M	100 1101	1101 0100			
N	100 1110	1101 0101			
O	100 1111	1101 0110			
P	101 0000	1101 0111			
Q	101 0001	1101 1000			
R	101 0010	1101 1001			
S	101 0011	1110 0010			
T	101 0100	1110 0011			
U	101 0101	1110 0100			
V	101 0110	1110 0101			
W	101 0111	1110 0110			
X	101 1000	1110 0111			
Y	101 1001	1110 1000			
Z	101 1010	1110 1001			

Nótese que en ambos casos, cada código provee de un valor binario único para cada letra y cada número. Tales valores son de gran utilidad al comunicar a la computadora con el ambiente exterior. Cuando se escribe en un teclado de la computadora, por ejemplo, éste se encarga de convertirlos en su representación binaria. En general, es conveniente que todos los símbolos se representen mediante secuencias de ceros y unos con una misma longitud. Así, lo que se transmite son los valores binarios de los códigos de las letras, los números y otros símbolos (ortográficos, de puntuación, etc.).

Nótese que los códigos de la tabla anterior se ordenan numéricamente: el valor binario del código representante de la A es menor que el valor binario de la B. Esto resulta útil en programas de computadora que permiten ordenar listas alfabéticamente. En la tabla solo se muestran las letras mayúsculas. Sin embargo, también hay códigos para las letras minúsculas. También es interesante que los tres bits más significativos del código ASCII para las letras mayúsculas son 100 para las letras de la A a la O, y 101 de la P a la Z. Si se reemplaza el 100 por 110 y el 101 por 111, se tienen los códigos ASCII para las letras minúsculas. Tal cambio es similar en el caso del código EBCDIC. En tal caso, los cuatro bits más significativos de los códigos cambian a minúsculas, en particular, si el 1100 se reemplaza por 1000, el 1101 se reemplaza por 1001, y el 1110 se reemplaza por 1010.

4.5.1. Códigos de detección de errores

Normalmente, las comunicaciones hacia una computadora y de la computadora hacia el exterior se hace mediante cables, alambres, o algún otro medio de transmisión de datos. La mayoría de estos medios funcionan en ambientes donde existe *ruido*, que se refiere a señales espurias que pueden llegar a afectar la transmisión. De hecho, por efecto de ruido, dentro de una comunicación digital un 0 puede aparecer después de la transmisión como un 1, y viceversa.

Para evitar este tipo de problemas, se han desarrollado códigos que permiten detectar si algún bit de una transmisión digital ha cambiado durante la misma. Uno de tales códigos se conoce como *código de bit de paridad*. En este código, se añade un bit extra al mensaje binario, que toma el valor de 0 ó 1 dependiendo de si el número de unos en la representación binaria de una letra tiene paridad par o impar, respectivamente. De tal modo, al recibir una secuencia que representa un símbolo, la computadora puede verificar la paridad del número total de unos en la representación. Si el número total de unos no corresponde con la paridad expresada por el bit de paridad, entonces el símbolo se rechaza y se puede enviar un mesaje de error solicitando la retransmisión del símbolo. Sin embargo, es notorio que si el medio de transmisión está sujeto a ruido, éste puede afectar a cualquier bit en el símbolo, puediendo cambiar dos bits, o hasta al propio bit de paridad. En ambos casos, el código de bit de paridad resulta inútil para detectar errores, por lo que se han desarrollado otros tipos de códigos de detección y corrección de errores más sofisticados que permiten una transmisión más segura de los datos binarios.

Capítulo 5

El cómputo digital básico

En este capítulo se discute la forma en que las operaciones de cómputo se realizan en una computadora digital. Se mencionan los detalles respecto a cómo los números se suman y se multiplican. También se mencionan otras operaciones importantes. Ya que todas las operaciones de cómputo se realizan sobre el sistema numérico binario, es necesario analizar las operaciones binarias para observar cómo se relacionan con los circuitos lógicos de una computadora digital.

5.1. La unidad aritmética básica

En una computadora digital, las operaciones aritméticas básicas se realizan en un componente digital llamado *Unidad Lógico-Aritmética* (*Arithmetic Logic Unit*, o simplemente ALU). Esta se relaciona con uno o más registros llamados *acumuladores*, donde se almacenan los resultados de las operaciones. Por simplicidad, se considera por ahora que la ALU se relaciona con un solo acumulador.

Una operación aritmética típica consiste en lo siguiente: un valor binario se lee de memoria y se da como entrada a la ALU, en donde se puede sumar, restar, multiplicar o dividir (dependiendo de la operación aritmética que se le instruya a la ALU) con el valor binario que se encuentre en el acumulador. El resultado de esta operación se almacena de nuevo en el acumulador. En el caso de una operación lógica, se realiza exactamente lo mismo, excepto que la operación debe ser lógica (AND, OR, NOT, XOR, etc.).

En la sección 3.4 se considera un sumador completo, cuyo circuito lógico se presenta en la Figura 3.17. En este capítulo se considera al sumador

completo como parte de la ALU. A continuación, la Figura 5.1 representa al sumador completo como un diagrama de bloque.

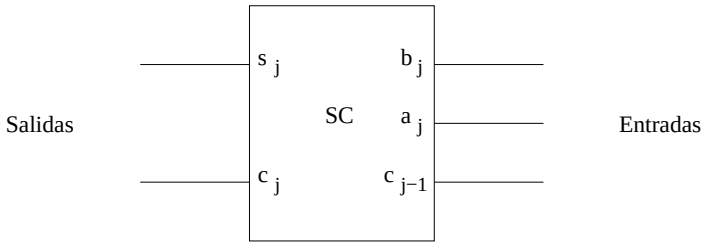

Figura 5.1: Diagrama de bloque de un sumador completo.

Este sumador completo (SC) suma tres números binarios de un solo bit b_j, a_j y c_{j-1}, para obtener su suma s_j y el acarreo a una siguiente etapa c_j. Recuérdese de la sección 3.4 que b_j y a_j representan los j-ésimos bits en un número binario, y que c_{j-1} es el acarreo proveniente de la columna anterior $j-1$. Para ilustrar esto, la Figura 5.2 presenta un sumador de tres bits, que permite sumar dos números binarios de tres bits ($a_2a_1a_0$ y $b_2b_1b_0$, respectivamente) utilizando sumadores para obtener un resultado también de tres bits ($s_2s_1s_0$).

Nótese que c_2, el acarreo de la columna más a la derecha, no se utiliza. Si la suma resulta en un número de cuatro bits, entonces se genera un *overflow*. Esto se discute en la Sección 2.3. El acarreo c_2 puede utilizarse entonces como indicador de que un overflow ha sucedido (particularmente si $c_2 = 1$). Esto puede generar, por ejemplo, que un mensaje de advertencia aparezca en la pantalla.

Para formar una ALU muy simple, se combina a continuación el sumador con un registro de entrada y salida paralela. Tal registro se discute en la Sección 3.8 (nótese que el circuito en la Figura 3.30 funciona como un regitro de entrada y salida paralela si las señales de control son $c_0 = c_1 = 0$, $c_2 = 1$, $c_3 = 0$). En la ALU de la Figura 5.3 se puede utilizar un registro más complejo, pero esto se evita por razones de simplicidad.

En la operación del circuito de la Figura 5.3 pueden verse dos líneas de control: ADD y CLR. Si se desea realizar una suma, debe colocarse $ADD = 1$ y $CLR = 0$. Supóngase que en tal caso, un número binario de tres bits se encuentra almacenado en el registro acumulador. La salida de cada flip-flop del registro se conecta con la entrada de su correspondiente sumador

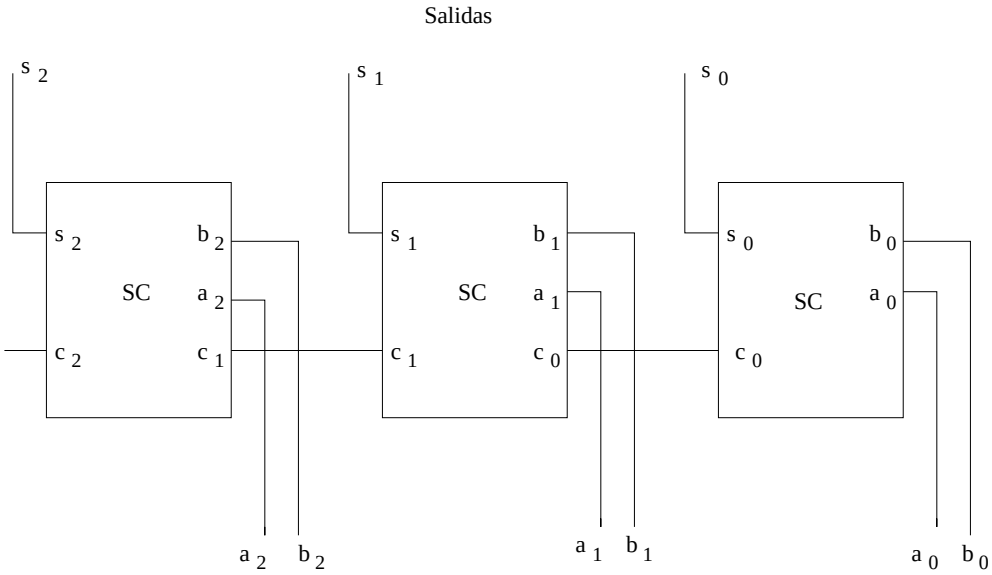

Figura 5.2: Un sumador de tres bits.

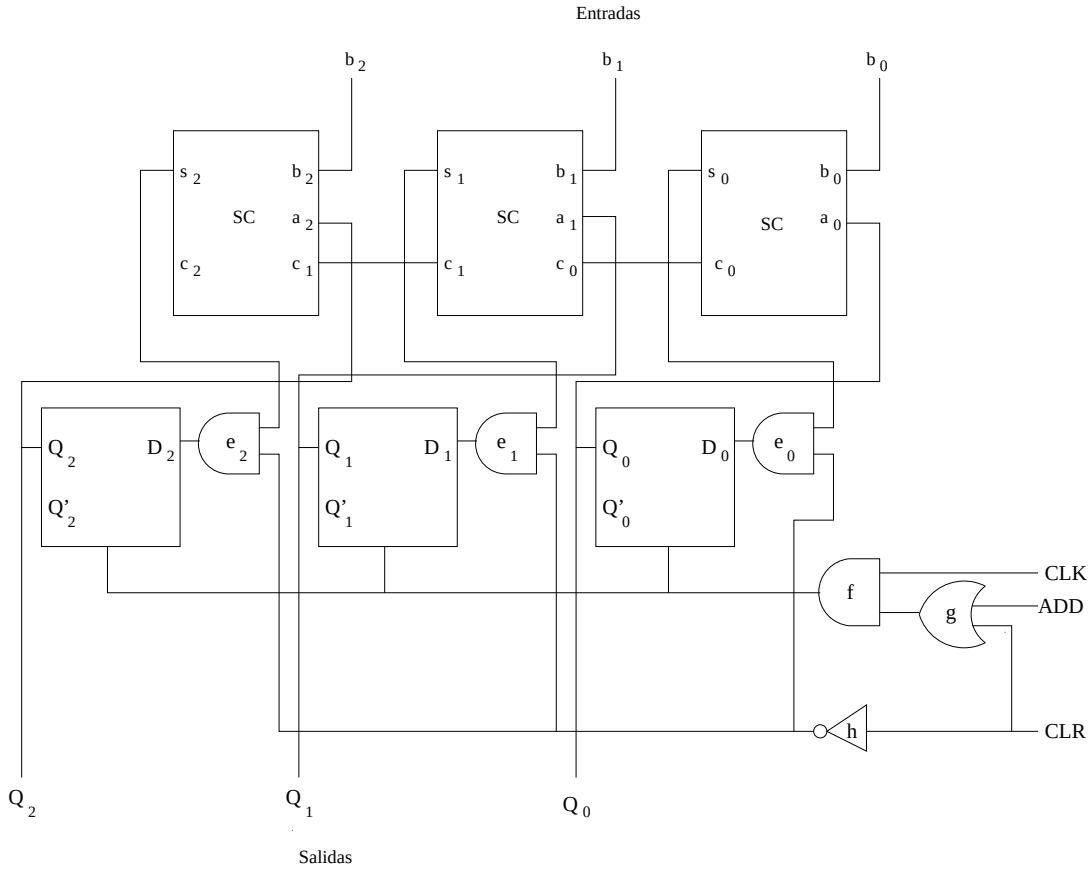

Figura 5.3: Una ALU simple de tres bits.

completo, de tal modo que el número almacenado en el acumulador es uno de los operandos a ser sumados. El otro número binario debe aplicarse en las entradas $b_2b_1b_0$. Después de realizar la adición, el resultado $s_2s_1s_0$ aparece en las salidas $Q_2Q_1Q_0$.

Ahora bien, el resultado debe aparecer almacenado en el acumulador. Ya que $CLR = 0$, entonces una entrada de cada compuerta AND e_2, e_1 y e_0 debe tener valor 1, de modo que $D_2 = s_2$, $D_1 = s_1$ y $D_0 = s_0$. Por lo tanto, *después* del siguiente pulso de reloj, el número que se almacena en el acumulador es el resultado de la suma. Nótese que se hace uso de flip-flops maestro-esclavo o de disparo por flanco. Así, la salida no cambia hasta *después* del pulso de reloj. Por tanto, si un número se encuentra almacenado en el acumulador, tal número permanece ahí hasta el siguiente pulso de reloj. Supóngase que el número a sumarse aparece a la salida de los flip-flops después del pulso de reloj que (arbitrariamente) puede nombrarse "pulso de reloj 1". Durante el tiempo en que el pulso de reloj está apagado (entre los pulsos de reloj 1 y 2) la suma se realiza. Entonces, al siguiente pulso de reloj (pulso de reloj 2) la entrada a los flip-flops es el resultado de la suma, pero la entrada a los flip-flops no ha cambiado. Así, las entradas a_2, a_1 y a_0 al sumador no cambian hasta el pulso de reloj 2. Los estados internos de los flip-flops han cambiado ya al valor deseado. Por tanto, solo hasta después del pulso de reloj 2, la salida de los flip-flops toma el valor del resultado de la suma. Nótese que toda esta operación supone que los sumadores son suficientemente rápidos, o que los pulsos de reloj se encuentran lo suficientemente apartados entre sí, de modo que la suma puede realizarse entre los dos pulsos de reloj.

Por otro lado, si ambas líneas de control CLR y ADD tienen valor 0, entonces ningún pulso de reloj o entrada se aplican a los flip-flops, por lo que el número almacenado en el acumulador permanece ahí. Si $CLR = 1$ y $ADD = 0$, entonces todas las entradas a los flip-flops toman valor de 0, ya que una de las entradas a las compuertas AND e_2, e_1 y e_0 tiene valor 0. Por lo tanto, se limpia el acumulador. Sin embargo, debe hacerse notar un hecho. El número de entrada $b_2b_1b_0$ debe almacenarse en un registro. Este registro podría ser parte de la ALU o ser externo a ella. Si es interno, entonces la ALU tiene asociados dos registros, que se conocen como acumuladores.

5.2. Aritmética modular

La suma hecha con una ALU difiere de la suma que se realiza a mano. Ya que el registro acumulador puede almacenar tan solo un número fijo de

bits, puede darse una condición de *overflow* (sobrecapacidad) en el resultado. Podría parecer que esta propiedad de la ALU es totalmente desventajosa, pero en esta sección y la siguiente se pretende mostrar cómo puede hacerse uso de ella.

Primero, es necesario familiarizarse con algunos detalles del almacenamiento de números, mediante un acumulador que trabaje en base 10. Un odómetro simple de automóvil puede utilizarse como ejemplo útil. Supóngase que se tiene un odómetro que llega hasta 99, y regresa a 0 (se utiliza el 99 en lugar de un número más grande como 99999 para contar con números pequeños en el ejemplo). Por tanto, las lecturas en el odómetro pueden ser:

00
01
02
⋮
10
11
12
⋮
96
97
98
99
00
01
02
⋮

El acumulador base 10 del odómetro puede almacenar números del 0 al 99, pero si se sobrepasa su capacidad con números adicionales, entonces vuelve a empezar de nuevo. De hecho, los acumuladores de las computadoras funcionan también así.

Ahora bien, supóngase que el odómetro muestra un valor de 26. No podemos estar seguros si el automóvil ha sido utilizado por 26 kilómetros o alguna de las siguientes distancias:

$$\begin{aligned} &26\\ &126\\ &226\\ &\vdots \end{aligned}$$

(Podría también tratarse de números negativos, pero este caso se considera en la siguiente sección).

Si se utiliza un registro que puede almacenar números hasta el 99, entonces los números 26, 126, 226, ... producen lecturas similares. De forma parecida, 47, 147, 247, 347, ... serían similares. Por esto, se introduce un símbolo que toma esto en cuenta. Considérese el símbolo $\equiv_{100}$. Para el primer ejemplo, se puede afirmar que:

$$26 \equiv_{100} 126 \equiv_{100} 226$$

El símbolo $\equiv_{100}$ significa entonces que esta equivalencia se da para un registro que almacena 100 números (0, 1, 2, ..., 99). Esto se lee como *equivalente módulo 100* ó *congruente, módulo 100*. Esto es, que 26 y 126 son equivalentes módulo 100. Nótese que el significado es que cada uno de estos resultados produce la misma salida del registro.

Si se desea encontrar los números que son equivalentes módulo 100 para un número dado, sólo se requiere sumar 100 y los múltiplos de 100 a tal número. Por ejemplo, para 26, sus equivalentes módulo 100 son 126, 226, 326, etc.

De hecho, no hay esencialmente ninguna diferencia si se utiliza un registro binario. Por ejemplo, utilizando un acumulador de tres bits, se tiene que los valores representables son:

000
001
010
011
100
101
110
111
000
001
010
011
100
⋮

Como ejemplo, supóngase que el registro tiene una lectura de 011_2. No se puede estar seguro cuántas "vueltas" se han dado sobre el mismo registro, de modo que 011_2 podría representar 3_{10}, 11_{10} ó 19_{10}. Utilizando la notación anterior, se puede afirmar que:

$$011 \equiv_8 1011 \equiv_8 10011$$

De forma similar, se puede escoger otro ejemplo:

$$100 \equiv_8 1100 \equiv_8 10100 \equiv_8 11100$$

Considerando esto de una forma diferente, supóngase que si se suma 1000_2 al registro de tres bits, se va a través de 8 pasos en un ciclo y se regresa al punto de inicio. De forma similar, para un acumulador de 4 bits, si se suma 16 al número binario almacenado, *no hay cambio* en el número almacenado. Aun cuando esto parece no llevar a ningún resultado práctico, en la siguiente sección se discute cómo estas ideas pueden utilizarse para simplificar los circuitos digitales de una computadora.

Antes de considerar mayores detalles de la aritmética modular, se discute la substracción binaria. Supóngase que se desea realizar la siguiente resta:

$$\begin{array}{rl} 1\,0\,1\,1 & \leftarrow \text{minuendo} \\ -\,0\,1\,1\,0 & \leftarrow \text{sustraendo} \\ \hline 0\,1\,0\,1 & \leftarrow \text{diferencia} \end{array}$$

Normalmente, la substracción se realiza columna por columna, comenzando por la derecha. Sin embargo, el bit de la tercera columna del sustraendo es mayor que el bit de la tercera columna del minuendo. La solución a este problema es "pedir prestado" una unidad de bit en la cuarta columna del minuendo. De este modo, la substracción se puede escribir como:

$$\begin{array}{r} 0\,10\,1\,1 \\ -\;\;0\;\;1\,1\,0 \\ \hline 0\;\;1\,0\,1 \end{array}$$

Ahora bien, considérese la siguiente substracción:

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

Ya que el sustraendo es mayor que el minuendo, la diferencia es negativa. La manera como se consideran números binarios negativos en la aritmética digital se explica en la siguiente sección.

5.3. Aritmética de complemento a 2

En la Sección 5.1 se discute un circuito para una ALU que puede utilizarse para realizar adiciones. Se utilizan sumadores completos para ir sumando cada uno de los bits. Sin embargo, si se desea realizar una resta, es siempre posible desarrollar un circuito lógico que la lleve a cabo. De manera análoga al sumador completo, se puede desarrollar un sustractor completo. Esto no se hace en la mayoría de los casos, ya que representaría usar circuitos adicionales y hacer más complejo el hardware, lo que incrementaría el costo y tamaño de la computadora. Por otro lado, también incrementaría su rapidez, lo que sería una ventaja.

En esta sección se presenta cómo tomar en cuenta las propiedades modulares de la ALU de modo que se pueda realizar una substracción utilizando el sumador modular. Así, no es necesario añadir hardware adicional para realizar restas, sino más bien se utiliza un procedimiento para trabajar los números negativos convenientemente.

Para entender cómo hacer una substracción sin un circuito sustractor, supóngase por ejemplo que se cuenta con un registro de 4 bits, y que se desea restar 0100_2 (4_{10}) de 1001_2 (9_{10}). Escribiendo la resta primero en base 10, más adelante se repite en base 2. Por tanto, se tiene que:

$$9 - 4 = 5$$

Ahora bien, supóngase que se suma 16_{10} al número almacenado en el registro. Como se discute en la sección anterior, el valor en el registro da un ciclo completo, por lo que *no hay cambio* en le número almacenado. Esto se puede escribir como:

$$16 + 9 - 4 = 5 + 16 \equiv_{16} 5$$

En seguida, se re-escribe la parte izquierda de la ecuación como:

$$15 + 1 + 9 - 4 = (15 - 4) + 1 + 9$$

Realizando la operación, se hace primero la resta de 15 menos 4 y luego se suma 1, y luego 9. Recuerde que se intenta evitar construir un sustractor, y por lo pronto, parece que no se ha logrado nada ya que todavía es necesario restar 4 de 15. Sin embargo, cuando esto se hace en forma binaria, la operación resulta muy simple. 15 es el número más grande que se puede almacenar en el registro de 4 bits:

$$15_{10} = 1111_2$$

Nótese que este número es solo una lista de unos. De modo que considerando la resta de este número, en el ejemplo 15 - 4 es:

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

El resultado de esta resta puede obtenerse directamente del sustraendo, mediante complementar cada uno de sus bits. De hecho, si se resta cualquier

valor binario de 1111_2, la diferencia es exactamente el complemento de cada bit del sustraendo. A esto se le conoce como *complemento a 1's*, y es una operación que no requiere hardware adicional. En general, todos los números binarios a procesar se almacenan en registros compuestos de flip-flops. Por ejemplo, supóngase que un número almacenado en el acumulador de la Figura 5.3. Si se desea complementar sus bits, se require sólamente tomar las salidas negadas de cada flip-flop. Este número se encuentra siempre disponible, de modo que no se requiere hardware adicional.

A continuación, se realiza la substracción 4 de 9 en binario. Usando este procedimiento, se tiene que $4_{10} = 0100_2$. Obteniendo el complemento a 1's, éste es 1011_2. El complemento a 2's se obtiene sumando 1 al complemento a 1's, lo que nos da 1100_2. Finalmente, para obtener la diferencia, se debe sumar este valor a $9_{10} = 1001_2$. Por lo tanto:

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

Nótese el bit más a la izquierda del resultado. Este bit no aparece en un registro de 4 bits. El registro solo contiene:

$$0101_2 = 5_{10}$$

Lo cual es la diferencia correcta. Si no se trabajara con registros, al sumar $15+1 = 16$, lo que sería una respuesta errada. Sin embargo, sumar 16 en un registro de 4 bits solo hace regresar al valor original de donde se comenzó, ya que el bit más a la izquierda se pierde. Así, se usa una propiedad de la aritmética modular como ventaja.

En términos generales, supóngase que se tiene un registro en el cual el número más grande que se puede almacenar es $R - 1$. Por ejemplo, en el caso del registro de 4 bits, se tiene que $R - 1 = 15$. Utilizando este sistema, supóngase ahora que se desea realizar la substracción:

$$D = A - B$$

Lo que realmente se realiza es:

$$D = (R - 1) - B + 1 + A$$

La operación $(R-1)-B$ es fácil de obtener, ya que sólo implica complementar cada bit de B. Las sumas siguientes producen el resultado correcto. De tal modo, solo se requieren sumadores y no sustractores para realizar la substracción.

5.3.1. Complemento a 2

El nombre de la operación que se discute anteriormente es *complemento a 2* de un número N. Se define de la siguiente forma:

$$C_2 = 2^k - N$$

donde k es el número de bits en el registro. Por ejemplo, si se tiene un registro de 4 bits, se tiene que $2^4 = 16_{10} = 10000_2$, y el complemento a 2 de $N = 0100_2$ es:

$$C_2 = 10000_2 - 100_2 = 1100_2$$

Nótese que si se toma $(R-1) - B + 1$ de la expresión para obtener la diferencia, en realidad lo que se hace es obtener el complemento a 2 de B. De la discusión previa, es fácil obtener el complemento a 2: se requiere sólo reemplazar cada bit por su complemento, y al resultado sumarle 1. Por lo tanto, si se trabaja con 4 bits, el complemento a 2 de 0100_2 es 1100_2.

Ahora bien, se sabe que el número almacenado en un registro es la representación binaria de un número decimal. Sin embargo, podría verse esto de manera diferente. La secuencia de bits, que se conoce como número binario, puede considerarse como un código que representa un número decimal. Es posible arreglar el código de modo que el número binario no fuera igual a su equivalente decimal. Más aun, es posible construir circuitos de compuertas lógicas que operen sobre estos números, y den el resultado correcto cuando se les convierta de nuevo en su forma decimal. Es notorio que el desarrollo de un código como el que se menciona sería una complicación innecesaria, y que muy probablemente los circuitos lógicos resultantes serían muy complicados. Por lo tanto, esto no se hace para los números positivos. Sin embargo, cuando es necesario trabajar con números negativos, es verdaderamente necesario usar algún tipo de código, ya que los registros almacenan ceros o unos, pero no símbolos matemáticos como el punto o el signo menos.

De los muchos códigos que pueden emplearse, se discute a continuación uno que es ampliamente usado y que hace el cómputo simple. Logra esto porque los resultados son correctos en términos de la aritmética modular.

De hecho, el complemento a 2 puede usarse para obtener tal codificación. Supóngase que se tiene un registro de 4 bits. Los números que almacena son:

$$
\begin{array}{ll}
\vdots & \\
1011 & \\
1100 & \\
1101 & \\
1110 & \\
1111 & \leftarrow \quad 2_{10} - 3_{10} = -1_{10} \\
0000 & \\
0001 & \\
0010 & \leftarrow \quad 2_{10} \\
0011 & \\
0100 & \\
0101 & \leftarrow \quad 2_{10} + 3_{10} = 5_{10} \\
0110 & \\
0111 & \\
1000 & \\
\vdots &
\end{array}
$$

Supóngase que se suma 3_{10} a un número. Esto significa ir bajando en la numeración hasta llegar a tres posiciones más abajo. Por ejemplo, considérese el número 2_{10}. De su posición, al sumarle 3_{10}, se obtiene la posición 5_{10}. Ahora bien, supóngase que se resta 3_{10}. Esto equivale a moverse hacia arriba en la numeración tres posiciones. El resultado es $1111_2 = 15_{10}$. Sin embargo se sabe que:

$$15_{10} \equiv_{16} -1_{10}$$

Esto es, que 15_{10} es equivalente módulo 16 a -1_{10}.

Si se define que 1111_2 represente a -1_{10} en un código, entonces se llega a la respuesta correcta para la resta. De forma similar, 1110_2 representa al -2_{10}. De este modo, se tiene una codificación para el registro de 4 bits que permite representar números de -7_{10} a 7_{10}:

Número decimal	Número binario
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

La información de esta tabla es consistente con la discusión anterior, pero nótese que los números negativos son simplemente el complemento a 2 de sus equivalentes positivos. Por ejemplo, para obtener la representación de -7_{10}, se toma el número $7_{10} = 0111_2$ y se obtiene su complemento:

$$1000_2 + 0001_2 = 1001_2 = -7_{10}$$

De forma similar, es posible obtener todos los otros números negativos en la tabla. Es notorio que si el bit más significativo es 0, se trata de un número positivo, mientras que si es 1, es negativo. *Nótese que un número no se hace negativo simplemente por reemplazar su bit más significativo de 0 a 1.*

Si se tienen b bits en un registro, se utilizan sólamente $b-1$ de ellos para representar la magnitud de un número. Por ejemplo, en la tabla anterior se tienen 4 bits, pero los números que se representan van de -7_{10} a 7_{10}. Si se trabaja solo con números positivos, entonces el rango de valores sería de 0_{10} a 15_{10}.

5.3.2. Uso del Complemento a 2 para la substracción

Utilizando un sistema de 4 bits y trabajando con la substracción, se presenta a continuación que el uso del complemento a 2 para representar números negativos realmente funciona. Supóngase que se desea restar 4_{10}

de 7_{10}. Esto resulta equivalente a sumar $7_{10} + (-4_{10})$. Considerando sus equivalentes binarios, se tiene que el complemento a 2 de $4_{10} = 0100_2$ es 1100_2. Sumando, se tiene que:

$$0111_2 + 1100_2 = (1)0011_2$$

Ignorando el bit más significativo entre paréntesis, se tiene que $0011_2 = 3_{10}$, que es la diferencia correcta al operar $7_{10} - 4_{10}$. De nuevo, nótese que se utilizan las propiedades de la aritmética modular para obtener el resultado correcto.

Considérese otro ejemplo: restar 6_{10} de 3_{10}. Entonces, se opera con el complemento a 2 de 0110_2, que es 1010_2, de modo que la resta queda como:

$$0011_2 + 1010_2 = 1101_2$$

De la tabla anterior se observa que $1101_2 = -3_{10}$, el resultado correcto.

Para este punto, es necesario aclarar una cosa: en algunos de los ejemplos anteriores se ha ignorado el bit de sobrecapacidad (*overflow*) mediante ignorar o eliminar el bit más a la izquierda del resultado. En realidad, este *no es* un ejemplo de *verdadera sobrecapacidad*. La única ocasión en que realmente sucede una sobrecapacidad es cuando el tamaño del resultado de una operación es demasiado grande para representarse con los bits disponibles en el registro. Por ejemplo, para el registro de 4 bits, si se realiza la suma:

$$7_{10} + 5_{10} = 0111_2 + 0101_2 = 1100_2$$

El resultado, de acuerdo con la tabla, es $1100_2 = -4_{10}$. El resultado debería ser 12_{10}. Por tanto, una verdadera sobrecapacidad ha ocurrido, y se obtiene un resultado incorrecto. Para este sistema de registro de 4 bits, una sobrecapacidad verdadera ocurre cuando el resultado es mayor que 7_{10}. De manera similar, en un resgitro de k bits, una verdadera sobrecapacidad ocurre cuando la magnitud de un resultado excede $2^{k-1} - 1$.

5.4. Multiplicación y división

Esta sección discute la multiplicación y división como se realizan por una computadora digital. Se inicia mediante describir qué se hace normalmente cuando se multiplican dos números binarios manualmente. De hecho, las reglas de la multiplicación binaria siguen exactamente las mismas reglas de

la multiplicación decimal que se aprende en la educación básica. Sin embargo, el uso de números binarios tiende a simplificar en mucho la operación. Supóngase que se desea multiplicar 1101_2 por 1011_2:

$$
\begin{array}{r}
1\,1\,0\,1 \\
\times\ 1\,0\,1\,1 \\
\hline
1\,1\,0\,1 \\
1\,1\,0\,1 \\
1\,1\,0\,1 \\
\hline
1\,0\,0\,0\,1\,1\,1\,1
\end{array}
$$

Se comienza multiplicando 1101_2 por el dígito menos significativo (1_2), para obtener el primer producto parcial. En seguida, se multiplica por el segundo dígito (1_2), produciendo el segundo producto parcial que se recorre hacia la izquierda una posición. Se continúa multiplicando con el tercer dígito (0_2). Como su valor es 0, simplemente se considera un recorrido de dos posiciones para la siguiente multiplicación. Al final, se realizan las sumas de cada columna, produciendo el producto final. De hecho, este proceso se repite sin importar el número de dígitos.

La multiplicación binaria resulta más sencilla que la multiplicación decimal, debido a que los productos parciales en binario se obtienen por multiplicar por 0 ó 1, lo que resulta en un recorrido a la izquierda en el caso del 0, o una copia del valor multiplicado en el caso del 1.

A continuación se describe cómo el proceso de multiplicación se realiza en una computadora digital. Considérese, por ejemplo, que el producto de dos números de 4 bits da como resultado un número de 8 bits. De este modo, se considera el uso de registros de 8 bits. Esto es si se multiplican dos números como por ejemplo 00001011_2 y 00001101_2. Nótese que los multiplicandos deben ser lo suficientemente pequeños para que su producto no exceda el tamaño del resgistro, lo que es el caso que se discute. Ahora bien, supóngase que se tienen a ambos multiplicandos en dos registros de corrimiento. Primero, se examina el bit más a la derecha del registro del primer multiplicando. Si es 1, entonces el segundo multiplicando se pasa a la ALU, donde se suma. Si el bit más a la derecha es 0, entonces se evita la suma. Se puede llevar a cabo esta operación mediante controlar la suma utilizando el bit más a la derecha del primer multiplicando conectado a la ALU. De este modo, la suma se realiza sólo si el bit más a la derecha del primer factor tiene valor de 1.

Cada vez que se encuentra un 0 o un 1 en el primer multiplicando, es necesario recorrer el contenido del registro del segundo multiplicando a la izquierda, de modo que los productos parciales se vayan sumando. El primer multiplicando se recorre a la derecha, lo que causa que el bit más a la derecha se pierda. Recuérdese que este bit controla la multiplicación, pero una vez utilizado, se pierde. Así, mediante corrimientos de los registros es posible multiplicar dos números binarios. El proceso debe repetirse por cada bit en el primer multiplicando. Cuando se termina, el contenido en el registro acumulador contiene el producto deseado.

Algunas computadoras digitales cuentan con circuitos cuya función es realizar la multiplicación. Tales circuitos contienen registros y acumuladores de modo que cuando se les provee de dos números binarios, producen su producto. Un circuito de este tipo se llama *multiplicador alambrado* (*hardwired multiplier*). No todos los multiplicadores alambrados funcionan como se ha descrito anteriormente. En ocasiones, se hacen simplificaciones a los circuitos. Sin embargo, la idea básica es la misma. Si una computadora no cuenta con un multiplicador alambrado, debe ser programada paso a paso mediante instrucciones que realicen la multiplicación. En el siguiente capítulo se discute tal labor de programación, que muchas veces involucra muchos pasos. Pero si se cuenta con un multiplicador alambrado, entonces la única instrucción a ejecutar es el comando para realizar multiplicación; los circuitos de la computadora, sin embargo, son más complejos.

En seguida se discute cómo una computadora digital realiza la división. Así como la multiplicación involucra sumas, la división involucra restas. Supóngase que se quiere dividir 10001111_2 entre 1011_2 (esta es la operación inversa de la multiplicación realizada anteriormente). El primer paso es verificar si 1011_2 es mayor que 1000_2. Ya que lo es, entonces no se intentaría el primer paso de la división. Sin embargo, la computadora no puede notar esto, de modo que realiza la siguiente división:

```
1011 | 10001111
     - 1011
     ------
```

Después de realizar la resta, se prueba si el resultado es negativo (se discute un circuito que prueba si un número es negativo más adelante). Si lo es, entonces no se realiza el siguiente paso en la división. La computadora se programa, de modo que todos los pasos anteriores y erróneos se ignoren.

En el ejemplo, es necesario hacer un nuevo intento:

```
                0 1
          ┌──────────
1 0 1 1   │ 1 0 0 0 1 1 1 1
        −   1 0 1 1
          ──────────
            0 1 1 0 1 1 1
```

(De hecho, la substracción se hace mediante el complemento a 2). De este modo, el procedimiento se repite hasta obtener el cociente 01101_2.

Las computadoras pueden también contar con un *divisor alambrado* (*hardwired divider*), o pueden ser programadas para dividir números. El segundo caso resulta en un circuito más sencillo, pero requiere de mucho tiempo de programación.

5.5. Números de punto flotante

Es frecuente que en cálculos científicos o tecnológicos es necesario trabajar con números muy grandes o muy pequeños. Para evitar escribir largos números con muchos dígitos, se acostumbra expresar tales números en términos de una expresión que consta de un número multiplicado por 10 elevado a alguna potencia. La potencia es un número entero. Por ejemplo, las siguientes representaciones son equivalentes:

$$1763000 = 0{,}1763 \times 10^7$$

Similarmente:

$$0{,}00000000112304 = 0{,}112304 \times 10^{-7}$$

En una computadora, a esta representación se le conoce como *notación de punto flotante* (*floating point notation*), ya que el cambio en el exponente efectivamente permite mover la posición del punto decimal, que "flota". En seguida se definen los términos utilizados en esta notación. Por ejemplo, los valores enteros 7 y -7 de las expresiones anteriores se les llama *exponente*, mientras que a 0,1763 y 0,112304 se les conoce con el nombre de *mantisa* o *parte fraccional*. De hecho, el primer nombre tiende a ser ambiguo, ya que este valor no es la mantisa de un logaritmo, de modo que aquí se prefiere el término parte fraccional.

Cuando se utilizan números de punto flotante en las computadoras digitales, normalmente se trabaja con potencias de 2 en lugar de potencias de 10. Por ejemplo, $64_{10} = 2^7$ puede escribirse como:

$$0{,}1 \times (10_2)^{1000_2}$$

Queda tal vez más claro de la siguiente forma:

$$0{,}1 \times 2^8$$

Es decir, la parte fraccional en binario, y el exponente en decimal (pero recuérdese, la computadora solo trabaja en binario).

A continuación, se aclara un punto que comúnmente causa confusión. Hay una diferencia entre el tamaño de un número y el número de cifras significativas. Por ejemplo, supóngase que se desea expresar una longitud en términos de millonésimas de centrímetro como 1,736,000. Aun cuando la medida real fuera de 1,736,401, no es posible medir con la suficiente exactitud para notar esto. En tal caso, se dice que el número (1,736,000) se maneja con cuatro cifras significativas.

Ahora bien, cuando se trabaja con números en punto flotante, en realidad se trabaja con dos números: la parte fraccional y el exponente. Cada uno de ellos debe almacenarse aparte, pero normalmente se les puede incluir en el mismo registro. Por ejemplo, si el registro tiene 12 bits, los 8 bits menos significativos pueden utilizarse para almacenar la parte fraccional, y los 4 bits restantes para el exponente.

Supóngase que se tiene el número:

$$0{,}00010110111 \times 2^{-6} = 0{,}10110111 \times 2^{-9}$$

Requeriría de 11 bits para almacenar la parte fraccional de la expresión a la izquierda de la ecuación, mientras que sólo requiere de 8 bits para almacenar la parte fraccional de parte derecha de la ecuación. Y sin embargo, son el mismo número. Si la parte fraccional se almacena en un registro de 8 bits, entonces la expresión a la izquierda del igual tendría que ser $0{,}00010110_2$. Esto representa una pérdida de cifras significativas, y por tanto, una pérdida en exactitud. Para evitar esto, los exponentes de los números de punto flotante se ajustan para que el bit inmediatamente a la derecha del punto binario tenga valor 1. Se dice que tal número ha sido *normalizado* o *escalado*. En tal caso, no se desperdicia espacio de memoria almacenando

ceros. Nótese que el punto binario y el cero a la izquierda *no* se almacenan. Por ejemplo, la parte fraccional almacenada de la última expresión sería 10110111_2. En un valor punto flotante escalado, la posición del punto binario se considera inmediatamente a la izquierda del número, y por tanto no se necesita almacenar.

Respecto al signo, tanto la parte fraccional como el exponente pueden ser positivos o negativos. Por tanto, debe haber una regla para bits de signo: un bit se utiliza para almacenar el signo del exponente, y otro bit para el signo de la parte fraccional (de hecho, la discusión sobre complemento a 2 en la Sección 5.3 provee de un procedimiento para indicar números negativos).

Para averiguar de qué tamaño es el número con que se puede trabajar en una computadora, es necesario hablar de almacenamiento. Un registro almacena un número dado en bits. Grupos de 8 bits se conocen con el nombre de *byte*. Normalmente, todos los registros dentro de una computadora tienen la misma longitud. A esto se le conoce con el nombre de *tamaño de palabra* (*word size*) de una computadora. Valores típicos de tamaño de palabra son 8, 16, 32, y 64.

En seguida, se analiza qué tan grande o qué tan pequeño puede ser un número en la computadora. Supóngase que el tamaño de palabra es de 32 bits. Estos deben almacenar tanto el exponente, como la parte fraccional, cada uno con su signo. Dos bits de signo de 32 bits dejan 30 bits disponibles. Normalmente, 6 bits se utilizan para el exponente, mientras que los otros 24 se dejan para la parte fraccional. Ahora bien, el número más grande que puede almacenarse como exponente es:

$$2^6 - 1 = 63_{10}$$

Y por lo tanto:

$$2^{63} = 9{,}223 \times 10^{18}$$

Este es un número muy grande. De forma similar, 2^{-63} es un número muy pequeño. Por lo tanto, la gama de valores numéricos de punto flotante que se pueden expresar en la computadora varían al ir dando valores binarios a la parte fraccional y al exponente.

Se supone anteriormente que en la representación de punto flotante el exponente se aplica únicamente a una potencia de 2. En realidad, se puede

utilizar cualquier otro número. Por ejemplo, en algunas computadoras, se trata de potencias de 16; circuitos digitales apropiados se desarrollan para tomar en cuenta esto. Ahora bien, si se cuenta con 6 bits para almacenar la magnitud del exponente, se tiene el mismo número 63_{10} para el exponente. Sin embargo, ahora se trata de una potencia de 16_{10}. De este modo, la parte exponencial puede ser tan grande como:

$$16^{63} = 7{,}237 \times 10^{75}$$

Esto es un número mucho más grande que el obtenido mediante utilizar el exponente como potencia de 2.

En cuanto a la parte fraccional, ésta se almacena en los restantes 24 bits. Por ejemplo, un número almacenado puede ser:

$$0{,}10110011110011001011 0011$$

Mientras más bits se almacenen, más precisión se tiene. Supóngase que sólo se almacenan 23 bits. El último bit a la derecha se perdería. Este bit representa un valor de:

$$2^{-24} = 0{,}0000000596_{10}$$

De hecho, al perder este valor, se tiene que los números $0{,}12461281_{10}$ y $0{,}12461282_{10}$ se almacenarían de la misma forma binaria. Nótese que hay siete cifras significativas en tales números porque hay siete ceros a la derecha del punto decimal en la representación base 10 de 2^{-24}.

Estos números son bastante precisos. En la mayoría de las aplicaciones científicas y tecnológicas, no es poslbe medir datos con tal precisión. Por otro lado, se pierde precisión cuando se realizan operaciones. Por ejemplo, cuando se multiplican dos números de 8 bits en su parte fraccional, se obtiene un número con 16 bits en la parte fraccional del resultado. Si se eliminan los 8 bits menos significativos, se pierde exactitud y precisión. Algunos programas de computadora requieren muchas operaciones, y si se va perdiendo precisión en cada una de ellas, el resultado final tendrá un gran error. Por tanto, se requiere de una gran precisión cuando se realizan operaciones complejas, y en ocasiones, siete u ocho cifras significativas resultan insuficientes. Sin embargo, la mayoría de las computadoras pueden ser programadas de modo que dos o más registros o localidades de memoria puedan utilizarse para almacenar un solo número, lo que provee de precisión adicional. Esta acción de agrupar registros se conoce comúnmente como *doble precisión*.

Anteriormente, se considera un tamaño de palabra de 32 bits. La mayoría de las computadoras actuales tienen tal tamaño de palabra, pero en el pasado, muchas computadoras contaban tan solo con 8 o 16 bits. Aun cuando puede parecer que estas computadoras no eran adecuadas para realizar operaciones, este no es el caso. Muchos programas realizan operaciones numéricas que no requieren muchas cifras significativas o números muy grandes o pequeños. En tal caso, no resulta necesario utilizar muchos bits para almacenar la parte fraccional y el exponente. Además, hay un gran número de programas que simplemente no se utilizan propiamente para realizar operaciones numéricas. Por ejemplo, programas que ordenan listas de nombres, mantienen un inventario, juegos, o controlan sistemas de alarma no requieren complicadas operaciones numéricas, dado que no trabajan con números con muchos dígitos. Y si se encuentra apropiadamente programada, una computadora con un tamaño pequeño de palabra puede utilizarse para cómputo científico o tecnológico. Como se discute anteriormente, se pueden utilizar varios registros o localidades de memoria para almacenar un solo número. Es claro que esto efectivamente representa una pérdida en la cantidad de memoria disponible. Sin embargo, se puede lograr operaciones más complejas si se cuenta con la memoria suficiente para mantener la precisión.

5.5.1. Suma y resta

Cuando se suman o restan dos números representados en punto flotante, los exponentes de ambos números deben tener el mismo valor. Por ejemplo, si se desea sumar $0{,}1001 \times 2^8$ y $0{,}1011 \times 2^7$ se requiere "mover" el punto de alguno de los dos valores, como por ejemplo:

$$0{,}1001 \times 2^8 + 0{,}01011 \times 2^8 = 0{,}11101 \times 2^8$$

Si solo se almacenan 4 bits para la parte fraccional, la operación provoca una pérdida de precisión debido a que en lugar de almacenar la respuesta correcta ($0{,}11101 \times 2^8$) se alamcena $0{,}1110 \times 2^8$. A esto se conoce como *error de redondeo*. Comúnmente, estos errores son lo suficientemente pequeños como para ser ignorados. Sin embargo, cuando se desea mayor precisión, una solución puede ser el uso de doble precisión.

5.5.2. Multiplicación y división

Cuando se multiplican o dividen números en punto flotante, se utilizan las reglas ordinarias de la multiplicación y la división de números con exponentes. Supóngase que se tiene dos números:

$$A = F_a \times 2^{E_a}$$
$$B = F_b \times 2^{E_b}$$

donde F_a y F_b son las partes fraccionarias, y E_a y E_b son los exponentes. Para multiplicar ambos números, se multiplican las partes fraccionarias y se suman los exponentes:

$$A \times B = (F_a \times F_b) \times 2^{E_a + E_b}$$

Por ejemplo, supóngase que:

$$A = 0{,}1011 \times 2^5$$
$$B = 0{,}1101 \times 2^7$$

Entonces:

$$A \times B = 0{,}010001111 \times 2^{12}$$

En una computadora digital, este número sería escalado y almacenado de la siguiente forma:

$$A \times B = 0{,}10001111 \times 2^{11}$$

Nótese que hay más bits en la parte fraccional del producto que en A o B, lo que significa que fácilmente puede ocurrir un error de redondeo.

Para realizar la división, se utiliza la siguiente expresión:

$$\frac{A}{B} = \frac{F_a}{F_b} \times 2^{E_a - E_b}$$

Es decir, cuando se realiza una división en punto flotante, se dividen las partes fraccionales y se restan los exponentes. Por ejemplo:

$$\frac{0{,}100001111 \times 2^{11}}{0{,}1011 \times 2^5} = 0{,}01101 \times 2^6 = 0{,}1101 \times 2^5$$

5.6. La unidad lógica-aritmética (ALU)

En la Sección 5.1 se discute una ALU sencilla que puede sumar dos números y almacenar su suma en un registro. En esta sección se discute una ALU más compleja que puede realizar varias operaciones, y que es similar a la ALU interna del procesador de una computadora pequeña.

Donde la ALU sencilla de la Figura 5.3 usa flip-flops D para el registro, en la ALU actual se utilizan flip-flops JK, ya que se obtiene un control adicional mediante su uso. Para cada operación que se desee obtener, se muestra un circuito sencillo. Después de discutir todas las operaciones, se muestra cómo todos estos circuitos simples pueden ser combinados. Ya que todas las etapas de la ALU son idénticas, se trabaja con una sola etapa. Por ejemplo, la ALU de la Figura 5.3 conisiste de tres etapas idénticas.

La ALU descrita aquí requiere de muchas terminales de control. Cada una de ellas se utiliza para controlar una sola operación. Se supone por simplicidad que la señal digital en las terminales de todas las demás operaciones tiene valor 0. Por ejemplo, en la Figura 5.3 hay dos terminales de control ADD y CLR. Si se desea que la ALU realice una suma, entonces ADD = 1 y CLR = 0.

Muchas señales se le aplican a los flip-flops JK mediante un par de compuertas OR como se muestra en la Figura 5.4. El significado de este circuito se aclara cuando se haya discutido la ALU completa.

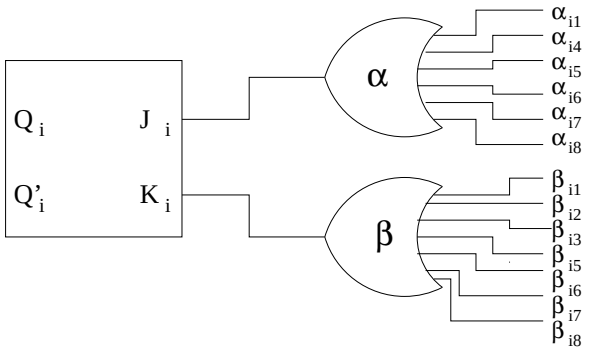

Figura 5.4: Entradas al i-ésimo flip-flop de la ALU.

5.6.1. Adición e incremento

Considérese un circuito que suma un número B al número almacenado en el acumulador. Se supone que hay $N+1$ bits almacenados en el acumulador; los dígitos del número B son:

$$b_N b_{N-1} \ldots b_3 b_2 b_1 b_0$$

Ahora bien, considérese la etapa de la ALU que suma el i-ésimo bit y almacena la suma resultante. Si los dígitos del número que está almacenado en el acumulador son:

$$a_N a_{N-1} \ldots a_3 a_2 a_1 a_0$$

entonces la i-ésima etapa realiza la suma $b_i + a_i + c_{i-1}$, donde c_{i-1} es el acarreo de la etapa anterior. El circuito que se considera aquí en realidad realiza dos funciones: además de sumar el número B al contenido del acumulador, también es capaz de añadir el valor 1 al contenido almacenado (en la Sección 5.3 se explica que añadir 1 es parte de obtener el complemento a 2). Por tanto, el circuito tiene dos terminales de control: una etiquetada ADD para realizar una suma ordinaria y otra etiquetada INC para incrementar el contenido del acumulador en una unidad.

Considérese la operación del circuito digital de la Figura 5.5. Si la señal ADD tiene valor 1, entonces INC debe tener valor 0. Es decir, solo una de estas dos señales de control puede tener valor de 1 en un momento dado.

En tal situación, el bit b_i se aplica en la entrada b_i del sumador completo (SC). Si el i-ésimo bit de B tiene valor 1, entonces $b_i = 1$; si tiene valor 0, entonces $b_i = 0$. Además, una entrada de cada compuerta AND **a** y **b** tiene valor 1, y por lo tanto:

$$J_i = s_i$$
$$K_i = s_i'$$

En tal caso, el flip-flop JK funciona como un flip-flop D. De hecho, este circuito por lo pronto funciona igual a una de las etapas de la ALU en la Figura 5.3.

Ahora bien, supóngase que ADD = 0 e INC = 1. Una entrada a la compuerta AND **e** tiene valor 0, por lo que el bit b_i no pasa por la compuerta. El circuito aún funciona como un sumador con entrada cero, excepto por

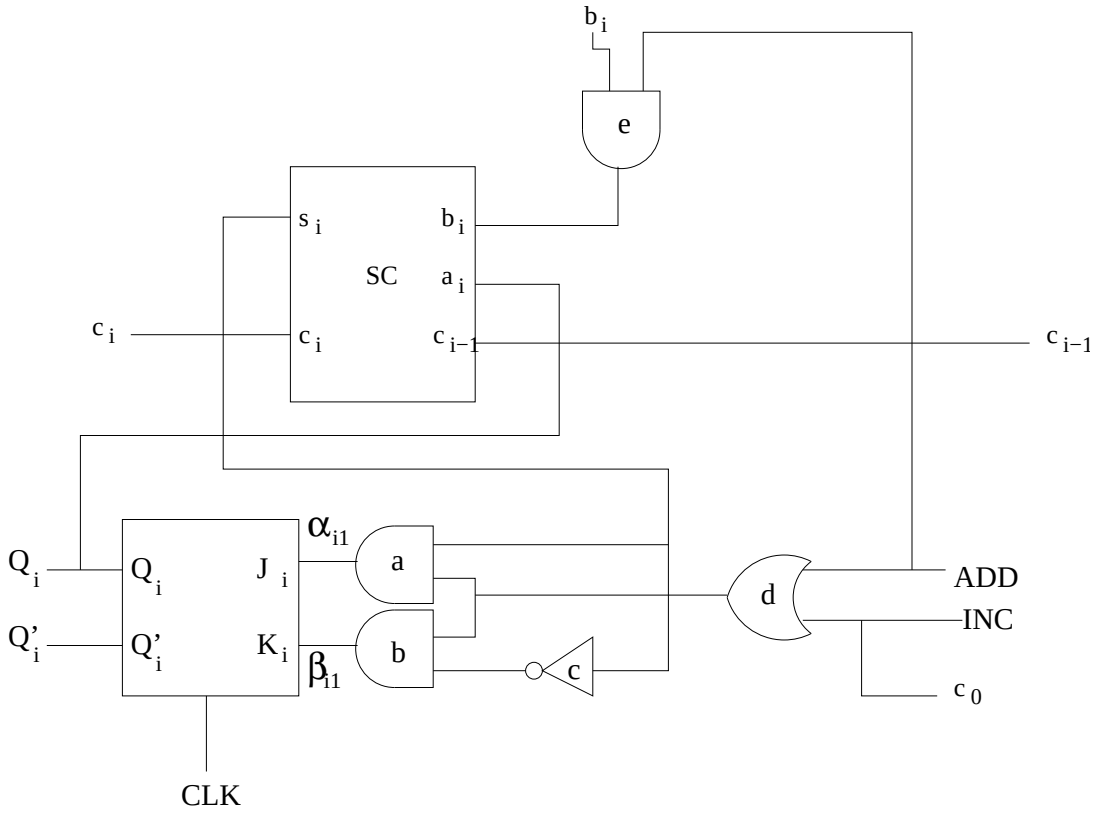

Figura 5.5: Etapa de la ALU que realiza ya sea la suma o el incremento.

una diferencia. La entrada c_0 del primer sumador no se utiliza durante la suma, ya que no hay acarreo de la columna más a la derecha. Cuando INC = 1, entonces tal entrada tiene valor de 1. Nótese que el primer sumador completo toma la suma $b_0 + a_0 + c_0$. Por lo tanto, una entrada c_0 actúa de la misma forma en que actúa b_0. Así, cuando INC = 1, entonces $c_0 = 1$, lo que hace que un 1 se sume al valor almacenado en el acumulador. Después del siguiente pulso de reloj, tal número se almacena en el acumulador.

Nótese las dos conexiones etiquetadas α_{i1} y β_{i1}. Supóngase que el flip-flop JK fuera reemplazado por el flip-flop con las compuertas OR de la Figura 5.4. En tal caso, la salida de la compuerta AND **a** en la Figura 5.5 se conectarían a la compuerta OR α por su entrada α_{i1}. De manera similar, la salida de la compuerta AND **b** se conectaría a la compuerta OR β, entrada β_{i1}.

Ahora bien, supóngase que todas las demás entradas α_i y β_i en la Figura 5.4 tienen valor 0. El circuito funcionaría exactamente como el circuito de la Figura 5.5, ya que si α_i y β_i tienen valor 0, entonces:

$$J_i = s_i$$
$$K_i = s_i'$$

tal y como originalmente se deseaba. En conclusión, la inclusión de las compuertas OR no cambia la operación del circuito ya que todas las demás entradas α_i y β_i tienen valor 0. De hecho, la presencia de las compuertas OR permite obtener las diferentes funciones de la ALU que se requieren. Nótese que cuando ADD = 0 e INC = 0, entonces la salida es $\alpha_i = \beta_i = 0$. Ahora bien, en cuanto a la operación de las compuertas OR α y β concierne, las entradas α_{i1} y β_{i1} podrían quitarse o dejar de considerarse, mientras que las otras entradas a las compuertas pueden utilizarse para obtener otras diferentes funciones.

5.6.2. Limpieza (CLEAR)

Si se desea "limpiar" el registro acumulador, es decir, poner un valor 0 en todos y cada uno de sus bits, entonces se utiliza el circuito que se muestra en la Figura 5.6. En este caso, $J_i = 0$ (nótese que no hay entrada a J). Cuando CLR = 1, entonces $K_i = 1$, y por lo tanto, el flip-flop se coloca en estado $Q_i = 0$ al siguiente pulso de reloj. Cuando se reemplaza el flip-flop por el circuito de la Figura 5.4, la entrada CLR se aplica en la entrada β_{i2}. Todas

las demás entradas α_i y β_i deben tener valor 0. Por lo tanto, con CLR = 1, se tiene que $J_i = 0$ y $K_i = 1$, y la limpieza deseada ocurre (nótese que la entrada α_{i2} es innecesaria, ya que su valor es 0).

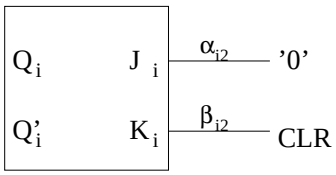

Figura 5.6: Etapa de la ALU que realiza la limpieza del acumulador.

5.6.3. AND lógica

Hay ocasiones cuando se desea realizar operaciones lógicas utilizando como entradas los valores del registro B y el acumulador (b_i y a_i). Considérese la operación AND. El i-ésimo bit del número binario B se compara con la salida Q_i, que es el i-ésimo bit almacenado en el acumulador. Si b_i y Q_i tienen ambos valor 1, entonces el valor de Q_i permanece siendo 1. Si cualquiera de los dos, o ambos b_i ó Q_i, tienen valor 0, entonces Q_i toma valor 0. El circuito que realiza tal operación se muestra en la Figura 5.7.

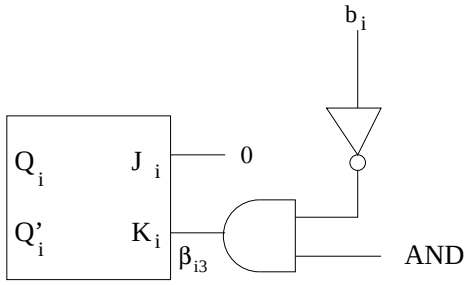

Figura 5.7: Etapa de la ALU que realiza la AND lógica.

Nótese que si AND = 1 y $b_i = 1$, entonces $\beta_{i3} = 0$, y por lo tanto, el estado del flip-flop no cambia. Si $Q_i = 1$, éste permanece con ese valor. De manera similar, si $Q_i = 0$, entonces el estado permance igual. Por otro lado, si $b_i = 0$, entonces cuando AND = 1, se tiene que $K_i = 1$ y $J_i = 0$, por lo que al siguiente pulso de reloj, $Q_i = 0$. Si AND = 0, entonces $\beta_{i3} = 0$, lo que hace que no haya cambio en el estado del flip-flop.

Si se reemplaza el flip-flop de la Figura 5.4 con β_{i3} conectado a la entrada correspondiente de la compuerta OR β, el circuito funciona tal y como se ha descrito. Nótese que si AND = 0, entonces $\beta_{i3} = 0$, por lo que esta entrada no interfiere con las otras entradas.

5.6.4. OR lógica

La operación OR lógica se realiza de la siguiente manera: el bit b_i de B se compara con el bit Q_i almacenado en el acumulador. Si cualquiera de los dos o ambos tienen valor de 1, entonces se coloca un valor de 1 en el i-ésimo flip-flop del acumulador en el siguiente pulso de reloj. Si ambos, Q_i y b_i tienen valor 0, entonces se coloca un 0 en Q_i. Por lo tanto, Q_i permanece sin cambio a menos que $b_i = 1$ y $Q_i = 0$. Solo entonces Q_i toma valor de 1. Un circuito que realiza esta operación se muestra en la Figura 5.8.

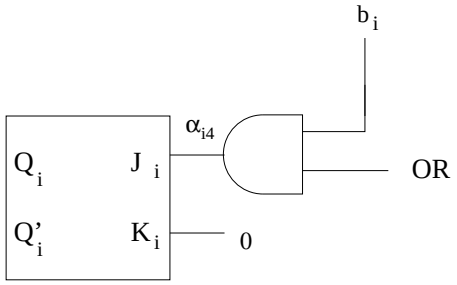

Figura 5.8: Etapa de la ALU que realiza la OR lógica.

Cuando OR = 1 y $Q_i = 1$ entonces Q_i permanece con valor 1, independientemente del valor de b_i. Si OR = 1 y $Q_i = 0$ con $b_i = 1$ entonces al siguiente pulso de reloj, Q_i toma el valor de 1. Así, el circuito opera correctamente la OR lógica. Nótese que si OR = 0, entonces $\alpha_{i4} = 0$. De este modo, se puede reemplazar el flip-flop por el flip-flop de la Figura 5.4.

5.6.5. XOR lógica

La operación XOR lógica se realiza mediante comparar el bit b_i de B con el bit Q_i del acumulador. Si cualquiera de ellos, Q_i ó b_i tienen valor 1, entonces, después del siguiente pulso de reloj, se almacena un valor de 1 en el flip-flop. Si ambos b_i y Q_i tienen valor 0, o si ambos b_i y Q_i tienen valor 1, al siguiente pulso de reloj, Q_i toma valor 0. Un circuito que implementa este comportamiento se muestra en la Figura 5.9.

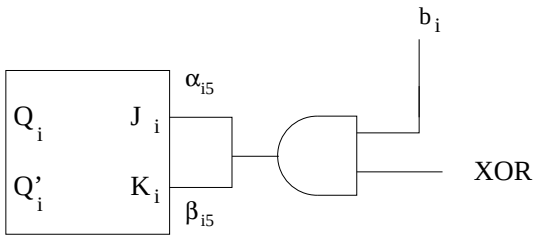

Figura 5.9: Etapa de la ALU que realiza la XOR lógica.

Si XOR = 1 y $b_i = 1$, entonces, después del pulso de reloj, el estado del flip-flop cambia. Así, si $Q_i = 0$, resulta en que $Q_i = 1$; de manera similar, si $Q_i = 1$, el resultado es $Q_i = 0$. Esto va de acuerdo con la operación XOR. También, si XOR = 1 y $b_i = 0$, si $Q_i = 0$, entonces Q_i permanece en ese estado; si $Q_i = 1$, entonces también permanece en tal estado. Finalmente, si XOR = 0, entonces $\alpha_{i5} = 0$ y $\beta_{i5} = 0$, por lo que el flip-flop puede substituirse por el flip-flop en la Figura 5.4.

5.6.6. Corrimiento a la derecha

Comenzando con las operaciones sobre los contenidos del acumulador, se inicia discutiendo las operaciones de corrimiento. Estos circuitos son similares en su operación al circuito de la Figura 3.30. Para correr los contenidos del acumulador a la derecha, la entrada de cada flip-flop debe provenir de la salida del flip-flop inmediatamente a su izquierda. El circuito utilizado se muestra en la Figura 5.10.

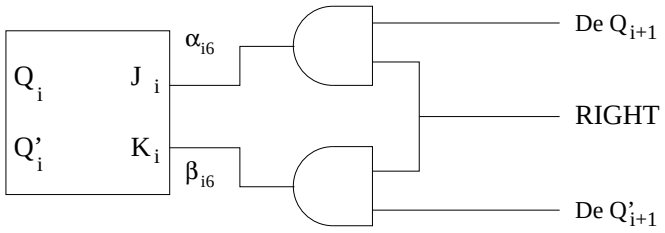

Figura 5.10: Etapa de la ALU que realiza el corrimiento a la derecha.

Nótese que el flip-flop más a la izquierda no tiene entrada. Para darle una entrada se reemplazan Q_{i+1} y Q'_{i+1} de la Figura 5.10 por entradas externas. Así, el acumulador funciona como un registro de corrimiento.

Ahora bien, cuando RIGHT = 1, entonces el corrimiento ocurre. Si RIGHT = 0, entonces $\alpha_{i6} = \beta_{i6} = 0$, y no hay corrimiento. De nuevo, es posible reemplazar el flip-flop por el circuito de la Figura 5.4.

5.6.7. Corrimiento a la izquierda

El circuito que corre los contenidos del acumulador a la izquierda es esencialmente el mismo que el circuito para corrimiento a la derecha, excepto que ahora las entradas provienen del flip-flop inmediatamente a la derecha. La entrada del flip-flop más a la derecha consiste de una entrada externa y su complemento. Un circuito que reliza el corrimiento a la izquierda se muestra en la Figura 5.11. De nuevo, nótese que cuando LEFT = 0, entonces $\alpha_{i7} = \beta_{i7} = 0$, por lo que se puede reemplazar el flip-flop por el circuito de la Figura 5.4.

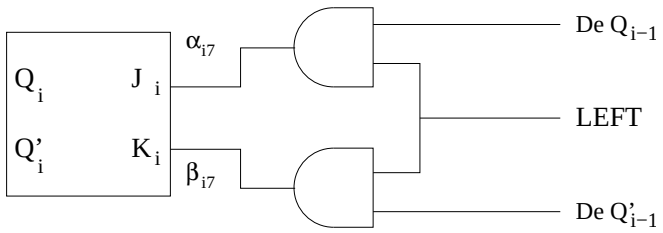

Figura 5.11: Etapa de la ALU que realiza el corrimiento a la izquierda.

5.6.8. Complemento

El circuito que se muestra en la Figura 5.12 muestra cómo se puede complementar todos los bits almacenados en el acumulador. Este es últil para obtener el complemento a 2. El flip-flop se conecta como un flip-flop T. Cuando COMP = 1, cambia el estado a su valor complemento. De nuevo, nótese que cuando COMP = 0, se tiene que $\alpha_{i8} = \beta_{i8} = 0$, por lo que también se puede reemplazar el flip-flop por el circuito de la Figura 5.4.

5.6.9. Conexión de entradas

Ya que se han considerado un número de operaciones, recuérdese que cuando los circuitos de las Figura 5.5 a 5.12 se conectan juntos, solo una señal de control puede tomar el valor de 1 en cualquier momento, mientras que el resto debe permanecer con valor 0. De hecho, por ahora es notorio que en

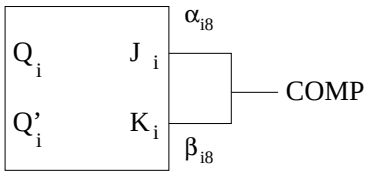

Figura 5.12: Etapa de la ALU que realiza complemento.

lugar de utilizar un solo flip-flop, se utiliza la conexión de la Figura 5.4, donde hay dos compuertas OR conectadas a las entradas J y K. Por lo tanto, cada una de las Figuras 5.5 a 5.12 es una parte que se han dividido en los puntos de conexión marcados como α_{ik} y β_{ik}, que se conectan a las entradas de las compuertas OR como se indica en la Figura 5.4. Cada operación funciona como se ha descrito en las secciones anteriores. Por ejemplo, supóngase que ADD = 1, y todas las demás señales de control tienen valor 0. Por lo tanto:

$$\alpha_{i4} = \alpha_{i5} = \alpha_{i6} = \alpha_{i7} = \alpha_{i8} = \beta_{i3} = \beta_{i5} = \beta_{i6} = \beta_{i7} = \beta_{i8} = 0$$

Y por lo tanto:

$$J_i = \alpha_{i1}$$
$$K_i = \beta_{i1}$$

Esto es exactamente la condición para el funcionamiento del circuito de la Figura 5.5, y significa que una operación de suma se realiza. Una consideración similar puede tenerse con todas y cada una de las otras operaciones y señales de control.

5.6.10. Inspección de salidas

En esta sección se discuten dos operaciones de la ALU que proveen de información sobre el valor almacenado en el acumulador, pero no cambian los datos almacenados de ninguna forma: inspección negativa y de cero.

Inspección negativa

Para esta inspección, se desea una señal de salida que indique un valor de 1 si el número almacenado en el acumulador es negativo, y 0 de otra forma. Se

supone que se utiliza una representación de complemento a 2. Así, el número es negativo si el bit más significativo tiene valor de 1, y será positivo si tiene valor 0. Por lo tanto, sólo se requiere examinar el estado del flip-flop más a la izquierda del acumulador para determinar si el número es negativo.

Un circuito que cumple con esta función se muestra en la Figura 5.13. La señal de salida se marca como NEG, de modo que si NEG = 1, entonces γ_1 tiene el valor de Q_N. Ya que Q_N es el bit almacenado más a la izquierda, γ_1 tiene valor de 1 cuando el número sea negativo, y tiene valor de 0 cuando el número sea positivo o cero.

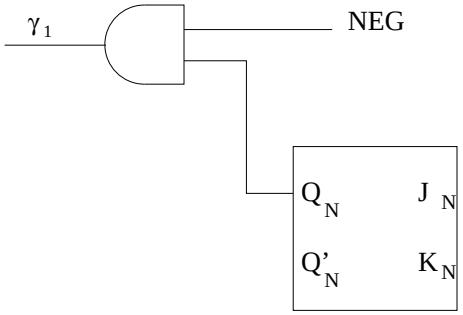

Figura 5.13: Circuito de inspección negativa.

Inspección de cero

También, frecuentemente es necesario verificar si el valor almacenado en el acumulador es 0. Si tal es el caso, entonces se desea saber si:

$$Q_0 = Q_1 = Q_2 = \cdots = Q_N = 0$$

En este caso, si:

$$Q'_0 = Q'_1 = Q'_2 = \cdots = Q'_N = 1$$

El circuito de la Figura 5.14 realiza la inspección de cero. Cuando la salida ZERO tiene valor de 1, entonces γ_2 tiene valor de 1 si y solo si se satisface la expresión anterior. Si $\gamma_2 = 0$ cuando ZERO = 1, entonces el valor almacenado en el acumulador no es 0. Nótese que cuando ZERO = 0, la salida γ_2 tiene valor 0.

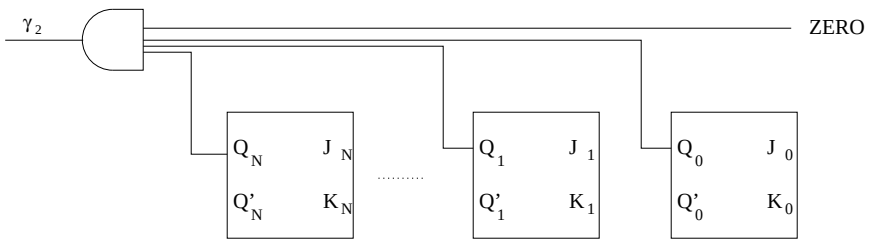

Figura 5.14: Circuito de inspección de cero.

5.6.11. Conexión de salidas

En las dos secciones anteriores se han tratado dos salidas correspondientes a las inspecciones negativa y de cero. Comúnmente, las ALU tienen una sola salida para ambas operaciones. Si ZERO = 1, entonces la salida da el resultado de la inspección de cero. Similarmente, si NEG = 1, entonces se da la salida de inspección negativa. Tal única salida puede obtenerse de las señales γ_1 y γ_2, como se muestra en la Figura 5.15. Recuérdese que las dos operaciones de inspección no pueden efectuarse al mismo tiempo.

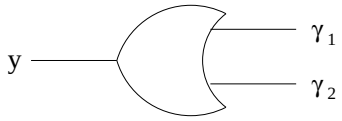

Figura 5.15: Circuito que provee una sola salida para las inspecciones negativa y de cero.

Así, se termina de discutir una ALU completa que realiza muchas operaciones. Sin embargo, no se han discutido todas las posibles operaciones que una ALU puede realizar, sino más bien, se han expuesto las más comunes. Más importante aún, es que la comprensión de esta ALU debe permitir el comprender otras ALU, aún cuando éstas realicen otras operaciones. En el siguiente capítulo se discute cómo la ALU se inserta dentro de una computadora digital completa.

Capítulo 6

La computadora digital

El objetivo de este capítulo es hacer una descripción tan completa como sea posible de la operación de una computadora digital, tanto desde el punto de vista del *hardware* como del *software*. Comenzando con la organización general de una computadora, se discute la forma en que las instrucciones se almacenan, la naturaleza y desarrollo de un lenguaje de máquina, y la tarea práctica de escribir un programa en lenguaje de máquina. También se describen, en general, el lenguaje ensamblador y los lenguajes de alto nivel.

6.1. Organización de una computadora digital

El diagrama de bloques de la Figura 6.1 puede utilizarse para describir la organización general de una computadora digital. Cuenta con cuatro unidades básicas. Todas las operaciones se realizan en la unidad lógica-aritmética (ALU). En una computadora pequeña, esta unidad consiste del tipo de elementos descritos en la Sección 5.6, posiblemente con registros adicionales; la ALU de muchas computadoras (hasta en computadoras pequeñas) frecuentemente contiene varios registros acumuladores. Anteriormente ya se han discutido todos los tipos de circuitos que se utilizan normalmente en una ALU típica.

En la Sección 5.6 se muestra que la ALU realiza las operaciones mediante seleccionar de un conjunto de señales de control, tales como ADD, CLEAR y OR. Estas señales se generan por la unidad de control, que recibe sus instrucciones de un programa. Algunas unidades de control son más complejas que otras. Por ejemplo, en la Sección 5.3 se discute la substracción mediante complementar el sustraendo, sumarle 1 y sumar el resultado al minuendo.

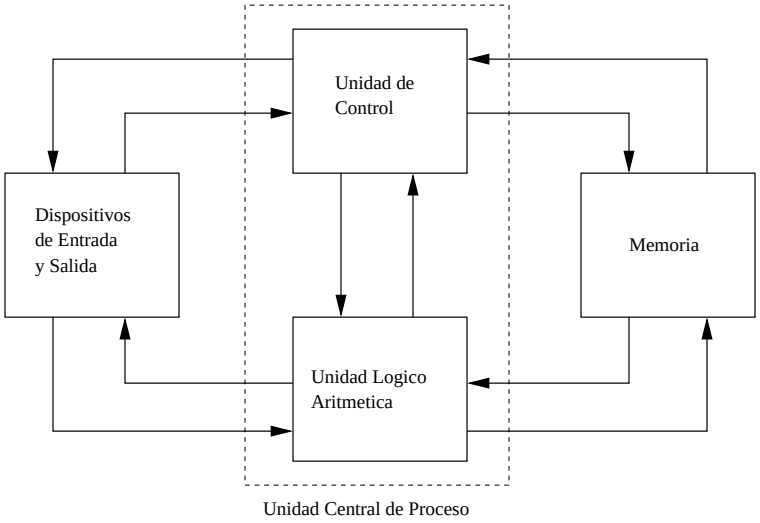

Figura 6.1: Diagrama de bloques de una computadora digital básica.

De este modo, tres instrucciones deben proveerse a la ALU: complemento, incremento y suma. Si la unidad de control fuera una muy simple, las tres instrucciones tendrían que proveerse por el programa mismo. Con una unidad de control más compleja, muy probablemente solo requiera una instrucción de substracción. La propia unidad de control se encargaría entonces de generar las tres señales de control apropiadas, en la secuencia adecuada, y en los tiempos precisos.

Además de controlar el flujo de datos desde y hacia los dispositivos de entrada y salida, la unidad de control también dirige las acciones de lectura y escritura de la memoria. La unidad de control también contiene el reloj maestro que se encarga de sincronizar la operación de todas las partes de la computadora. Nótese la línea punteada de la Figura 6.1, que conjunta la unidad de control y la ALU. En muchas computadoras, ambas unidades se encuentran unidas en un solo circuito integrado. En tal caso, este circuito se conoce como *microprocesador*. De cualquier modo, la ALU y la unidad de control juntas se consideran componentes de la *unidad central de proceso* (*central processing unit*, o simplemente, CPU).

A su vez, la memoria puede considerarse dividida en dos partes: la *unidad de memoria principal* (*main memory unit* o MMU) y la memoria auxiliar. La MMU es una memoria de acceso aleatorio como las que se discuten en

el Capítulo 4. En las computadoras pequeñas, esta unidad consiste de una memoria semiconductora. La memoria auxiliar, por otro lado, consiste de discos y cintas, como las descritas en el mismo Capítulo 4.

Para poder utilizar una computadora, es necesario proveerle de programas y datos. Después de que el procesamiento se ha llevado a cabo, la computadora a su vez debe ser capaz de proveer los resultados. La comunicación con la computadora es función de los dispositivos de entrada y salida. El dispositivo de entrada más común es el teclado, en el cual un programador puede escribir, letra por letra, su programa. El teclado genera las secuencias de unos y ceros que la computadora interpreta. Otra forma de entrada puede ser mediante discos y cintas, en los que las secuencias de unos y ceros ya se encuentran almacenados. Nótese que las cintas y los discos pueden considerarse tanto como dispositivos de entrada y salida, así como memoria auxiliar. Si su función es proveer datos o programas, o si los resultados del procesamiento finalmente se almacenan en ellos, entonces se les considera dispositivos de entrada/salida; si almacenan valores como resultados intermedios, entonces puede considerárseles como parte de la memoria.

Existen otros tipos especiales de dispositivos de entrada. Por ejemplo, las lectoras de los supermercados pueden leer los códigos de barras que representan los precios de los artículos. Estos lectores proveen de información a una computadora, la cual al final obtiene el total de la compra, pero puede además realizar actividades más complejas, como utilizar la información de los artículos comprados para manejar el control de inventarios.

El dispositivo de salida más común es la pantalla. Esta no es otra cosa mas que un *tubo de rayos catódicos* (*cathodic ray tube* ó simplemente CRT) parecida a un receptor de televisión. Otro dispositivo de salida común es la impresora, que permite obtener la salida en hojas de papel.

En muchos casos, la entrada y salida de una computadora se manejan de forma enteramente diferente a lo que se describe aquí. Por ejemplo, los datos de entrada pueden provenir de sensores de temperatura, luz o humedad colocados en lugares específicos de un aparato o un edificio. La salida puede consistir entonces en señales que controlen actuadores, los cuales echan a andar calefactores o humidificadores en el propio aparato o edificio.

6.1.1. *Buses*

Aun cuando sólo se muestran dos líneas conectando los varios componentes de una computadora digital en la Figura 6.1, en realidad tales conexiones se realizan mediante un número de pequeños alambres que van de las terminales de un componente a las terminales de otro. Esta colección de alambres se conoce con el nombre de *bus*. Supóngase que la computadora usa palabras de 16 bits. Es necesario entonces que exista un alambre en el bus por cada bit de la palabra. En la Sección 4.1 se explica que debe haber una línea de entrada por cada dirección binaria utilizada por la memoria. Por ejemplo, en la Figura 4.3 se tienen dos terminales de entrada para las direcciones de una memoria de cuatro palabras. Además, debe haber una terminal de selección del circuito, y finalmente, terminales que conduzcan las señales de lectura y escritura en memoria. Todas estas líneas constituyen conexiones adicionales en el bus.

En la Sección 5.6 se discuten el uso por separado de líneas de control para cada operación de la ALU. Cada una de estas líneas de control deben también incluirse en el bus, además de otras líneas de control, como por ejemplo, conexiones para la señal de reloj. Ya que los circuitos de la computadora utilizan corriente directa, el bus debe contener también líneas de alimentación eléctrica, normalmente de 5 volts (marcado como V_{cc}) y 0 volts (llamado "tierra" o GND). Finalmente, para permitir la expansión de la computadora, como por ejemplo incrementar la capacidad de memoria, el bus debe contener líneas extra.

Parecería que no se requiere que todas las líneas de conexión pasen por todos los componentes de la computadora. Sin embargo, para evitar problemas de conexión entre ellos, se hace que todo componente tenga toda conexión disponible, aun cuando no requiera utilizarla. De hecho, normalmente en la actualidad el bus de casi todas las computadoras, pero especialmente en las computadoras pequeñas, consiste en un conjunto de alambres de metal impresos en una tarjeta plástica llamada *tarjeta madre* (*motherboard*). Esta tarjeta provee de acceso a todas las terminales de la computadora, ya que se pueden colocar conectores a ella. Por ejemplo, nuevos componentes pueden añadirse mediante conectarlos al bus.

6.2. Instrucciones de memoria – transmisión de información

En la Sección 5.6 se discuten varias operaciones que la ALU es capaz de realizar sobre valores binarios. En la Sección 6.5 se describe estas operaciones como instrucciones en lenguaje de máquina que, tomándolas en la forma secuencial de un programa en lenguaje de máquina, dirigen su operación. Por lo pronto, en esta sección se discuten los detalles de cómo las instrucciones se almacenan en la computadora, en particular las instrucciones específicas que se utilizan para controlar a la memoria.

Primeramente, se considera cómo se direcciona la memoria. En la Sección 4.1 se menciona que la dirección de la palabra a leer o escribir de la memoria se presenta como un número binario. Hay una terminal por cada dígito binario de la dirección. Para direccionar la memoria, las señales apropiadas deben colocarse en las líneas de dirección del bus. Estas señales se proveen de un registro especialmente diseñado para almacenar, en binario, las direcciones de memoria. A este registro se le conoce como *registro de dirección de memoria* (*memory address register* o MAR). El MAR, entonces, normalmente contiene el valor binario de la dirección de la palabra que se lee o escribe de la memoria. En la Figura 4.3 se muestran un par de flip-flops que, considerados en arreglo, forman un registro que puede ser utilizado como MAR.

Otro registro asociado con la memoria principal es el *registro separador de memoria* (*memory buffer register* o MBR). Este registro tiene la función de almacenar la palabra que se lee de ó se escribe a la memoria.

Considerando ambos registros MAR y MBR, se requieren apenas unas cuantas instrucciones para manejar la memoria:

1. Escribe la palabra almacenada en el MBR a la memoria principal, en la dirección almacenada en el MAR.

2. Lee la palabra almacenada en la dirección almacenada en el MAR, y colócala en el MBR.

3. Limpiar el MAR o copiar una palabra al MAR.

A veces, en computadoras sencillas, estas instrucciones se combinan con otras. Por ejemplo, se discute más adelante una instrucción para almacenar los contenidos del acumulador en una localidad de memoria que se especifica

en la instrucción misma. En realidad, tal instrucción sola puede tomar el lugar de varias otras instrucciones, una para almacenar los contenidos del acumulador en el MBR, y entonces otra para almacenar los contenidos del MBR en una localidad expecífica de memoria. Sin embargo, esta secuencia de instrucciones podría ser automáticamente generada por la unidad de control. De hecho, algunas computadoras antiguas no cuentan con un MBR, por lo que utilizan otros registros, como el acumulador.

6.2.1. Estructura de la palabra

Al considerar la estructura de las palabras almacenadas en la memoria, recuérdese que hay dos tipos de información binaria almacenada en la memoria: instrucciones y datos. Los datos pueden ser un valor numérico, que puede ser de punto flotante o un número sin exponente; o puede ser un código para información alfanumérica.

Cada palabra almacenada en la memoria que representa una instrucción normalmente contiene dos partes: un código de instrucción y una dirección de memoria; sin embargo, en algunas computadoras, se utilizan palabras más complejas, conteniendo más información. La instrucción propiamente dicha es el código de la instrucción. De hecho, cada instrucción tiene un código binario que la representa. Por ejemplo, 000001 podría indicar que los contenidos de una dirección de memoria dada deben sumarse con el contenido del acumulador.

Para ilustrar esto, se trabaja con una palabra de 16 bits, como la que se muestra en la Figura 6.2

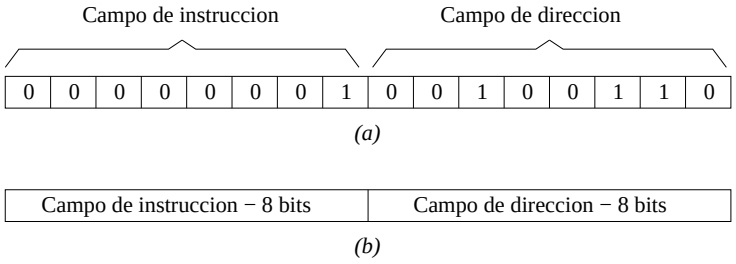

Figura 6.2: *(a)* Una palabra de 16 bits con un campo de instrucción de 8 bits y un campo de dirección de 8 bits; *(b)* una representación más simple de esta palabra.

La palabra se divide en dos sub-palabras, llamadas *campos* (*fields*). Un campo se utiliza para el *código de instrucción* (*operational code* u *op-code*), y el otro para la dirección de memoria. Considérese la palabra específica que se muestra en la Figura 6.2:

$$00000001\,00100110_2$$

Los primeros 8 bits representan el código de la instrucción, y los 8 bits finales representan una dirección de memoria. Por ejemplo, los primeros 8 bits (00000001_2) pueden indicar a la computadora para sumar los contenidos de la dirección de memoria con los contenidos del acumulador, mientras que los últimos 8 bits (00100110_2) especifican una dirección de memoria. Por lo tanto, la palabra $00000001\,00100110_2$ significa sumar el número binario contenido en la dirección 00100110_2 con el número binario almacenado previamente en el acumulador.

La palabra $00000001\,00100110_2$ es un ejemplo de un *enunciado* (*statement*) de un programa en lenguaje de máquina, un programa escrito en binario que dirige la operación de la computadora. Para eliminar posibles confusiones o errores al utilizar números binarios, se acostumbra escribir el programa en su equivalente hexadecimal, como por ejemplo:

	Campo de instrucción	Campo de dirección
Binario	00000001	00100110
Hexadecimal	01	26

La representación hexadecimal es más fácil de manejar, y muchos programas en lenguaje de máquina se escriben en hexadecimal. Sin embargo, el programa real se encuentra, por supuesto, en binario.

Hasta aquí, se discute una palabra que contiene un campo de instrucción y un campo de dirección. La estructura de esta palabra se utiliza para dirigir a la computadora para realizar una operación, posiblemente en conjunción con una dirección de memoria. En el ejemplo anterior, la instrucción causa que el contenido de la dirección de memoria dada se sume con el contenido del acumulador. Otra instrucción puede utilizarse, igualmente, para almacenar el contenido del acumulador en una dirección de memoria específica.

En los ejemplos subsecuentes se utiliza esta estructura de palabra dado que es simple. Esto elimina una gran cantidad de detalles innecesarios

en la discusión. Sin embargo, pueden utilizarse otros tipos de estructuras de palabra. Por ejemplo, algunas computadoras utilizan palabras con dos campos de dirección. Suponiendo que se desea transferir el contenido de una dirección de memoria a otra dirección de memoria, una palabra con dos campos de dirección podría ser muy conveniente para expresar esto en una sola instrucción.

Además, existen otras formas de instrucciones para las computadoras. Muchas computadoras utilizan 8, 16, 32 o hasta 64 bits. Si 8 o más bits se utilizan para especificar una dirección en memoria, entonces una palabra de 8 bits no puede proveer espacio para un campo de instrucción (*op-code*) y un campo de dirección. De tal modo, muchas computadoras utilizan secuencias de palabras para desarrollar instrucciones. Por ejemplo, supóngase que hay dos palabras en la secuencia. La primer palabra puede proveer del código de operación, mientras que la segunda puede especificar la localidad de memoria. De hecho, muchas computadoras utilizan 16 bits para especificar una dirección de memoria, con lo que se tiene que pueden direccionarse hasta 65,536 localidades de memoria. En este caso, una instrucción se compone de tres palabras de 8 bits: una para el código de operación, y dos palabras para la dirección de memoria.

Al programar una computadora, la estructura de sus instrucciones debe ser conocida previamente. Normalmente, los manuales de instrucción del procesador de la computadora contienen muchos de estos detalles.

6.3. Ejecución de instrucciones

Para ilustrar la ejecución de instrucciones, a continuación se escribe un programa simple en lenguaje de máquina, especificando algunas instrucciones adicionales. Para fines didácticos, se utilizan palabras de 16 bits. Las instrucciones son:

- sumar los contenidos de una dirección en memoria con el acumulador $(0000\,0001_2 = 01_{16})$
- limpiar el acumulador $(0000\,0010_2 = 02_{16})$
- transferir el contenido del acumulador a una dirección dad de memoria $(0001\,0100_2 = 14_{16})$
- detener el cómputo $(0111\,0111_2 = 77_{16})$

Con estas instrucciones, se escribe un programa sumamente simple. Se supone que la computadora ejecuta las instrucciones en orden secuencial, es decir, ejecuta la instrucción en la primera dirección de memoria, luego la instrucción en la segunda dirección, etc. Para este programa, se listan tanto las instrucciones como la dirección de memoria donde se encuentran almacenadas. Se supone que ya se encuentran en la memoria de trabajo de la computadora. Se presentan tanto la notación binaria como la notación hexadecimal.

Dirección		Palabra	
Binario	Hexadecimal	Binario	Hexadecimal
00000001	01	0000001000000000	0200
00000010	02	0000000100010000	0110
00000011	03	0000000100010001	0111
00000100	04	0001010000010010	1412
00000101	05	0111011100000000	7700
00010000	10	0000000000000011	0003
00010001	11	0000000000000100	0004

Al operar este programa, recuérdese que los primeros 8 bits de la instrucción especifican el código de operación de la instrucción, y que los últimos 8 bits especifican la dirección en memoria. La primera instrucción es:

$$0200_{16}$$

Esta es la instrucción para limpiar el acumulador. El valor en el campo para la dirección de memoria no tiene significado aquí, y los dos últimos dígitos hexadecimales se ignoran por parte de la computadora. De hecho, podría utilizarse cualquier valor en esas posiciones.

La siguiente instrucción es:

$$0110_{16}$$

Los primeros 8 bits (01_{16}) indican que el número almacenado en la dirección de memoria 10_{16} debe sumarse con el contenido del acumulador. El número almacenado en la dirección 10_{16} es:

$$0000000000000011_2 = 0003_{16}$$

Por lo tanto, después de que esta instrucción se ejecuta, el valor 0003_{16} se coloca en el acumulador (almacenado, por supuesto, en binario).

La siguiente instrucción es:

$$0111_{16}$$

Esta instrucción especifica que el contenido de la dirección de memoria 11_{16} deben sumarse con el contenido del acumulador. El número almacenado en la dirección 11_{16} es:

$$0000000000000100_2 = 0004_{16}$$

Al finalizar la ejecución de esta instrucción, el número almacenado en el acumulador es el resultado de la suma del número previamente almacenado (0003_{16}) más 0004_{16}. La operación es como sigue:

$$0003_{16} + 0004_{16} = 0007_{16}$$

La siguiente instrucción es:

$$1412_{16}$$

Los primeros 8 bits (14_{16}) indican que el contenido del acumulador debe almacenarse en la dirección de memoria 12_{16}. Tras la ejecución de esta instrucción, la palabra almacenada en la dirección de memoria 12_{16} es:

$$0000000000000111_2 = 0007_{16}$$

Finalmente, se ejecuta la instrucción:

$$7700_{16}$$

Los primeros 8 bits (77_{16}) causan que el cómputo se detenga. El resultado de este simple programa es, entonces, sumar dos valores numéricos en diferentes localidades de memoria y escribir el resultado en una tercera localidad de la memoria. En este simple programa no se toma en cuenta ninguna salida de datos. Esto se considera más adelante.

Por lo pronto, se consideran algunas operaciones de la computadora en un mayor detalle. El código de operación informa a la computadora para que realice alguna operación específica. Cuando la instrucción se ejecuta, a la ALU se le provee de las señales de control apropiadas. Esto se ejemplifica utilizando una computadora sencilla. Supóngase que los 8 bit más a la izquierda de la instrucción (el código de operación) se almacena en un registro, conocido como *registro de instrucción* (*Instruction Register* o simplemente

IR). El resto de los bits de la instrucción se consideran almacenados en el MAR. Ahora bien, tomando solo el contenido de IR, éste se da como entrada a un *decodificador de instrucciones* (*instruction decoder*). Este dispositivo convierte la información almacenada en IR al conjunto apropiado de señales de control. Un decodificador de instrucciones muy simplificado se muestra en la Figura 6.3. Este circuito puede decodificar un código de operación de la forma:

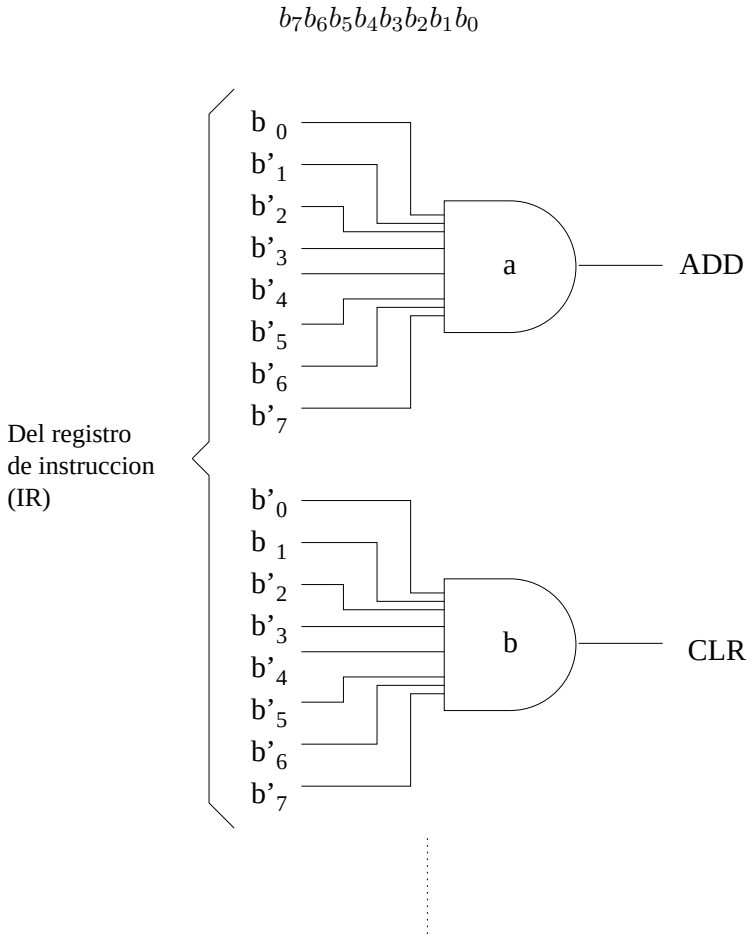

Figura 6.3: Parte de un decodificador de instrucciones.

Por ejemplo, si el código de operación es 00000001_2, entonces la salida de la compuerta AND **a** toma el valor de 1, lo que hace que la señal de control

ADD tome valor 1 y el resto de las señales de control tengan un valor de 0. Similarmente, si el número almacenado en IR es 00000010_2, entonces sólo la salida de la compuerta AND **b** tiene valor 1, y la señal de control CLR se produce.

Cuando se ejecuta un programa, el número almacenado en cada localidad de memoria es una secuencia de ceros y unos. Tal número no provee de ninguna información que indique si se trata de un código de operación o un dato. Esto se indica sólamente por el orden en que cada palabra se lee. Por ejemplo, supóngase que se lee 0110_{16} de una localidad de memoria, y que esto se trata como una instrucción. Tal instrucción indica que los contenidos de la localidad de memoria con la dirección 10_{16} se suman al número contenido en el acumulador. El contenido leído de la localidad con dirección 10_{16} es tratado, entonces, como un dato. De este modo, se considera que el contenido del primer byte representa un código de operación, mientras que el contenido de la dirección de memoria se considera un dato. En seguida se explica cómo la computadora puede ir siguiendo la secuencia de instrucciones.

La computadora cuenta con un registro de un solo bit (en realidad, un flip-flop) conocido como *registro de búsqueda-ejecución* (*fetch-execute register* o FER). Su función es indicar a la computadora si se encuentra buscando una instrucción de la memoria o ejecutando una instrucción (durante lo cual la información que se extrae de la memoria son datos). Si el valor contenido en FER es 1, entonces la computadora considera que la información que se extrae de la memoria es una instrucción. Si el contenido de FER es 0, entonces la información tomada de la memoria se trata como dato. Antes de describir con mayor detalle la operación del FER, es necesario tomar en cuenta otro registro, llamado *contador de programa* (*program counter* o PC), el cual contiene la dirección de la instrucción que se está ejecutando. Cada vez que una instrucción se ejecuta, se debe pasar a través de una secuencia de pasos. Además de los pasos propios para realizar la instrucción, hay pasos adicionales que llevan el orden secuencial de la ejecución, entre los cuales está incrementar el número (dirección) almacenado en el PC en una unidad.

Considérese ahora la ejecución de un programa. Cuando se inicia la operación, el contenido del FER tiene valor 1. Además, el contenido del PC toma el valor de la dirección del inicio del programa, por ejemplo 00000001_2 (se trabaja aquí con direcciones de memoria de 8 bits). Ya que el contenido del FER es 1, la información binaria traída de la memoria es tratada como una instrucción. Como el contenido del PC es 00000001_2, entonces se busca

el contenido de la localidad de memoria con la dirección 01_{16}. Los 8 bits más significativos se colocan en el IR, mientras que los 8 bits menos significativos se ponen en el MAR. En este momento, la computadora puede ejecutar la instrucción, de modo que el contenido del FER toma el valor 0. Por otro lado, el contenido de IR se proporciona al decodificador, y así, se generan las señales de control apropiadas para la realización de la instrucción. La dirección almacenada en el MAR se lee, y la instrucción descrita por el código de operación almacenado en IR se lleva a cabo. Así, se ejecuta la instrucción. Al terminar la ejecución, el contenido del FER toma el valor de 1 de nuevo, y el contenido de PC se incrementa, de modo que toma el valor $00000010_2 = 02_{16}$. Ahora, se busca la información binaria contenida en la localida de memoria con dirección 02_{16}, tratándola como instrucción. Y el proceso se repite una y otra vez por cada instrucción del programa. Nótese que la secuencia de pasos para la ejecución de instrucciones descrita aquí puede tener variaciones, como por ejemplo, el contenido del PC podría incrementarse mucho antes.

En la descripción del programa sencillo, el PC se ve afectado por una operación de incremento que aumenta su contenido binario en una unidad. Sin embargo, las computadoras digitales cuentan con otras operaciones que afectan el contenido del PC, no necesariamente para solo incrementarlo, sino para cambiar su contenido completamente. Esto significa que el programa no necesariamente se ejecuta como una secuencia rígida de pasos, sino que puede incluir ciclos o selecciones, lo que aumenta grandemente la versatilidad de la programación de la computadora.

Recuérdese que la computadora no distingue la diferencia entre instrucciones y datos. Por ejemplo, supóngase que la tercera instrucción del programa se cambiara por:

$$0101_{16}$$

En tal caso, el contenido de la localidad de memoria con dirección 01_{16}, que es el código de operación de la instrucción para limpiar el acumulador, se suma al contenido del acumulador. Esto no tiene sentido, pero la computadora no tiene manera de interpretarlo. La computadora funciona como se le instruye. Así, dado que el valor en la dirección 01_{16} es 0200_{16}, tras la ejecución de esta instrucción el contenido del acumulador es:

$$0200_{16} + 0003_{16} = 0203_{16}$$

Después de la ejecución, este valor se almacena en la dirección 12_{16}.

En este ejemplo, se lee una instrucción y se utiliza como dato. Puede haber serias consecuencias por comenter un error así. Supóngase que en lugar de leer una instrucción, accidentalmente se escribe un dato en una localidad de memoria de una instrucción que todavía no ha sido ejecutada. Cuando se llega a la dirección de memoria de tal instrucción, muy probablemente no se encuentra un código de operación válido. En tal caso, el decodificador de instrucciones no genera ninguna señal de control. Esto hace que se detenga el cómputo, indicando un error difícil de detectar. Más aún, si por casualidad se tiene un código de operación válido almacenado, el resultado del cómputo no será el esperado, por lo que la situación de error persiste de una manera más sutil.

6.4. La computadora digital completa

En esta sección, y tras las descripciones y consideraciones anteriores, se puede discutir por fin una computadora digital completa. Se describe una computadora digital de 16 bits a la que se ha hecho referencia anteriormente, notando que las teorías de operación pueden aplicarse a una computadora con cualquier tamaño de palabra. La Figura 6.4 muestra un diagrama de bloques de una computadora muy pequeña y sencilla. Aun cuando se muestran sólo líneas interconectando los varios componentes de la computadora, recuérdese que tales líneas representan buses, compuestos de varios alambres individuales. En seguida, se discuten los varios componentes de la computadora. Solo se mencionan brevemente aquéllos que han sido discutidos previamente con mayor detalle.

La unidad lógico-aritmética (ALU) se relaciona con el registro acumulador, y se constituye de varios circuitos lógicos para obtener las operaciones deseadas. Se muestra también un registro adicional: el registro B. Este puede ser utilizado para almacenar algún número que se opera conjuntamente con el contenido del acumulador. En general, la ALU puede accesar otros registros, lo que le permite realizar operaciones más complejas, como multiplicación y división. Nótese que la salida del acumulador puede ser redirigida a la unidad de control, y a partir de aquí, a varias otras partes de la computadora, como podría ser la unidad de memoria.

La unidad de memoria (MU) contiene a la memoria principal. Ya que se suponen 8 líneas para el direccionamiento de memoria, se utiliza una pequeña memoria de 256 palabras. Las memorias de computadora, por lo general, utilizan muchísima más memoria, por lo que ésta se hace disponible mediante

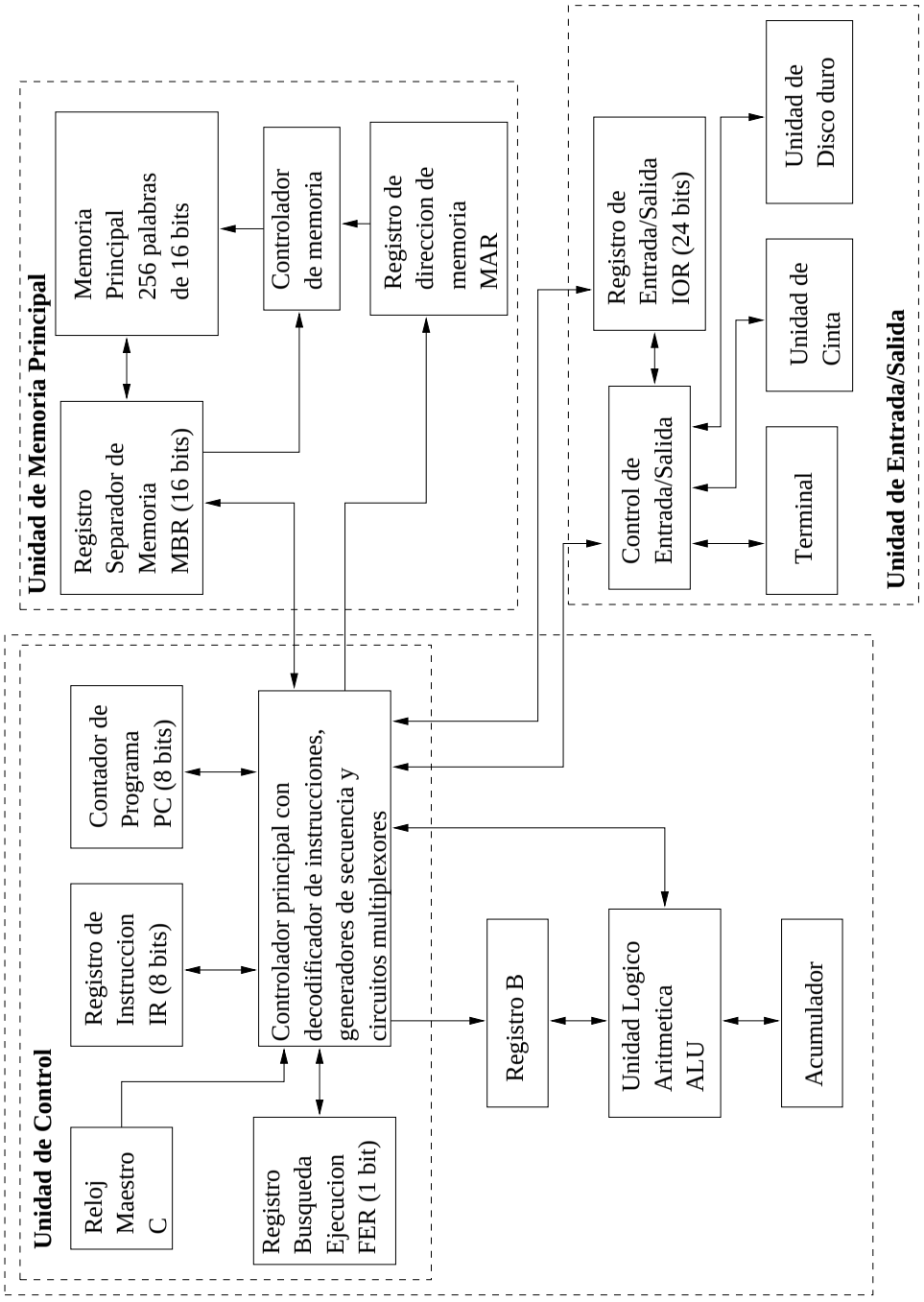

Figura 6.4: Diagrama de bloques de una pequeña computadora digital.

multiplicar las líneas de direccionamiento. Sin embargo, el tamaño de 256 palabras es sólo ilustrativo de este ejemplo de computadora sencilla. Por otro lado, el control de memoria se discute más a fondo en el Capítulo 4. Una dirección proveniente de la unidad de control se coloca en el MAR; si se da una instrucción de lectura de memoria, entonces el contenido en la localidad direccionada por el MAR se pasa al MBR; si se tiene una instrucción de escritura, entonces el contenido del MBR se copia a la localidad de memoria direccionada.

La unidad de control (CU) dirige la operación de toda la computadora. Contiene el decodificador de instrucciones (Figura 6.3), el cual "traduce" el código de operación relativo a la instrucción en señales de control que manipulan a la ALU, la memoria, los registros, y demás componentes de la computadora. También en la unidad de control se encuentran el IR, el FER y el PC, que se discuten en la sección anterior.

El reloj maestro es también parte de la unidad de control. Previamente, se ha descrito cómo el reloj maestro mantiene la sincronización entre las diferentes partes de la computadora. En realidad, dentro de una computadora existen más de una secuencia de pulsos de reloj, debido a que ciertas partes de la computadora tienden a funcionar a velocidades muchos mayores que otras partes. En general, la memoria es mucho más lenta que la unidad central de proceso. Normalmente, se realiza una instrucción de lectura de una sola palabra de la memoria, seguida muchas otras operaciones que no involucran a la memoria. Sería un desperdicio de tiempo si todas las operaciones se realizaran a la velocidad de un ciclo de la memoria, y por tal razón, el reloj maestro normalmente se utiliza para generar varios conjuntos de pulsos a diferentes frecuencias. Por ejemplo, la Figura 6.5 muestra dos conjuntos de pulsos, unos más rápios que otros. Sin embargo, la relación entre ellos se mantiene constante. Así, C_1 podría utilizarse como reloj de las partes más veloces de la computadora, mientras que C_2 podría utilizarse para la memoria. Esto presupone, por supuesto, que hay varias operaciones de las partes más veloces de la computadora por cada ciclo de memoria. La unidad de control, por tanto, debe contener los circuitos necesarios para asegurar que la memoria no sea llamada antes de que las otras operaciones hayan concluido.

La unidad de entrada/salida contiene todos los dispositivos de entrada/salida. En el diagrama de bloques de la Figura 6.4 sólo se muestran una terminal, una unidad de cinta, y una unidad de disco duro. En realidad, puede haber un gran número de estos tipos de dispositivos u otros dispositi-

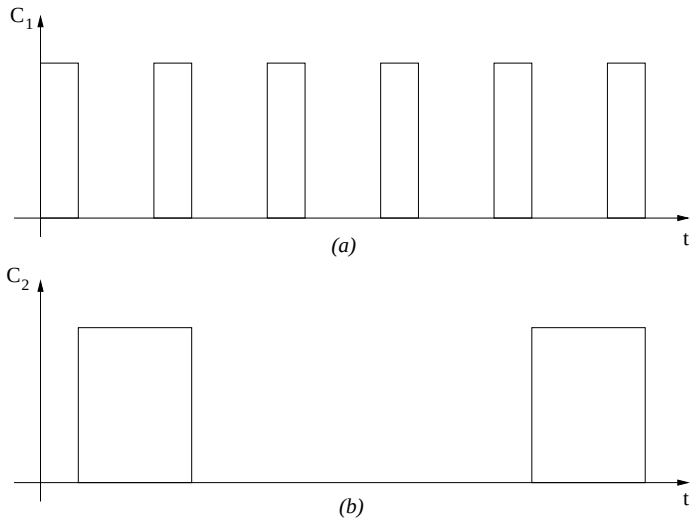

Figura 6.5: Pulsos de reloj *(a)* con temporización normal; *(b)* con temporización más lenta, para usarse con la memoria.

vos, como impresoras, graficadores, escaners, etc. Más aún, pueden utilizarse otros dispositivos de entrada/salida diseñados para comunicar información más especializada, como se menciona en la Sección 6.1 respecto a sensores para medir la temperatura, y a la vez, actuadores que la computadora controla.

Aun cuando la Figura 6.4 solo muestra un registro asociado con la unidad de entrada/salida, tal unidad puede contar en realidad con una mayor capacidad de almacenamiento. Los dispositivos de entrada/salida son normalmente mucho mas lentos que las demás partes de la computadora. Por ejemplo, la computadora puede producir datos binarios a una velocidad mucho mayor de que la pantalla puede presentarlos. Los datos, en este caso, se almacenan en una memoria del dispositivos de entrada/salida, conocida como *buffer*. De este modo, la pantalla toma los datos de su buffer, a su propio paso. De hecho, el buffer puede ser una sección de la RAM. Si no hubiera buffer, cada vez que se producieran datos de salida, la computadora tendría que hacer una pausa hasta que la pantalla terminara de presentar todos los datos. El uso del buffer de entrada/salida eliminia la necesidad de muchas de este tipo de pausas.

Cuando se coloca datos en el buffer, tales datos ocupan espacio de almacenamiento. Cuando los datos se leen del buffer y se despliegan por una pantalla, este espacio de almacenamiento se vacía, y el buffer puede recibir más datos. Si el buffer se llena, entonces no puede recibir más datos, y la computadora debe hacer una pausa. Estos y otros detalles se discuten más adelante, al presentar la programación de los dispositivos de entrada/salida.

6.5. Programación en lenguaje de máquina

En esta sección se presenta un pequeño lenguaje de máquina, y se muestra su utilización en el desarrollo de programas muy sencillos. Los lenguajes de máquina no representan un estándar, y el que se presenta aquí no es un lenguaje para una computadora en particular. Sin embargo, las ideas que se describen a continuación son aplicables a todas las computadoras. El lenguaje de máquina propuesto aquí se basa en mucho en las instrucciones que se dan a la ALU de la Sección 5.6. Además, se añaden otras instrucciones útiles que se incluyen en la mayoría de las computadoras. Sin embargo, inicialmente no se especifican instrucciones de entrada/salida. Estos se discuten más adelante.

Recuérdese que, en esta computadora sencilla, la palabra de una instrucción contiene 16 bits. Los 8 bits más significativos toman valores binarios de las instrucciones de la siguiente tabla. Los 8 bits menos significativos normalmente contienen direcciones de memoria. Para algunas instrucciones, la dirección de memoria no resulta un dato apropiado, y se le da un valor de 00000000_2. En realidad, podría dársele cualquier valor, ya que para efectos prácticos la computadora lo ignora.

Instrucción	Código Binario	Código Hexadecimal
Suma el contenido de la localidad de memoria dada con el acumulador	00000001	01
Limpiar el acumulador	00000010	02
AND Lógico	00000011	03
OR Lógico	00000100	04
XOR Lógico	00000101	05
Corrimiento a la derecha	00000110	06
Corrimiento a la izquierda	00000111	07
Complemento	00010000	10
Incremento	00010001	11
Verificación negativa	00010010	12
Verificación de cero	00010011	13
Colocar en la dirección de memoria dada	00010100	14
Salto incondicional	00100000	20
Salto si cero	00100001	21
Salto si negativo	00100010	22
Continua	00000000	00
Detener el cómputo	01110111	77

Las primeras once instrucciones se discuten en la Sección 5.6, en términos de operaciones de la ALU, y se detallan en las Secciones 6.2 y 6.3. Además, la instrucción para almacenar el contenido del acumulador en una dirección dada de memoria (14_{16}) y la instrucción para detener el cómputo (77_{16}) también se discuten en la Sección 6.3. En seguida, tras describir brevemente las nuevas instrucciones, se muestra la programación en lenguaje de máquina mediante algunos ejemplos de programas muestra.

A la penúltima instrucción, 00_{16}, se le llama *continua*. Esta instrucción *no hace nada*, es decir, al llegar a ella la computadora simplemente procede a la siguiente instrucción. Aun cuando esta instrucción no parece tener ninguna aplicación práctica, resulta muy útil, sobre todo cuando es necesa-

rio introducir algunos tiempos de espera en el programa. Otras instrucciones nuevas son todas aquéllas que involucran saltos. El *salto incondicional* causa que el contenido del PC cambie. Recuérdese que el PC contiene la dirección de la localidad de memoria en la cual la *siguiente* instrucción debe buscarse. Normalmente, el PC sólo se incrementa en una unidad. Sin embargo, ocasionalmente el programador puede requerir repetir algún segmento de instrucciones del programa, o evitarlo. El salto incondicional permite esto. Por ejemplo, considérese la instrucción:

$$0010000001000000_2 = 2040_{16}$$

Esta instrucción hace que el PC termine conteniendo el valor 40_{16}. De no haber otra instrucción de salto en esta dirección, el PC sigue incrementándose en una unidad, como es normal. Así, la siguiente instrucción a ser ejecutada es aquélla en la dirección 41_{16}.

Las otras dos instrucciones de salto se les conoce como *saltos condicionales*. Funcionan en forma similar al salto incondicional, excepto que la ocurrencia del salto depende del valor almacenado en el acumulador. Por ejemplo, supóngase que PC = 04_{16} y que se ejecuta la instrucción:

$$2142_{16}$$

Esta instrucción representa un salto si el valor almacenado en el acumulador es igual a cero. Si el contenido del acumulador es 00_{16}, entonces la siguiente dirección almacenada en el PC es 42_{16}. Así, la siguiente instrucción a ejecutar se toma de esta dirección. Por otro lado, si el valor almacenado en el acumulador *no es cero*, entonces el salto condicional es ignorado. Esto es, el PC toma el valor de 05_{16}, donde se busca la siguiente instrucción a ejecutar.

La instrucción salto si negativo trabaja esencialmente igual al salto si cero, excepto en que ahora el salto ocurre si el contenido del acumulador es negativo.

6.5.1. Programas simples

Para mostrar las ideas que se han discutido hasta ahora, a continuación se escriben dos programas en lenguaje de máquina. El objetivo no es propiamente enseñar a programar, sino más bien explicar el lenguaje de máquina y la operación de la computadora digital.

En el primer programa se realiza la resta de un número N_1 de otro número N_2. Si el resultado es negativo, entonces se suma 10_{10} al número, mientras que si el resultado es positivo o cero, se le suma 5_{10}. Después de hacer todo esto, al resultado se le añade otro número N_3. El programa y sus direcciones de memoria se presenta en la tabla siguiente.

Dirección	Contenido binario	Contenido hexadecimal	Comentario
1	0000001000000000	0200	Limpiar el acumulador
2	0000000110010001	0191	Lee sustraendo
3	0001000000000000	1000	Complementar sustraendo
4	0001000100000000	1100	Sumar 1 al sustraendo (Complemento a 2)
5	0000000110010010	0192	Sumar minuendo
6	0010001001010000	2250	Salto si negativo
7	0000000101010111	0157	Suma el contenido de 57_{16} al acumulador
8	0000000000000000	0000	Continua
9	0000000110010011	0193	Suma el contenido de 93_{16} al acumulador
A	0111011100000000	7700	Detener
50	0000000101100010	0162	Sumar el contenido de 62_{16} al acumulador
51	0010000000001000	2008	Salto incondicional a la direccción 08_{16} (PC = 08_{16})
57	0000000000000101	0005	Dato $5_{16} = 5_{10}$
62	0000000000001010	000A	Dato $A_{16} = 10_{10}$
91	0000000000000001	0001	Dato $1_{16} = 1_{10}$
92	0000000000000111	0007	Dato $7_{16} = 7_{10}$
93	0000000000001001	0009	Dato $9_{16} = 9_{10}$

Considérese la operación de tal programa. Los números N_1, N_2 y N_3 se almacenan en las direcciones de memoria 91_{16}, 92_{16} y 93_{16}, respectivamente. Los valores de estos números son los siguientes:

$$N_1 = 1_{16} = 1_{10}$$
$$N_2 = 7_{16} = 7_{10}$$
$$N_3 = 9_{16} = 9_{10}$$

En este punto, se revisa la operación del programa paso a paso. La instrucción en la dirección de memoria 1_{16} causa que el acumulador se limpie, de tal modo que el valor inicial del acumulador es cero. La segunda instrucción (en la dirección de memoria 2_{16}) hace que el contenido de la dirección de memoria 91_{16} se sume al contenido del acumulador. Este valor es N_1, el cual debemos restar de N_2. Para hacer esto, se obtiene el complemento a 2, mediante complementar y sumarle una unidad. Las siguientes dos instrucciones en las localidades de memoria 3_{16} y 4_{16} realizan tales operaciones. De este modo, hasta este punto, el acumulador almacena el valor equivalente a $-N_1$.

La siguiente instrucción (en la dirección de memoria 5_{16}) provoca que el contenido de la dirección de memoria 92_{16} se sume al acumulador. Esto suma N_2 con $-N_1$, de modo que se obtiene la resta deseada $N_2 - N_1$. Es en este punto donde se decide si el resultado es negativo y se salta con la instrucción en la dirección 6_{16}. Esto significa que si el valor almacenado en el acumulador es negativo, entonces el flujo del programa cambia ("salta") a la dirección 50_{16}. Si el número almacenado en el acumulador no es negativo, entonces la siguiente instrucción que debe ejecutarse es aquélla contenida en la dirección 7_{16}. Tal instrucción causa que el contenido de la dirección de memoria 57_{16} se sume al acumulador. Como se ha almacenado un valor de 5_{10} en tal dirección, entonces este valor se suma al contenido del acumulador. Así, se tiene ahora en el acumulador el valor $N_2 - N_1 + 5_{10}$.

La instrucción en la dirección 8_{16} es *continua*. La computadora la obtiene, y prosigue a ejecutar la siguiente instrucción. Más adelante, se detalla que en realidad esta instrucción se requiere como un punto seguro para saltar de vuelta al cuerpo principal del programa.

La siguiente instrucción a ejecutar es aquélla contenida en la localidad de memoria con la dirección 9_{16}. Esta instrucción hace que el contenido de la localidad de memoria 93_{16} se sumen con el contenido del acumulador. Hasta este punto, entonces, se tiene que en el acumulador se ha realizado la operación $N_2 - N_1 + 5_{10} + N_3$. El cómputo finaliza, y la computadora debería

detenerse. En realidad, un programa más práctico requiere que se presente el resultado al usuario, por lo que sería necesario añadir instrucciones para tal fin. Pero tales instrucciones no se han incluido en este ejemplo, dado que todavía no se ha discutido la salida de datos.

Ahora bien, considérese de nuevo el salto en la dirección 6_{16}. Supóngase que se tienen valores diferentes almacenados en las direcciones 91_{16} y 92_{16}, de modo que la diferencia $N_2 - N_1$ resulta negativa. Al realizar el salto, resulta que la siguiente instrucción a ejecutar es aquélla almacenada en la dirección 50_{16}. Esta instrucción suma el contenido de la dirección 62_{16}, que es un 10_{10}, al contenido del acumulador. Por lo tanto, si $N_2 - N_1$ es negativo, se realiza la instrucción para obtener en el acumulador el valor $N_2 - N_1 + 10_{10}$.

Nótese que la última instrucción se encuentra en la dirección de memoria 50_{16}. Después de ejecutarla, la siguiente instrucción a ejecutar es aquélla contenida en la dirección 51_{16}, que es un salto incondicional que coloca la dirección 8_{16} en el PC. La instrucción siguiente a ejecutar es, entonces, aquélla contenida en la dirección de memoria 8_{16}. Esta es la instrucción *continua*, que en realidad no hace nada, pero que sirve aquí como un punto seguro de retorno. Así, la siguiente instrucción a ejecutar es aquélla en la dirección 9_{16}, que como se ha dicho anteriormente, se encarga de sumar el valor N_3 al contenido del acumulador, y el cómputo se detiene. De este modo, el resultado almacenado en el acumulador si $N_2 - N_1$ es negativo es el valor $N_2 - N_1 + 10_{10} + N_3$.

Supóngase ahora que se desea cambiar los datos y volver a ejecutar el programa. Como los datos se almacenan en las direcciones 91_{16}, 92_{16} y 93_{16}, se requiere entonces sólamente cambiar los valores que se encuentran almacenados, sin necesidad de modificar el resto del programa. Esto es muy conveniente, particularmente si el programa es muy largo.

En seguida se considera otro ejemplo de un programa simple en lenguaje de máquina. El programa anterior, aun cuando realiza lo estipulado, no hace algo que se pudiera considerar útil. De este modo, se propone realizar un programa con un objetivo mejor definido. Supóngase que se desea realizar la suma de los primeros 100 números enteros:

$$1 + 2 + 3 + 4 + \cdots + 98 + 99 + 100$$

Un programa que hace esta operación directamente tendría muchos pasos, y sería muy tedioso para programar. Sin embargo, la siguiente tabla

presenta un programa en lenguaje de máquina que utiliza los saltos para acortar el programa y eliminar la repetición.

Dirección	Contenido hexadecimal	Contenido binario
1	0200	0000001000000000
2	0181	0000000110000001
3	1100	0001000100000000
4	1481	0001010010000001
5	1000	0001000000000000
6	1100	0001000100000000
7	0180	0000000110000000
8	2190	0010000110010000
9	0200	0000001000000000
A	0181	0000000110000001
B	0182	0000000110000010
C	1482	0001010010000010
D	2001	0010000000000001
80	0065	0000000001100101
81	0000	0000000000000000
82	0000	0000000000000000
90	7700	0111011100000000

La operación de este programa comienza limpiando el contenido del acumulador con la instrucción en la dirección 1_{16}. La siguiente instrucción en la dirección de memoria 2_{16} provoca que el contenido de la localidad de memoria con dirección 81_{16} se sume con el acumulador. Al principio, tal número tiene valor cero. El contenido de esta localidad de memoria es el siguiente número a ser sumado. La siguiente instrucción (en la dirección 3_{16}) le suma 1 al contenido del acumulador. Con la siguiente instrucción, en la dirección 4_{16}, el contenido del acumulador ahora se almacena en la dirección 81_{16}. El valor anterior en esta dirección se pierde.

El número que se acaba de almacenar en la dirección 81_{16} eventualmente se añade a la suma, pero antes es necesario verificar si no excede el valor de 100_{10}. Para hacer esto, el contenido del acumulador se complementa (con la instrucción en la dirección 5_{16}) y se incrementa (con la instrucción en la dirección 6_{16}). Así, se ha calculado el complemento a 2 del número a verificar. La instrucción en la dirección 7_{16} hace que el número almacenado en la dirección 80_{16} se sume al complemento a 2 obtenido. Nótese que el valor almancenado en esta dirección es el valor 101_{10}. Si después de la operación el

contenido del acumulador es cero, no se debe continuar sumando el contenido de la localidad de memoria 81_{16} con la suma. La siguiente instrucción, en la dirección 8_{16}, es un salto si se verifica un valor cero en el acumulador. De hecho, si el número almacenado en la dirección 81_{16} es 101_{10}, el resultado de la diferencia es cero, por lo que la siguiente instrucción a ejecutarse después del salto se encuentra en la dirección 90_{16}. La instrucción almacenada en esta dirección es 77_{16}, que significa detener el proceso. Así, sólamente si el número que se almacena en la localidad de memoria 81_{16} tiene el valor 101_{10}, el cómputo se completa. Pero si el valor almacenado en tal dirección es menor que 101_{10}, entonces el proceso prosigue.

Regresando a la instrucción de salto en la dirección 8_{16}, supóngase que que el salto falla, es decir, el contenido del acumulador no es cero. La siguiente instrucción a realizarse es aquélla contenida en la dirección 9_{16}, y ésta causa que el acumulador se limpie.

Siguiendo con el programa, la instrucción en la dirección A_{16} hace que el contenido de la dirección 81_{16} se sume con el contenido del acumulador. Nótese que el valor en esta dirección es 1_{10}. La instrucción en la dirección B_{16} hace que el contenido de la dirección 82_{16} se sume al contenido del acumulador. Tal contenido en la dirección 82_{16} es todavía cero.

Ahora bien, la instrucción de la dirección C_{16} hace que el contenido del acumulador se coloque en la dirección de memoria 82_{16}. Este valor es 1_{10}, de modo que ésta es la primera suma.

La siguiente instrucción en la dirección D_{16} hace que se salte incodicionalmente a la dirección 1_{16}. Por lo tanto, el programa regresa a la primer instrucción, haciendo un ciclo. Si se va a través del programa de nuevo, en esta ocasión se almacena un valor 2_{10} en la dirección de memoria 81_{16}. Después de verificar que este número es menor que 101_{10}, tal valor se suma al número almacenado en la dirección 82_{16} ($2_{10} + 1_{10} = 3_{10}$).

Si se va a través del programa varias veces, en ciclos, es posible observar lo siguiente: en cada ciclo se van sumando los valores $1, 2, 3, \ldots$ a la suma. La suma acumulada se va almacenando en la dirección 82_{16}. Antes de sumarse, el número se almacena en la dirección 81_{16}, de modo que antes de que se realice la suma, se verifica si tal valor no es mayor que $65_{16} = 101_{10}$, mediante realizar la resta usando el complemento a 2. Cuando el resultado de la resta sea cero, el proceso termina.

En caso de que se desee cambiar el programa para obtener la suma de los primeros N números $(1+2+3+\cdots+N)$, es necesario tan solo cambiar el número almacenado en la dirección 80_{16} de 101_{10} al valor $N+1_{10}$.

Como es el caso en los dos ejemplos anteriores de programas simples en lenguaje de máquina, antes de considerar los detalles para escribir un programa, es necesario entender primero la idea básica que debe realizarse, es decir, el *algoritmo* que realiza. Por ejemplo, se podría expresar la idea básica del último programa de la siguiente forma. Se desea obtener el resultado de la suma de los 100 primero enteros. Al resultado se le puede dar el nombre de **SUMA**, y a un número entero, el nombre de **N**. Así, al inicio, se puede hacer que ambos, **SUMA** y **N** tengan valor 0. En seguida se incrementa **N** en una unidad. Es necesario verificar ahora que el valor de **N** sea menor que 101_{10}. Si este es el caso, entonces **N** se suma a **SUMA**, y el proceso se repite. Por otro lado, si **N** es igual a 101_{10}, entonces se detiene el proceso, y el valor de **SUMA** representa el resultado.

La idea detrás de un programa puede representarse de varias maneras. Un ejemplo utilizado tradicionalmente en varios cursos de programación, es el *diagrama de flujo* (*flow chart*). En general, se trata de una figura gráfica que sirve para representar y entender de manera más sencilla al programa, en lugar de utilizar sólamente una descripción verbal. La Figura 6.6 muestra un ejemplo de la descripción del programa anterior en términos de un diagrama de flujo.

Nótese cómo el diagrama describe el flujo de control del programa. El bloque con forma de diamante se utiliza cuando debe tomarse una decisión (es decir, si hay un salto condicional). Si $N-101=0$, entonces se sale del bloque de decisión por la salida marcada con "Si", y el proceso termina. En caso contrario, si $N-101$ no tiene valor 0, se sale del bloque de decisión por la salida con etiqueta "No", se incrementa **N** en una unidad y la operación se repite.

6.5.2. Salida de datos

En esta sección se considera la salida de datos de la computadora mediante el ejemplo de un programa sencillo. Cada dispositivo de entrada/salida se conecta a una *tarjeta de interface* (*interface board*), la cual a su vez se conecta con el bus de la computadora. La tarjeta de interface convierte la información digital del bus a alguna forma de información que pueda ser usada por el dispositivo de entrada/salida (y viceversa). Cada dispositivo

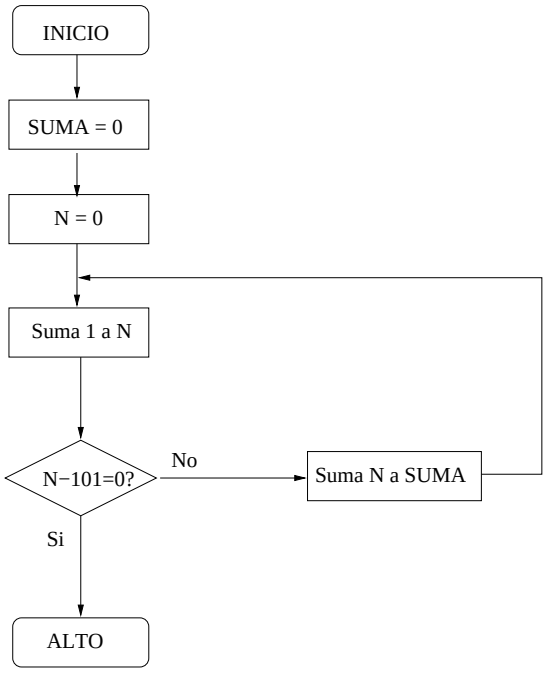

Figura 6.6: Diagrama de flujo del algoritmo que suma los primeros 100 números enteros.

de entrada/salida generalmente tiene su propia tarjeta de interface, pero frecuentemente una sola tarjeta puede usarse para muchos dipositivos. Es en las tarjetas de interface donde se encuentran los buffers (memoria) asociados con los dispositivos de entrada/salida, y se utilizan para almacenar datos a enviar o recibir. En algunas computadoras, los buffers son parte de la memoria principal. El bus conecta todas las tarjetas de interface, cada una de las cuales se conecta con su dispositivo de entrada/salida respectivo. Los dispositivos de entrada/salida, como terminales, impresoras, unidades de cinta, disco, etc., son llamados también con el nombre genérico de *periféricos* (*peripherals*).

Supóngase que en la computadora sencilla que se ha descrito hasta ahora, todos los datos a salir deben colocarse en el acumulador y entonces, mediante la instrucción adecuada, se transfiere a las terminales del bus reservadas para la entrada/salida. Para hacer esto, se utiliza la siguiente instrucción en lenguaje de máquina:

$$00110000_2 = 30_{16}$$

La instrucción 30_{16} sólo especifica que hay datos por salir. Sin embargo, la instrucción también debe indicar cuál dispositivo debe utilizarse, y qué función debe realizar tal dispositivo. Por ejemplo, supóngase que se tiene una unidad de cinta. La información es normalmente almacenada en la cinta en forma de grupos de datos llamados "bloques". Se puede, entonces, hacer la instrucción de tal modo que se le ordene a la unidad de cinta recorrer toda la cinta para comenzar con el primer bloque. Alternativamente, se puede hacer que la instrucción indique a la unidad de cinta que salte uno o varios bloques de datos, de modo que nuevos datos puedan escribirse sin afectar los datos ya contenidos en la cinta.

Como se considera para la computadora una palabra de instrucción con 16 bits, entonces la forma de la instrucción de salida debe ser:

$$00110000 d_3 d_2 d_1 d_0 f_3 f_2 f_1 f_0$$

Los primerso 8 bits indican que se trata de una instrucción de salida. Los bits d_i y f_i son adicionales, e indican el tipo y función del dispositivo. De hecho, los bits d_i se utilizan para indicar un código del dispositivo de salida, mientras que los bits f_i indican si el primer bloque debe o no saltarse. Se propone el siguiente código para diferentes dispositivos:

Dispositivo	Binario	Hexadecimal
Terminal	0000	00
Impresora	0001	01
Unidad de Disco Flexible	0010	02
Unidad de Cinta	0011	03

Para las funciones de la unidad de cinta del ejemplo, se tienen las siguientes funciones:

Función	Binario	Hexadecimal
Recorrer la cinta al inicio	0001	01
Saltar al siguiente bloque	0010	02

Así, si por ejemplo se desea desplegar datos en la terminal, la instrucción sería:

$$0011000000000000_2 = 3000_{16}$$

En este caso, los últimos dos bits no tienen un significado definido, por lo que pueden ignorarse.

O bien, supóngase que se desea colocar un resultado en una cinta magnética, y se desea que la cinta inicie desde el principio y se escriba sobre cualquier otra información que ya se encuentre en ella. Entonces, la instrucción sería:

$$0011000000110001_2 = 3031_{16}$$

Por otro lado, si se desea saltar en la cinta hasta el último bloque de datos y escribir nuevos datos después, la instrucción tendría la forma:

$$0011000000110010_2 = 3032_{16}$$

Para el caso de la unidad de disco flexible habría instrucciones similares, en cuyo caso la pista donde puede escribirse la salida puede especificarse.

Todas las interfaces de entrada/salida se conectan a las mismas terminales del bus, que utilizan un decodificador similar al que se presenta en la Figura 6.3, causando que la interface del dispositivo apropiado se active y opere de forma adecuada. El decodificador recibe los bits:

$$d_3 d_2 d_1 d_0 f_3 f_2 f_1 f_0$$

Esto envía una señal de instrucción a la interface apropiada, de modo que el dispositivo de entrada/salida correcto opere y permita la salida o entrada de datos.

Cuando se da una salida de datos, primero se coloca una instrucción de salida en las terminales del bus de entrada/salida. En seguida, los datos numéricos binarios se colocan en las mismas terminales del bus. Nótese que esto números de proveen en *secuencia*. Supóngase que, en esta computadora sencilla, hay una instrucción que causa que el contenido del acumulador se coloque en las líneas de entrada/salida del bus. Tal instrucción puede ser:

$$01110000_2 = 70_{16}$$

De este modo, primero se requiere la instrucción de salida, y enseguida, la instrucción 70_{16}. Después de esto, se coloca un dato de salida en el acumulador, y a continuación, se da de nuevo la instrucción 70_{16}. Por ejemplo, supóngase que se desea imprimir el número almacenado en la dirección de memoria 91_{16} en la terminal, y además, almacenar el número $0011000000000000_2 = 3000_{16}$ en la dirección de memoria 26_{16} (nótese que ésta es la instrucción que opera sobre la terminal). Entonces, la salida de datos se realiza mediante ejecutar las siguientes instrucciones (en hexadecimal):

0200
0126
7000
0200
0191
7000

La primera instrucción limpia el acumulador. Enseguida se suma el contenido de la dirección 26_{16} al contenido del acumulador, lo que coloca la instrucción de salida por terminal en el propio acumulador. La siguiente instrucción (7000_{16}) causa la salida del contenido actual del acumulador sobre las líneas del bus. El acumulador se limpia, y entonces se le colocan los datos numéricos de salida. Ya que el acumulador se limpió anteriormente, ahora contiene los datos númericos de salida. Enseguida, este número se provee a las líneas de entrada/salida. Nótese que este procedimiento de

salida de datos puede variar de computadora a computadora. Sin embargo, entender este procedimiento debe permitir comprender los procedimientos en diferentes computadoras. De hecho, los detalles de salida de datos en una computadora dada pueden involucrar diferentes instrucciones, pero las ideas básicas son esencialmente las mismas.

Los dispositivos de entrada/salida son comúnmente mucho más lentos que la operación de la computadora. De hecho, es muy posible que los datos se provean en una tasa más rápida de lo que son impresos. El buffer de entrada/salida (véase la Sección 6.4) intenta resolver este problema. Sin embargo, si los datos se proveen lo suficientemente rápido, el buffer puede llenarse. Es por esto que la unidad de entrada/salida cuenta con un registro de un bit llamado *bandera* (*flag*). Si el buffer se llena, la bandera toma valor 0; si el buffer no está lleno y puede recibir más datos, la bandera tiene valor 1. Cuando el valor de la bandera es 0, esto actúa como una señal para la unidad de control y el proceso de salida entra en una pausa. Sin embargo, la salida de datos prosigue. Una vez que el registro de entrada/salida puede recibir más datos, la bandera toma el valor de 1 y el proceso de salida continua.

Algunas computadoras tienen otras banderas más sofisticadas, conocidas como *interrupciones* (*interrupts*). Estas señales causan que el proceso de la computadora se detenga y prosiga después si alguna acción o problema predecible ocurre en un dispositivo de entrada/salida. Por ejemplo, si la impresora se queda sin papel, el proceso se detiene.

Se pueden escribir programas que reciban datos de entrada en momentos específicos. En tal caso, los datos de entrada se colocan en el registro de entrada/salida, y cuando se da la instrucción apropiada de entrada, los contenidos de este registro se copian al acumulador. Así, se puede dar entrada de datos cuando se requiera.

6.6. Lenguaje ensamblador

Programar utilizando lenguaje de máquina resulta muy tedioso. El programador debe recordar los códigos de instrucción que corresponden a cada instrucción, o al menos, debe continuamente revisar su definición. Por otro lado, en casi todas las computadoras reales la lista de instrucciones es mucho mayor que la definida para la computadora sencilla. Esto hace que la labor de utilizar las instrucciones sea muy difícil. Más aún, el programador debe recordar en todo momento las direcciones de memoria donde se encuentran

almacenadas las variables y las instrucciones, lo que es particularmente dificultoso cuando entre las instrucciones se involucran saltos. Como un ejemplo de estos problemas, supóngase que se requiere realizar la siguiente suma:

$$x = a + b + c + d$$

Debe haber una localidad de memoria reservada para cada una de las variables x, a, b, c y d, y las direcciones de esas localidades deben en todo momento recordarse. Si se tiene un número relativamente pequeño de variables, la tarea es más o menos simple; sin embargo, en un programa grande, con muchas variables, el programador debe entonces crear una tabla para recordar la posición de las variables, lo que normalmente consume mucho tiempo y esfuerzo.

Para hacer la labor de programación menos tediosa, se han desarrollado otros tipos de lenguajes para el uso del programador, en lugar de utilizar lenguaje de máquina. En esta sección, se discute el *lenguaje ensamblador* (*assembly language*), el cual está muy cercanamente relacionado con el lenguaje de máquina, pero resulta mucho más fácil de utilizar por parte de los programadores.

Siempre que una computadora realiza alguna clase de proceso, un programa en lenguaje de máquina (binario o ejecutable) dirige su operación. Consecuentemente, siempre que otra forma de lenguaje se utilice, ésta debe ser traducida a lenguaje de máquina. Un programa conocido como *ensamblador* (*assembler*) dirige la acción de la computadora, de modo que traduce un programa del lenguaje ensamblador a lenguaje de máquina. El lenguaje ensablador es, entonces, un conjunto de instrucciones que se utilizan para escribir un programa en ensamblador. A continuación, primero se describe un lenguaje ensamblador simple, y luego se comenta cómo la computadora lo usa.

Las instrucciones en lenguaje de máquina se realizan en términos de códigos binarios. En lenguaje ensamblador, los códigos se reemplazan por *pnemónicos* (*pnemonics*), que comúnmente son más fáciles de recordar que los códigos binarios. Por ejemplo, en lugar de escribir $01_{16} = 00000001_2$ para la operación de suma, simplemente se escriben las letras ADD (la palabra inglesa para "suma"). Nótese que ADD es mucho más fácil de recordar que 00000001. Por supuesto, se considera aquí que se puede dar una secuencia de letras como entrada a la computadora. Esto implica el uso de los códigos mencionados en la Sección 4.5.

Enseguida, se presenta un lenguaje ensamblador sencillo. Nótese que, como es el caso de los lenguajes de máquina, los lenguajes ensabladores no tienen un estándar. De este modo, por convención se utilizan tres letras para designar una instrucción en lenguaje ensamblador. De hecho, es conveniente utilizar un número fijo de letras para las instrucciones en lenguaje ensamblador, pero esto no siempre es necesario. En general, se puede pensar que una instrucción en ensamblador corresponde directamente con una instrucción en lenguaje de máquina. Nótese que los pnemónicos del lenguaje ensamblador son mucho más fáciles de recordar que los códigos binarios que representan. Otra ventaja de utilizar lenguaje ensamblador es que el programador no tiene que recordar las direcciones en memoria de los datos almacenados. Tales valores se les da el nombre de *variables*, y como parte del lenguaje, puede dárseles un nombre. Comúnmente, tal nombre puede constar hasta de seis caracteres alfanuméricos.

Instrucción en lenguaje de máquina	Instrucción en lenguaje ensamblador
01	ADD
02	CLE
03	AND
04	ORR
05	XOR
06	SHR
07	SHL
10	COM
11	INC
12	NEC
13	ZEC
14	PIM
20	BRU
21	BRZ
22	BRN
70	CON
77	STP

Como ejemplo de un programa escrito en lenguaje ensamblador, a continuación se re-escribe el programa del primer ejemplo de la Sección 6.3, ahora en lenguaje ensamblador:

```
CLE
ADD DATA1
ADD B
PIM ANS
STP
DATA1:3
B:4
ANS:0
```

DATA1, B y ANS son nombres de variables. DATA1 es equivalente a especificar la dirección de memoria 10_{16}. Similarmente, se escribe B en lugar de la dirección 11_{16} y ANS en lugar de 12_{16}. Así, el programador no requiere recordar las direcciones de memoria. En cualquier momento en que se hace referencia a DATA1, se utiliza la dirección adecuada de memoria.

Ahora bien, a continuación se describe cómo la computadora trabaja sobre un programa escrito en lenguaje ensamblador para producir un programa en lenguaje de máquina. Se utiliza el programa ensamblador previamente mencionado. Este programa se escribe sólo una vez, y la mayoría de los programadores no necesitan hacerle modificaciones. El ensamblador se carga en memoria de la misma forma en que se carga cualquier otro programa que ejecute la computadora. Enseguida, el programa en lenguaje ensamblador se le provee al ensamblador como *datos de entrada*. El ensamblador traduce el programa en lenguaje ensamblador a un programa en lenguaje de máquina, y sólo entonces se cuenta con un programa que puede ser ejecutado por la computadora. Sin embargo, no fue necesario que el programador escribiera código en lenguaje de máquina. Sólo es necesario escribir un programa en ensamblador.

Se puede utilizar alternativamente el siguiente procedimiento: el ensamblador traduce el programa en lenguaje ensamblador a un programa en lenguaje de máquina que se almacena en memoria, de modo que éste último puede a la vez almacenarse en, por ejemplo, un disco. Después de esto, es posible "cargar" el programa en lenguaje de máquina a partir del disco, y ejecutarlo. De nuevo, el programador no requiere escribir su programa en lenguaje de máquina.

En este punto surge la pregunta: ¿cómo trabaja un ensamblador? Recuérdese que cada enunciado en un programa en lenguaje ensamblador se le provee como datos de entrada. Si se utiliza una terminal, cada letra genera un código (véase Sección 4.5), de modo que cada instrucción en lenguaje

ensamblador puede representarse por diferentes secuencias de ceros y unos. Cuando el ensamblador se carga en la computadora, cada código se almacena en una dirección diferente de memoria. De manera similar, cada código que corresponde al programa en lenguaje ensamblador se almacena en otras direcciones de memoria. Ahora, cuando un enunciado del programa en lenguaje ensamblador se proporciona al ensamblador, su código se compara con los códigos de las instrucciones en lenguaje ensamblador. Cuando se encuentra un código igual al código de la instrucción, la instrucción correspondiente en lenguaje de máquina se obtiene y añade al programa en lenguaje de máquina que se está generando.

Respecto a la transformación de variables a direcciones de memoria, se sabe que todas las variables utilizadas en un programa en lenguaje ensamblador se escriben siempre *después* de una instrucción. Así, cualquier información alfanumérica que siga después de las tres letras de una instrucción puede identificarse como variable. El ensablador, entonces, requiere tomar en cuenta el número y tipo de variables, a fin de apartar las direcciones necesarias para almacenar las variables del programa.

Durante la formación del programa en lenguaje de máquina a partir de un programa en lenguaje ensamblador, es común que este último se lea dos veces (aunque en algunas computadoras se lee únicamente una sola vez, y ya almacenado en memoria, se revisa dos veces). La primera lectura o pasada se utiliza para identificar todas las variables y establecer localidades de memoria para cada una de ellas. El programa se va revisando, y conforme se van hallando variables, se almacena su código. El nombre usado en el programa para la variable se almacena en una *tabla de símbolos*. Esta tabla eventualmente contiene también las direcciones en memoria de cada variable. Una variable puede utilizarse muchas veces dentro del programa, pero puede aparecer solo una vez en la tabla de símbolos.

En el espacio de memoria del programa, debe haber localidades de memoria asignadas tanto para cada variable como para cada instrucción del programa. Durante la primera pasada, una localidad de memoria se reserva para cada instrucción; las localidades de memoria para las variables se reservan después de reservar localidades para las instrucciones. Considérese por ejemplo el programa anterior en lenguaje ensamblador. La primera instrucción es CLE. Una palabra de memoria se ha reservado para esta instrucción. Es importante notar que las localidades de memoria se reservan *en orden estrictamente secuencial*. La primera instrucción recibe la dirección de memoria con menor valor numérico, por ejemplo, $00000001_2 = 01_{16}$. A

continuación, se encuentra la instrucción ADD DATA1. Se reserva la dirección de memoria 02_{16} para ADD, mientras que se da entrada a DATA1 en la tabla de símbolos, pero todavía no se ha reservado una localidad de memoria para ella. Similarmente, al encontrarse ADD B se reserva la dirección 03_{16} para ADD, y se da entrada a B en la tabla de símbolos. La instrucción siguiente, PIM, se coloca en la dirección 04_{16}, y se da entrada a la variable ANS en la tabla de símbolos. Cuando se encuentra la instrucción STP, se le asigan la dirección 05_{16}. Ahora bien, se encuentra la cadena DATA1:3. Los dos puntos indican que DATA1 es una variable. Es hasta ahora que se le asigna la dirección en memoria 06_{16} a DATA1. Tal información se añade a información sobre DATA1 en la tabla de símbolos. En forma parecida, se reserva la dirección 07_{16} para B, cuyo valor es 4, y finalmente, se reserva la dirección 08_{16} para ANS, cuyo valor es 0.

Como es notorio, se da un valor inicial a cada dato mediante los dos puntos (:). Este caracter indica que el número que le sigue es el valor que debe almacenarse en la dirección de memoria de la variable adecuada. Nótese que debe haber una inicialización para cada una de las variables, aunque esto no sea del todo cierto para todo ensamblador. Por ejemplo, después de que la suma se ha realizado, se coloca el resultado en la variable llamada ANS. Se requiere, entonces, que ANS tenga un valor conocido desde un principio, por lo que se hace la indicación ANS:0 para ello. Esto es necesario para reservar memoria para la variable ANS. Nótese que el valor 0 es arbitrario. Se puede utilizar cualquier valor. Una vez que ANS se coloca en memoria, su valor se re-escribe una y otra vez. Se supone aquí que las localidades de memoria se asignan en orden, partiendo de una dirección inicial 01_{16}. Sin embargo, varios ensabladores pueden variar en tal procedimiento.

En la primera pasada del ensamblador se reservan todas las localides de memoria necesarias para el programa. Durante la segunda pasada los pnemónicos se convierten a sus códigos equivalentes en lenguaje de máquina, y se almacenan en las direcciones de memoria reservadas para las instrucciones. Esta es la primera parte de la palabra en lenguaje de máquina. Si el nombre de una variable está asociado con una instrucción, entonces se le busca en la tabla de símbolos, de modo que la dirección de la variable se vuelve la segunda parte de la palabra en lenguaje de máquina. Este proceso se repite paso a paso, instrucción por instrucción, hasta que todo el programa en lenguaje ensamblador se ha convertido en una programa en lenguaje de máquina, cuyas instrucciones se almacenan en secuencia.

Después de ejecutar el programa ensamblador, el programa generado se escribe generalmente en disco. La memoria principal se limpia, y el nuevo programa está listo para ejecutarse.

Puede ser que la memoria no sea lo suficientemente grande para almacenar el ensamblador, la tabla de símbolos y el programa en lenguaje de máquina. Pero si la memoria es lo suficientemente grande para completar la primera pasada, entonces se puede hacer lo siguiente: se puede iniciar la segunda pasada, pero si la memoria se llena, entonces la segunda pasada entra en una pausa. La memoria que contiene la parte del programa en lenguaje de máquina se puede transferir a disco, después de lo cual se libera parte de la memoria, y se continua con la segunda pasada. Este proceso se puede repetir mientras la memoria se llene, y hasta que el programa en lenguaje de máquina termine de generarse.

6.7. Lenguajes de alto nivel

Aun cuando el lenguaje ensamblador es mucho más fácil de utilizar que el lenguaje de máquina, resulta todavía muy tedioso de programar. Por ejemplo, supóngase que se desea escribir un programa que evalúe la siguiente ecuación:

$$x = (a + b)(c - d)/(a - b)$$

Para realizar esta simple ecuación se requiere escribir muchas líneas de código en ensamblador, lo que requiere un gran esfuerzo para el programador. Es por esto que se han desarrollado otro tipo de lenguajes que permiten al programador trabajar con elementos de un nivel de abstracción más alto que lo que puede expresarse utilizando lenguaje ensamblador. Existen en la actualidad un gran número de lenguajes de programación de alto nivel que pueden utilizarse para expresar operaciones cada vez más complejas. Ejemplos de estos lenguajes son Fortran, Pascal, y C.

A continuación, se muestra cómo la expresión anterior puede evaluarse utilizando lenguaje C.

```
int a = 3, b = 4, c = 5, d = 3, x = 0;
x = (a + b)*(c - d)/(a - b);
printf(x);
```

Nótese que el asterisco (*) es el símbolo utilizado en C para representar la multiplicación.

El programa que realmente se ejecuta en la computadora es un equivalente de este programa, pero en lenguaje de máquina. Cuando se utiliza un lenguaje de alto nivel, el programa *fuente* (escrito en el lenguaje de alto nivel) se provee como datos básicos para la construcción de su equivalente en lenguaje de máquina, de manera muy similar a como un programa en lenguaje ensamblador son datos para el ensamblador. Sin embargo, el programa que convierte un programa fuente a su equivalente en lenguaje de máquina es mucho más complejo y sofisticado que el ensamblador, ya que una sola instrucción en lenguaje de alto nivel equivale a una secuencia de instrucciones en lenguaje de máquina. Los programas que convierten un programa en lenguaje de alto nivel a lenguaje de máquina pueden ser *compiladores* (*compilers*) o *intérpretes* (*interpreters*). Aun cuando permiten una mayor flexibilidad en la labor de programación, el uso de lenguajes de alto nivel implica el seguimiento de reglas rigurosas y restrictivas para el programador. Sin embargo, los lenguajes de alto nivel siguen siendo la opción más adecuada para el desarrollo de programas, comparativamente con el lenguaje ensamblador y el lenguaje de máquina.

Bibliografía

1. D.D. Gajski. *Principios de Diseño Digital.* Pearson Education, 1997.
2. M.S. Ghausi. *Circuitos Electrónicos discretos e integrados.* Interamericana, 1987.
3. J.P. Hayes. *Diseño de Sistemas Digitales y Microprocesadores.* McGraw-Hill, 1984.
4. M. Morris Mano. *Arquitectura de Computadoras.* Prentice-Hall, 1983.
5. M. Morris Mano. *Diseño Digital.* Prentice-Hall, 1987.
6. M. Morris Mano. *Lógica Digital y Diseño de Computadores.* Prentice-Hall, 1979.
7. M. J Murdocca y V.P. Heuring. *Principios de Arquitectura de Computadoras.* Prentice-Hall, 2002.
8. W. Stallings. *Organización y Arquitectura de Computadores.* Prentice-Hall, 2000.
9. H. Taub and D. Schilling. *Digital Integrated Electronics.* McGraw-Hill, 1977.
10. R.J. Tocci and L.P. Laskowski. *Microprocessors and Microcomputers.* Prentice-Hall, 1982.
11. C.A. Wiatrowski and C.H. House. *Circuitos Lógicos y Sistemas de Microcomputadoras.* Limusa, 1987.