GRADO EN INGENIERÍA INFORMÁTICA

Módulo de Formación Básica

FUNDAMENTOS DE COMPUTADORES

Dr. D. Daniel Romero Pérez

viu Universidad Internacional de Valencia

Este material es de uso exclusivo para los alumnos de la VIU. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la VIU, sin autorización expresa de la misma.

Edita
Universidad Internacional de Valencia

Grado en

Ingeniería Informática

Fundamentos de Computadores

Módulo de Formación Básica

6ECTS

Dr. D. Daniel Romero Pérez

Índice

Leyenda

abc **Glosario**
Términos cuya definición correspondiente está en el apartado "Glosario".

Tema 1.
Introducción a los computadores

Los computadores digitales desempeñan un rol muy importante en la sociedad moderna actual, por lo que muchas veces se dice que estamos en la era de la información. Hoy en día los computadores se encuentran integrados en muchas ramas de nuestra economía y de nuestra sociedad, incluyendo actividades tan diversas como las transacciones de negocios, las comunicaciones, el transporte, el tratamiento médico y el ocio. En el área industrial desempeñan un papel sumamente importante encargándose del diseño, la producción, la distribución y las ventas de distintos productos. En el campo de la ciencia y el desarrollo ingenieril han contribuido de forma relevante por medio del descubrimiento de grandes hallazgos científicos y de grandes avances en los campos aeroespacial, la bioingeniería, las tecnologías de la salud y de los alimentos, de la construcción, entre otras, que de otra manera hubieran sido inalcanzables o tardarían decenas de años en salir a la luz.

Una de las características más importante de un computador digital es su capacidad de generalizar. Por medio de una secuencia de instrucciones que constituyen lo que llamamos **programa**, éste puede operar con cualquier tipo de datos especificado por el usuario. Tanto los datos como el programa se pueden cambiar en función de las necesidades concretas del usuario, de ahí su gran flexibilidad. Como resultado, los computadores digitales de propósito general pueden ejecutar una gran variedad de tareas de procesamiento de información en un amplio espectro de aplicaciones.

El ejemplo más conocido de un **sistema digital** es la **computadora de propósito general**. Un sistema digital se ocupa de manipular elementos discretos de información, constituidos por un conjunto finito de elementos de información discreta. Los elementos discretos de información se representan en un sistema digital por cantidades físicas llamadas señales. Estas señales eléctricas no son más que voltajes y corrientes que en la mayoría de los sistemas digitales, usan solamente dos valores discretos, 0 y 1, por lo que se denominan **señales binarias.**

Los valores discretos se especifican mediante rangos de valores de voltajes denominados nivel ALTO (*HIGH*) y nivel BAJO (*LOW*). Los valores de voltajes para los niveles ALTO y BAJO difieren para las entradas y las salidas de los circuitos digitales, siendo los rangos de las entrada mayores que los de las salidas. Esta diferencia permite que los circuitos funcionen correctamente a pesar de las posibles variaciones que pueden ocurrir en su comportamiento, y debido a la presencia de ruido que pueden restar o sumar pequeños valores de voltajes en las salidas. Típicamente, para el caso de las salidas, el rango de voltaje para el nivel ALTO oscila entre 4,0 y 5,5 voltios, mientras que para el nivel BAJO se encuentra entre 0,5 y 1,0 voltios. En el caso de las entradas, un valor se reconoce como nivel ALTO si se encuentra entre 3,0 y 5,5 voltios, y como nivel BAJO si está entre 0,5 y 2,0 voltios.

1.1. Estructura general de computador

La figura 1 muestra el diagrama en bloques de un computador digital de propósito general. La **memoria** se ocupa de almacenar tanto los programas como los datos de entrada, de salida y los intermedios. El bloque denominado **CPU** (Central Processing Unit) por sus siglas en inglés, incluye la **unidad aritmética/lógica** (*ALU*) y la **unidad de control** (CU). La **ALU** ejecuta las operaciones aritméticas y de otro tipo que se especifican en el programa. La **unidad de control** se encarga de

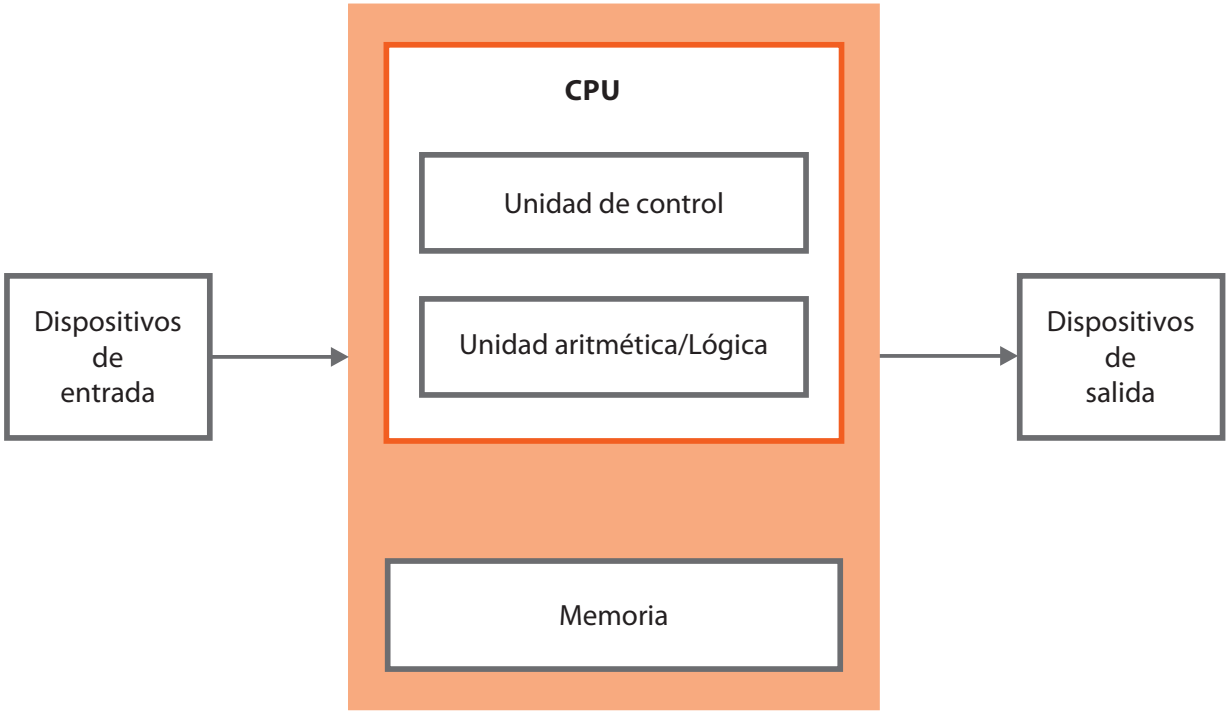


Figura 1. Diagrama en bloques de un computador de propósito general. Fuente: Arquitectura de Von Neumann. En Wikipedia. Recuperado el 17 de octubre de 2016. Pulse aquí para ver la fuente.

supervisar el flujo de información entre las diferentes unidades. Tanto el programa como los datos son transferidos a la memoria mediante un dispositivo de entrada, como bien puede ser un teclado. Los resultados de los cálculos realizados pueden ser presentados al usuario mediante un dispositivo de salida, como es el caso de la pantalla de un monitor. Asimismo, un computador digital puede alojar varios dispositivos de entrada/salida, incluyendo un disco duro, CD/DVD-ROM, impresora y escáner, convirtiéndose así en un sistema digital muy potente, capaz de realizar cálculos aritméticos complejos y de ser programado para tomar decisiones basadas en condiciones internas y externas.

1.2. Introducción a los sistemas digitales

1.2.1. Definición de sistema digital

Un **sistema digital** es un dispositivo diseñado para la generación, transmisión, procesamiento o almacenamiento de **señales digitales.** También se puede decir que es una combinación de dispositivos diseñados para manipular cantidades físicas o información que estén representadas en forma digital; es decir, que sólo puedan tomar valores discretos. La figura 2 muestra la inserción de un sistema digital dentro de un sistema más complejo, conformado también por componentes que manejan señales de entrada/salida analógicas.

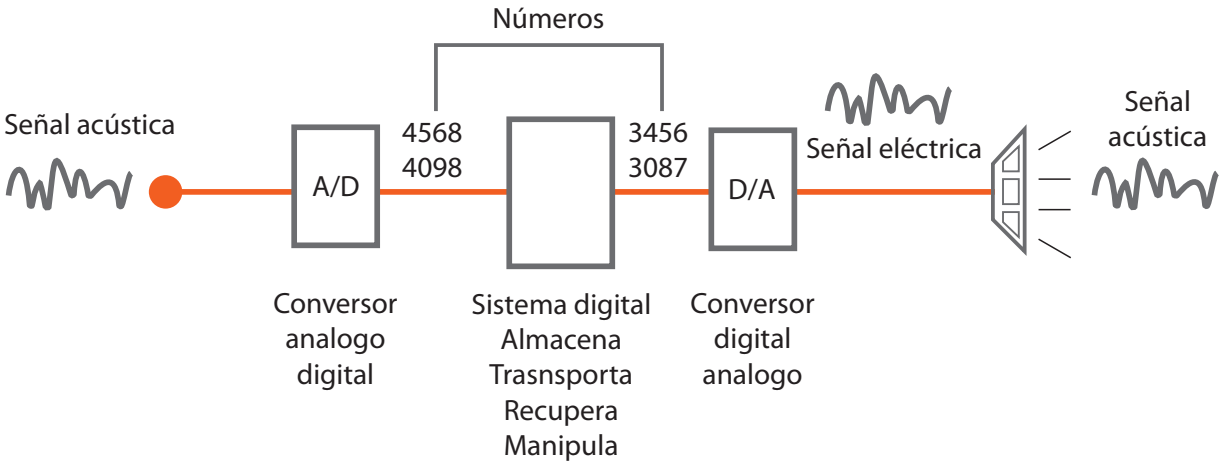


Figura 2. Sistema digital incorporado dentro un sistema mixto (analógico digital).
Fuente: Electronica 2 - UTP. (2017). En Sistemas Digitales. [online]. Pulse aquí para ver la fuente.

1.2.2. Señal digital

Las **señales digitales**, a diferencia de las analógicas, no varían de forma continua sino que cambian en pasos o en incrementos discretos. La mayor parte de las señales digitales utilizan códigos binarios o de dos estados. Para obtener las señales digitales, se realiza un proceso de conversión a partir de las señales analógicas aplicando los siguientes pasos:

- **Muestreo:** consiste en tomar muestras periódicas de la amplitud de la onda analógica. La velocidad con la que se toman las muestras, es decir, el número de muestras por segundo, se conoce como **Frecuencia de Muestreo**.

- **Cuantificación:** consiste en medir el nivel de voltaje de cada una de las muestras y asignar un único nivel de salida.

- **Codificación:** consiste en traducir los niveles de salidas asignados durante la cuantificación al sistema binario. También existen otros sistemas de codificación que pueden ser utilizados.

En la figura 3 se muestra gráficamente los tres pasos empleados en la conversión de las señales digitales.

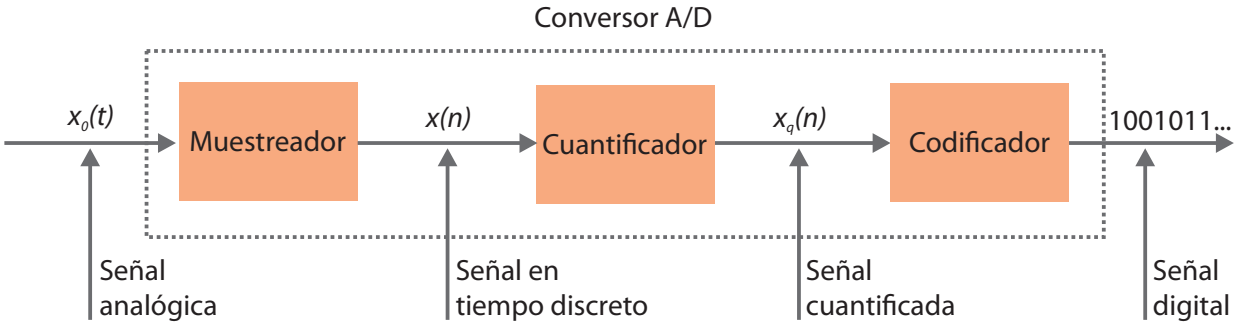


Figura 3. Conversión de una señal analógica en digital. Fuente: Conversor de señal analógica a digital. (2017). Es.wikipedia.org. Recuperado el 7 February 2017. Pulse aquí para ver la fuente.

1.2.3. Tipos de sistemas digitales

Los sistemas digitales se clasifican en dos tipos que son:

Sistemas Combinacionales: son aquellos en los que la salida del sistema sólo depende de la entrada actual, sin que se tenga en cuenta estados anteriores de la entrada o la salida. Por lo tanto, no necesita contener módulos de memoria. Estos sistemas están compuestos únicamente por **compuertas lógicas** interconectadas entre sí, siguiendo una función lógica que emplea las operaciones básicas del **álgebra de Boole**, por lo que también se conocen como **funciones booleanas.**

Entre los circuitos combinacionales clásicos tenemos los siguientes:

- Lógicos (Generador/detector de paridad, multiplexor y demultiplexor, codificador y decodificador, conversor de código, comparador).

- Aritméticos (sumador, restador).

- Combinación de ambos (Unidad aritmético lógica).

Sistemas Secuenciales: en este tipo de sistemas, la salida depende de la entrada actual y de las anteriores. Este tipo de sistemas necesitan elementos de memoria que almacenan la información pasada del sistema. El sistema secuencial más simple es el biestable (dos estados posibles), de los cuales el **tipo D** es el más utilizado.

En éste tipo de circuitos entra a jugar un papel importante el **factor tiempo**, el cual no se había considerado en los circuitos combinacionales. Según como se maneje el factor tiempo en los circuitos

secuenciales, estos se pueden clasificar en ***circuitos secuenciales síncronos*** y ***circuitos secuenciales asíncronos*.**

En los **circuitos secuenciales asíncronos los cambios de estados ocurren al ritmo natural** impuesto por las propias compuertas lógicas utilizadas en su implementación, lo que produce retardos en cascadas entre los diferentes elementos del circuito, lo cual puede ocasionar algunos problemas de funcionamiento ya que estos retardos no están bajo el control del diseñador y, además, no son idénticos para cada compuerta lógica utilizada.

Los **circuitos secuenciales síncronos sólo permiten un cambio de estado en los instantes marcados o autorizados** por una señal de **sincronismo de tipo oscilatorio denominada reloj** (un cristal o un circuito capaz de producir una serie de pulsos regulares en el tiempo), solucionando así los problemas que tienen los circuitos asíncronos debido a los cambios de estado no uniformes dentro del sistema o circuito.

Tema 2.
Representación de la información

A un dígito binario, que puede ser 0 o 1, se le llama **bit**. La información está representada en las computadoras digitales por grupos de bits. Un conjunto de 8 bits se le denomina **byte**, y comprende los números del 0 al 255. Usando diferentes técnicas de codificación, se pueden construir grupos de bits no solamente para representar números binarios sino también otros grupos de símbolos discretos. Los grupos de bits, adecuadamente ordenados, pueden especificar incluso instrucciones para la computadora y datos para procesar. A continuación veremos diferentes códigos de numeración comúnmente utilizados en los sistemas digitales así como la conversión entre ellos.

2.1. Sistemas de códigos y numeración

2.1.1. Representación de los números

El sistema de numeración binario empleado en el diseño de computadores es un sistema de numeración posicional. Mediante dicho sistema se puede representar cualquier número por medio de una cadena de dígitos donde la posición de cada dígito tiene un peso asociado. De esta forma, el valor de un número determinado equivale a la suma ponderada de todos sus dígitos, como por ejemplo:

$$1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$$

En el ejemplo anterior, cada peso constituye una potencia de 10 que es igual a 10^n, donde *n* representa la posición del dígito contando desde la derecha.

Dentro de los sistemas numéricos que se pretenden utilizar se encuentran el sistema decimal, el sistema octal, el sistema binario y el sistema hexadecimal. Cada uno de estos sistemas tiene una base, la cual indica la cantidad de símbolos del sistema. En general, un número de base *r* contiene *r* dígitos, 0, 1, 2,..., *r*-1, y se expresa como una potencia de *r* según la fórmula general:

$$D = \sum_{i-n}^{m-1} A_i r^i$$

Cuando un número se expresa en notación posicional, se escriben solamente los coeficientes y el punto de la base:

$$A_{n-1}A_{n-2} \dots A_1A_0, A_{-1}A_{-2}\, A_{-m+1}A_{-m}$$

En general, se llama al «,» punto de base. A A_{n-1} se le denomina dígito más significativo (MSD, del inglés *most significant digit*) mientras que a A_{-m} dígito menos significativo (LSD, del inglés *less significant digit*) del número. Para distinguir números con bases diferentes, se suele encerrar los coeficientes entre paréntesis y se coloca un subíndice para indicar la base del número en el paréntesis derecho:

$$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

$$(1101)_4 = 1 \times 4^3 + 1 \times 4^2 + 0 \times 4^1 + 1 \times 4^0 = 81$$

$$(1101)_8 = 1 \times 8^3 + 1 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 577$$

$$(1101)_{16} = 1 \times 16^3 + 1 \times 16^2 + 0 \times 16^1 + 1 \times 16^0 = 4353$$

Sistema decimal: de forma general, el sistema numérico decimal es un sistema de base 10 debido a que contiene 10 símbolos correspondientes a los números del 0 al 9. Este sistema es el más usado en la vida cotidiana, ya que a través de éste se puede representar cualquier valor numérico. Al tener 10 dígitos, se pueden formar números de diferentes cantidades. Un número con una sola cifra tiene 10 posibles cantidades que van de 0 a 9, mientras que un número de dos cifras puede tener hasta 100 posibles cantidades.

Entonces la cantidad de un número se obtiene de la siguiente forma:

$$348 = 8 \times 10^0 + 4 \times 10^1 + 3 \times 10^2$$

$$348 = 8 \times 1 + 4 \times 10 + 3 \times 100$$

$$348 = 8 + 40 + 300$$

Por ejemplo, la representación del número decimal 568,25 como suma de valores de cada dígito es:

$$568{,}25 = 5 \times 10^2 + 6 \times 10^1 + 8 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

$$568{,}25 = 5 \times 100 + 6 \times 100 + 8 \times 100 + 2 \times 0{,}1 + 5 \times 0{,}01$$

$$568{,}25 = 500 + 60 + 8 + 0{,}2 + 0{,}05$$

Sistema binario: el sistema de numeración binario es un sistema en base 2 cuyos dígitos son 0 y 1. Un número binario como el 11010 se expresa mediante una cadena de 1s y 0s. El número decimal equivalente a un número binario se puede determinar mediante su expansión en una serie de potencias de base 2. Por ejemplo:

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (26)_{10}$$

Sistema octal y hexadecimal: existen también dos bases adicionales que son importantes en los sistemas digitales. El sistema de numeración octal y el hexadecimal, que usan como base el 8 y el 16. El sistema octal posee ocho valores diferentes y se usan los dígitos del 0 al 7 del sistema decimal. Sin embargo, el sistema hexadecimal representa 16 valores que incluyen los números del 0 al 9 y las letras A, B, C, D, E y F. Como ejemplos de números en ambos sistemas tenemos:

$$(127)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 = (87)_{10}$$

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46687)_{10}$$

En la tabla 1 se presenta la equivalencia de cada símbolo del sistema hexadecimal con respecto a los demás sistemas numéricos.

Decimal (base 10)	**B** (base 2)	**Octal** (base 8)	**Hexadecimal** (base 16)
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabla 1. Equivalencia entre sistemas numéricos. Fuente: elaboración propia.

2.1.2. Conversión entre distintas bases

Para convertir números entre cualquier par de bases se puede utilizar el siguiente método general:

1. Los enteros y las fracciones se convierten de forma separada.
2. La **parte entera** se convierte empleando la división repetitiva por la nueva base y usando la secuencia de residuos generada para crear el nuevo número. La aritmética se lleva a cabo en términos de la base anterior.
3. La **parte fraccionaria** o decimal se convierte mediante la multiplicación repetitiva por la nueva base, empleando los enteros generados para representar la fracción convertida. Igualmente, la aritmética se lleva a cabo en la base anterior.

EJEMPLO 2.1. Convierta el número decimal 278,632 en el numero binario equivalente.

Paso 1. La parte entera es 278 y la parte decimal es 0,632.

Paso 2. Conversión de la parte entera.

División	Residuo generado		
278/2			
139/2	0	Bit menos significativo (LSB).	
69/2	1		
34/2	1		
17/2	0	Lectura hacia **arriba** para formar: →	100010110
8/2	1		
4/2	0		
2/2	0		
1/2	0		
0	1	Bit más significativo (MSB).	

Nótese que una vez formado un residuo, este no toma parte en la aritmética, por tanto, el proceso de conversión de la parte entera siempre tendrá un final.

Paso 3. Conversión de la parte fraccionaria.

Multiplicación	Entero generado		
0,632 × 2 = 1,264	1	Bit más significativo (MSB).	
0,264 × 2 = 0,528	0		
0,528 × 2 = 1,056	1		
0,056 × 2 = 0,112	0		
0,112 × 2 = 0,224	0	Lectura hacia **abajo** para formar: →	,101000011
0,224 × 2 = 0,448	0		
0,448 × 2 = 0,896	0		
0,896 × 2 = 1,792	1		
0,792 × 2 = 1,584	1		

Una vez formado un entero, éste ya no se usa. El proceso puede ser indefinido, por lo que en general, se realiza hasta satisfacer los requerimientos de precisión.

2.1.3. Tipos de códigos

Código BCD: el código decimal más frecuentemente usado es el código de dígitos decimales codificados en binario (*BCD, Binary Coded Decimal*), que asigna una representación binaria sin signo de 4 bits a cada dígito entre 0 y 9, no usándose las palabras del código entre 1010 y 1111 (del 10 al 15 en decimal). La conversión entre las representaciones BCD y decimal se puede llevar a cabo simplemente sustituyendo 4 dígitos BCD por cada dígito decimal y viceversa. Sin embargo, en la práctica se suelen agrupar dos dígitos BCD en un **byte** de 8 bits, que puede representar cualquier valor comprendido entre 0 y 99. Con base en lo anterior, el código BCD no es equivalente a código base 2. El código BCD se representa en la siguiente tabla:

Decimal	BCD	2421	Exceso-3	Biquinario
0	0000	0000	0011	0100001
1	0001	0001	0100	0100010
2	0010	0010	0101	0100100
3	0011	0011	0110	0101000
4	0100	0100	0111	0110000
5	0101	1011	1000	1000001
6	0110	1100	1001	1000010
7	0111	1101	1010	1000100
8	1000	1110	1011	1001000
9	1001	1111	1100	1010000

Tabla 2. Código BCD y otros códigos comunes. Fuente: elaboración propia.

Si se desea representar un valor decimal de más de un dígito en BCD, se debe tomar el código BCD de cada dígito y colocarlo en el peso correspondiente. Por ejemplo, si queremos representar el número decimal 45 en BCD, nos quedaría 0100 0101, donde los primeros 4 dígitos 0100 equivalen al 4, y los últimos 4 dígitos 0101 equivalen al 5.

Los pesos del código BCD son 8, 4, 2 y 1, respectivamente, y por esta razón el código se denomina a veces **código 8421**. Alternativamente, los pesos 2, 4, 2 y 1 se usan para generar el **código 2421** que se muestran en la tabla 2. Este código literal se denomina código autocomplementado, ya que para todo dígito entre 0 y 9, se puede obtener la palabra código del complemento a 9 complementando cada bit de la palabra código del dígito.

También se muestra otro código autocomplementado en la tabla 2, denominado **código de 3 en exceso**. Este no es un código ponderado pero se genera a partir del código BCD sumando 0011 (3 en binario) a cada palabra del código. Los códigos decimales pueden tener también más de 4 bits. Un ejemplo de ello es el **código biquinario** mostrado en la tabla 2 y que utiliza 7 bits. En este código, el primer bit de una palabra del código se emplea para indicar si el dígito decimal se encuentra entre 5 y 9, mientras que el segundo bit indica si se encuentra en el rango de 0 a 4. Los últimos 5 bits de cada palabra código se usan para seleccionar uno de cinco números dentro del rango, donde cada bit corresponde a un número del rango.

Código Gray: el código Gray es un código con una característica fundamental. La variación entre una combinación y otra es de solo un bit. Esta es una característica muy importante en muchas aplicaciones como detección y corrección de errores en sistemas de comunicaciones digitales, de allí la importancia del mismo.

Para construir el código se debe iniciar con los símbolos del código binario en el bit de menor peso, es decir con el 0 y 1. Después se refleja el código resultante, es decir, que se continúa el código colocando el 1 y 0, luego se coloca el bit del siguiente peso con la mitad de las posiciones en 0 y la otra mitad en 1.

00
01
11
10

El **código Gray** de tres dígitos se forma empleando el código de dos dígitos como la base de reflexión y agregando 0s arriba y 1s debajo nuevamente.

000
001
011
010
110
111
101
100

Finalmente se obtiene el **código Gray** (tabla 3) de 4 bits.

Binario	Gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Tabla 3. Código Gray. Fuente: elaboración propia.

Tema 3.
Principios del diseño digital

3.1. Compuertas lógicas

Las **compuertas lógicas** son elementos de electrónica digital que permiten realizar operaciones lógicas entre cantidades binarias. El **álgebra de Boole,** que veremos en el siguiente apartado**,** es el área matemática específica que estudia las operaciones lógicas y sus propiedades.

3.1.1. Compuerta lógica NOT

La compuerta NOT, conocida también como negador o inversor, es una compuerta que permite realizar la operación lógica NOT. Esta operación lógica produce, dada una entrada binaria, el valor contrario a ésta. Es decir, si a la compuerta NOT entra un 0 lógico, su salida será un 1 lógico y viceversa (figura 4).

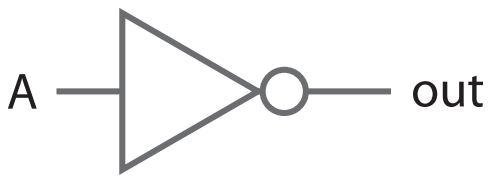


Figura 4. Símbolo de la compuerta NOT. Fuente: pulse aquí para ver la fuente.

Cuando hablamos de operaciones lógicas, nos referimos a niveles de verdad. Un 0 lógico equivale a **FALSO** y un 1 lógico equivale a **VERDADERO**. Por tanto, los resultados de un circuito lógico están determinados por un concepto denominado "**tabla de verdad**", la cual define un valor lógico del circuito de salida para una determinada combinación de entrada.

La **tabla de verdad** de la operación lógica **NOT** es la siguiente:

Entrada	Salida
A	**X**
0	1
1	0

Tabla 4. Tabla de verdad de la compuerta NOT. Fuente: elaboración propia.

En el álgebra de Boole, la compuerta NOT también tiene su propia expresión. Sea *A* la entrada de la compuerta NOT y *X* la salida, luego la expresión Booleana de la operación lógica NOT vendrá dada por:

$$X = \bar{A}$$

3.1.2. Compuerta lógica AND

La compuerta AND, es una compuerta que permite realizar la operación lógica AND. Esta operación tiene siempre como mínimo dos entradas. La tabla de verdad de la operación lógica AND para dos entradas es la siguiente:

Entradas		Salida
A	**B**	**X**
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 5. Tabla de verdad de la compuerta AND de dos entradas. Fuente: elaboración propia.

Para el caso en que la compuerta tiene 3 entradas, la **tabla de verdad** de la operación lógica quedaría:

Entradas			Salida
A	**B**	**C**	**X**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0

Entradas			Salida
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Tabla 6. Tabla de verdad de la compuerta AND de tres entradas. Fuente: elaboración propia.

En el álgebra de Boole, la compuerta AND viene dada por la expresión:

$$X = A \cdot B,$$

Mientras que su símbolo es (figura 5):

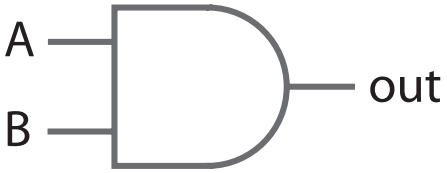


Figura 5. Símbolo de la compuerta AND. Fuente: pulse aquí para ver la fuente.

Como puede apreciarse en las tablas anteriores, la característica fundamental de la operación lógica AND es que su salida será un 1 lógico si y solo si todas sus entradas son 1 lógico.

3.1.3. Compuerta lógica OR

La compuerta OR es una compuerta que permite realizar la **operación lógica OR**. Al igual que la compuerta AND, la operación lógica OR tiene siempre como mínimo dos entradas. La **tabla de verdad** de la operación lógica OR para dos entradas sería:

Entradas		Salida
A	**B**	**X**
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 7. Tabla de verdad de la compuerta OR de dos entradas. Fuente: elaboración propia.

Tanto para dos como para más de dos entradas, la característica fundamental de la operación lógica OR es que su salida será un 0 lógico sí y sólo sí todas sus entradas son 0 lógico.

El símbolo de la compuerta OR se representa en la figura 6:

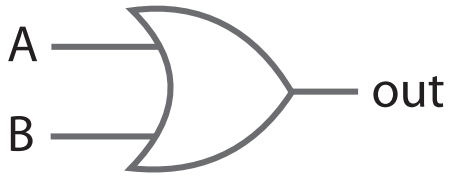


Figura 6. Símbolo de la compuerta OR. Fuente: pulse aquí para ver la fuente.

En el álgebra de Boole, la expresión de la compuerta OR es:

$$X = A + B$$

3.1.4. Compuerta lógica NAND

La compuerta NAND permite realizar la **operación lógica NAND**. Esta operación lógica tiene también como mínimo dos entradas. La **tabla de verdad** de la operación lógica NAND con dos y tres entradas sería:

Entradas		Salida
A	**B**	**X**
0	0	1
0	1	1
1	0	1
1	1	0

Tabla 8. Tabla de verdad de la compuerta NAND de dos entradas. Fuente: elaboración propia.

Entradas			Salida
A	**B**	**C**	**X**
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 9. Tabla de verdad de la compuerta NAND de tres entradas. Fuente: elaboración propia.

Basado en las tablas anteriores, se puede apreciar que la característica fundamental de la operación lógica NAND es que la salida será 0 lógico sí y solo sí todas sus entradas son 1 lógico.

El símbolo de la compuerta NAND es:

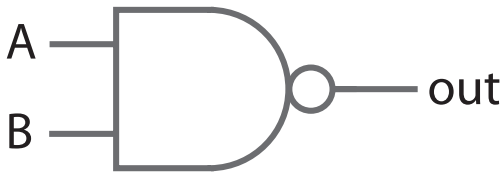


Figura 7. Símbolo de la compuerta NAND. Fuente: pulse aquí para ver la fuente.

En el álgebra de Boole, la expresión de la compuerta NAND es:

$$X = \overline{A \cdot B}$$

3.1.5. Compuerta lógica NOR

La compuerta NOR permite realizar la **operación lógica NOR**. Al igual que las compuertas anteriores excepto la NOT, la operación lógica NOR tiene siempre como mínimo dos entradas. La **tabla de verdad** de la operación lógica OR para dos entradas es la siguiente:

Entradas		Salida
A	**B**	**X**
0	0	1
0	1	0
1	0	0
1	1	0

Tabla 10. Tabla de verdad de la compuerta NOR de dos entradas. Fuente: elaboración propia.

Entradas			Salida
A	**B**	**C**	**X**
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Tabla 11. Tabla de verdad de la compuerta NOR de tres entradas. Fuente: elaboración propia.

Basado en las tablas de la verdad anteriores, se puede apreciar que la característica fundamental de la operación lógica NOR es que la salida será 1 lógico si y solo si todas sus entradas son cero lógico. El símbolo de la compuerta NOR se muestra en la siguiente figura:

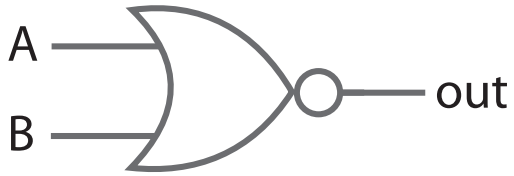


Figura 8. Símbolo de la compuerta NOR. Fuente: pulse aquí para ver la fuente.

La expresión de la compuerta NOR en el álgebra de Boole es:

$$X = \overline{A + B}$$

3.1.6. Compuerta OR exclusiva XOR

La compuerta **OR Exclusiva** o también conocida como **XOR**, es una compuerta que permite realizar la operación lógica XOR. Esta operación siempre tiene como mínimo dos entradas. Su **tabla de verdad** para el caso de dos y tres entradas es:

Entradas		Salida
A	**B**	**X**
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 12. Tabla de verdad de la compuerta XOR de dos entradas. Fuente: elaboración propia.

Entradas			Salida
A	**B**	**C**	**X**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabla 13. Tabla de verdad de la compuerta XOR de tres entradas. Fuente: elaboración propia.

Las tablas anteriores nos indican que la característica fundamental de la operación lógica XOR es que la salida será 1 lógico sí y solo sí la suma de 1s lógicos a la entrada es impar. El símbolo correspondiente a la compuerta XOR se muestra en la siguiente figura.

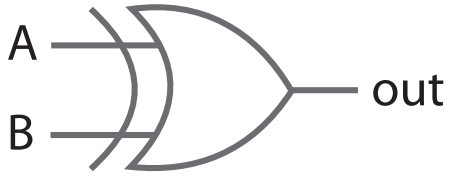


Figura 9. Símbolo de la compuerta XOR. Fuente: pulse aquí para ver la fuente.

Y su expresión en el álgebra de Boole es:

$$X = A \oplus B$$

3.1.7. Compuerta NOR exclusiva XNOR

La compuerta **NOR Exclusiva** o también conocida como **XNOR**, es una compuerta que permite realizar la operación lógica XNOR. Esta operación siempre tiene como mínimo dos entradas. Su **tabla de verdad** para el caso de dos y tres entradas es:

Entradas		Salida
A	**B**	**X**
0	0	1
0	1	0
1	0	0
1	1	1

Tabla 14. Tabla de verdad de la compuerta XNOR de dos entradas. Fuente: elaboración propia.

Entradas			Salida
A	**B**	**C**	**X**
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 15. Tabla de verdad de la compuerta XNOR de tres entradas. Fuente: elaboración propia.

Las tablas anteriores nos indican que la característica fundamental de la operación lógica XNOR es que la salida será 0 lógico sí y solo sí la suma de 1s lógicos a la entrada es impar. El símbolo correspondiente a la compuerta XNOR se muestra en la siguiente figura.

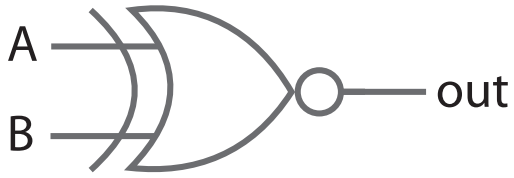


Figura 10. Símbolo de la compuerta XNOR. Fuente: pulse aquí para ver la fuente.

Y su expresión en el álgebra de Boole es:

$$X = \overline{A \oplus B}$$

3.2. Álgebra de Boole

George Boole publicó en 1854 la obra titulada "Investigación de las leyes del pensamiento", basada en las teorías matemáticas de la lógica y la probabilidad. En esta obra formuló la idea del álgebra de operaciones lógicas conocida hoy en día como Álgebra de Boole. En 1938, Shannon aplicó este álgebra para demostrar que las propiedades de los circuitos de conmutación eléctricos se podían representar con un álgebra booleana bivaluada que se denominó álgebra de conmutación. La definición formal de álgebra booleana que se da a continuación fue formulada en 1904 por Huntington.

"En el nivel más amplio, el álgebra booleana es una estructura algebraica definida sobre un conjunto de elementos, B = {0,1}, con dos operadores binarios, OR (+) y AND (•), que satisfacen determinada propiedades".

$$A + B = B + A$$

3.2.1. Leyes del álgebra de Boole

En el álgebra de Boole existen una serie de postulados y leyes que resuelven determinados problemas en las implementaciones de circuitos lógicos. Estos son:

Ley conmutativa:

a) La ley conmutativa indica que el orden de las variables no altera el resultado. Entonces la ley conmutativa para la adición (**operador OR**) de dos variables se describe como:

$$A + B = B + A$$

b) Esta misma ley se puede describir para la multiplicación (**operador AND**) de dos variables como:

$$A \cdot B = B \cdot A$$

Ley asociativa:

a) La ley asociativa para la adición (**operador OR**) de tres variables se describe de la siguiente forma:

$$A + (B + C) = (A + B) + C$$

b) La ley asociativa para la multiplicación (**operador AND**) de tres variables se describe como:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

Ley distributiva:

a) La ley distributiva para tres variables se describe de la siguiente forma respecto al **operador OR:**

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

b) La ley distributiva para tres variables se describe de la siguiente forma respecto al **operador AND**:

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

Postulado del cierre o clausura:

a) Si A y B son dos elementos del álgebra **de Boole**, entonces, el resultado obtenido de aplicar el **operador OR** a ambos elementos también pertenece al álgebra.

$$\text{Si } \{A, B\} \in Z \text{ entonces } A + B \in Z$$

b) Si A y B son dos elementos del álgebra de Boole, entonces, el resultado obtenido de aplicar el **operador AND** a ambos elementos también pertenece al álgebra.

$$\text{Si } \{A, B\} \in Z \text{ entonces } A \cdot B \in Z$$

Postulado de los elementos de identidad:

a) Un elemento de identidad con respecto al **operador OR** es designado por el símbolo 0 y cumple:

$$A + 0 = 0 + A = A \text{, donde } A \in Z$$

b) Un elemento de identidad con respecto al **operador AND** es designado por el símbolo 1 y cumple:

$$A \cdot 1 = 1 \cdot A = A \text{, donde } A \in Z$$

Postulado del complemento.

Para cada elemento $A \in Z$, existe un elemento $\bar{A} \in Z$ (llamado complemento de A) tal que:

a) $A + A = 1$

b) $A \cdot A = 0$

En este álgebra, el significado de los **operadores AND y OR** son distintos de la aritmética clásica. Llama la atención el hecho de que aparezca la propiedad distributiva del operador OR sobre el AND. También, en los postulados no aparecen los inversos de los operadores **AND y OR** y, sin embargo, se dispone de un operador nuevo como es el complemento. Los postulados del álgebra han sido listados a pares, parte (a) y parte (b). Una parte puede obtenerse a partir de la otra mediante el intercambio de los elementos unitarios (0 y 1) y los operadores binarios (**AND y OR**). Esto se conoce como el **Principio de Dualidad**.

Además de los seis postulados anteriormente expuestos, el álgebra **de conmutación** viene acompañado de siete teoremas que se detallan a continuación. Estos teoremas, que se definen a partir de los postulados, heredan el **principio de dualidad** de estos y, por tanto, se listan en pares (a) y (b).

Teoremas del álgebra de conmutación.

1. **Teorema de idempotencia.**

   a) $A + A = A$

   b) $A \cdot A = A$

2. **Teorema de los elementos dominantes.**

   a) $A + 1 = 1$

   b) $A \cdot 0 = 0$

3. **Teorema involutivo.**

   $\overline{(\bar{A})} = A$

4. **Teorema de absorción.**

   a) $A + (A \cdot B) = A$

   b) $A \cdot (A + B) = A$

5. **Teorema de consenso.**

   a) $A + (\bar{A} \cdot B) = A + B$

   b) $A \cdot (\bar{A} + B) = A \cdot B$

6. **Teorema asociativo.**

   a) $A + (B + C) = (A + B) + C$

   b) $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

7. **Leyes de Morgan.**

   a) $\overline{A + B} = \bar{A} \cdot \bar{B}$

   b) $\overline{A \cdot B} = \bar{A} + \bar{B}$

8. **Leyes de Morgan generalizadas.**

   a) $\overline{A + B + C \ldots} = A \cdot B \cdot C \cdot \ldots$

   b) $\overline{A \cdot B \cdot C \ldots} = \bar{A} + \bar{B} + \bar{C} + \ldots$

Las demostraciones de los teoremas anteriores se pueden realizar de manera **algebraica,** donde se trata de obtener unos de los miembros de la igualdad del enunciado del teorema, aplicando los postulados y leyes listados previamente al otro miembro de la igualdad. La demostración de los teoremas del **álgebra de conmutación** se deja al lector como ejercicio. Para ello pueden consultar los libros de textos siguientes: (Mano & Kime, 2005), (Molina, Díaz, & Escudero, 2004) y (Flórez, 2010).

3.2.2. Simplificación de funciones booleanas

El Álgebra de Boole es un instrumento muy útil para la simplificación circuitos digitales. Considere el siguiente ejemplo dada la función booleana representada por:

$$F = \bar{A}BC + \bar{A}B\bar{C} + AC$$

La implementación de ésta ecuación con puertas lógicas se muestra en la figura 11 (a). A las variables de entrada *A* y *C* se le han realizado el complemento con **Inversores** para obtener *A*1 y *C*1. Los tres términos de la expresión se realizan con tres puertas **AND**. La puerta **OR** forma la **OR lógica** de tres de los términos. Considere ahora una simplificación de la expresión *F* aplicando algunas de los postulados y teoremas anteriormente expuestos:

$$\begin{aligned} F &= \bar{A}BC + \bar{A}B\bar{C} + AC \\ &= \bar{A}B(C + \bar{C}) + AC \\ &= \bar{A}B \cdot 1 + AC \\ F &= \bar{A}B + AC \end{aligned}$$

Como puede observarse, la expresión inicial se reduce a sólo dos términos y puede ser realizada con compuertas lógicas según se muestra en la figura 11 (b) en la página siguiente. Es evidente que el circuito de (b) es más simple que el de (a), aun cuando ambos realizan la misma función.

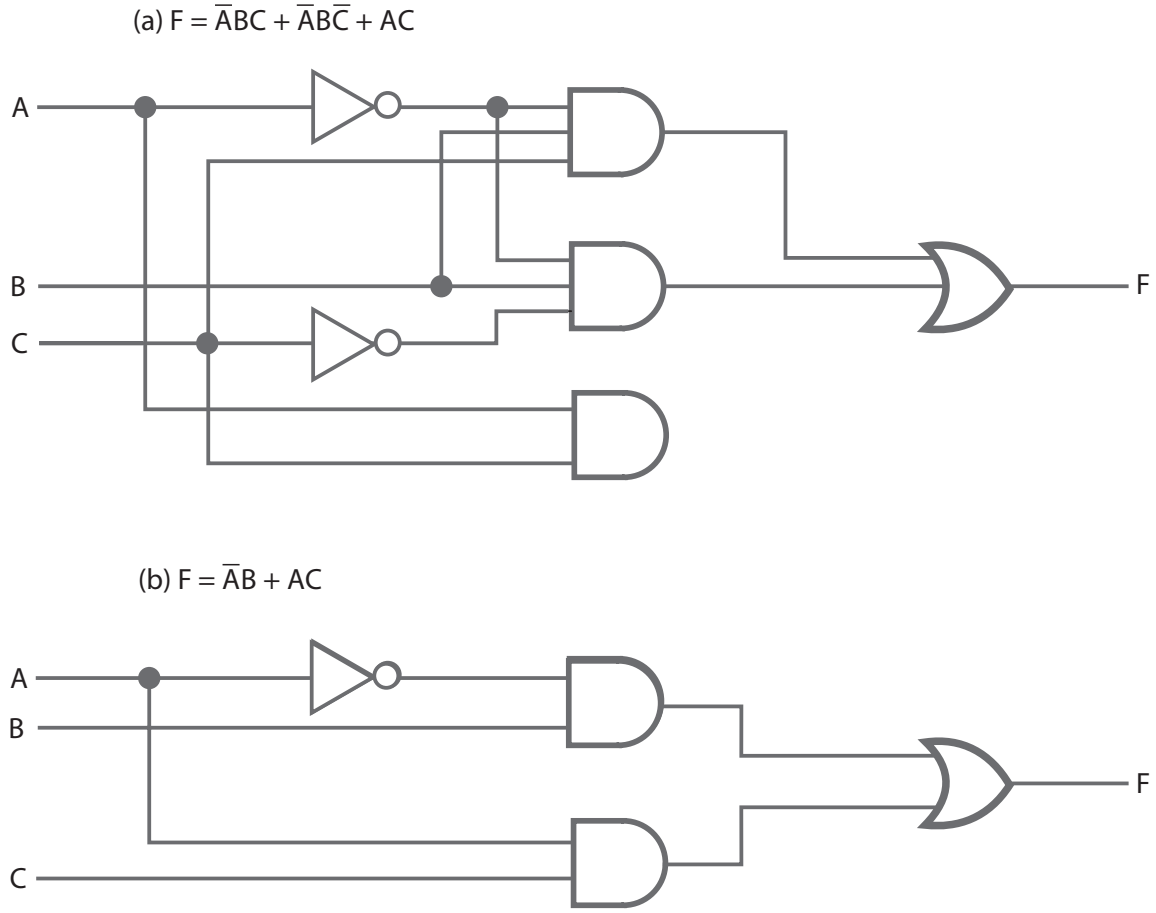


Figura 11. Circuito lógico de la función F (a) Sin simplificar (b) Simplificado. Fuente: Mano, M. & Kime, C. (2005). Implementación de funciones booleanas con puertas [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

Si se implementa una ecuación booleana con compuertas lógicas, cada término requiere una puerta, y cada variable dentro del término indica una entrada para la compuerta. Definimos un ***literal*** como una variable única dentro de un término que puede estar complementado o no. La expresión para la función de la figura 11 (a) tiene tres términos y ocho literales; la de la figura 11 (b) tiene dos términos y cuatro literales. Reduciendo el número de términos, el número de literales, o ambos en una expresión booleana, muchas veces es posible obtener un circuito más sencillo. El **Álgebra de Boole** se aplica para reducir una expresión con el fin de obtener un circuito más simple. Cuando las funciones son muy complejas, es muy difícil encontrar la mejor expresión basada en sumas de términos y literales, aun cuando se empleen programas de computadora. Sin embargo, frecuentemente se incluyen ciertos métodos para reducir expresiones en las herramientas por computadora para sintetizar circuitos lógicos con los que se pueden obtener buenas soluciones.

3.3. Formas canónicas

Una **función booleana** se puede escribir expresada algebraicamente de diferentes maneras. Sin embargo, hay formas concretas de escribir las ecuaciones algebraicas que se consideran como **formas canónicas**. Las formas canónicas facilitan los procedimientos de simplificación para expresiones booleanas y frecuentemente da lugar a circuitos lógicos más deseables. La forma canónica contiene

términos producto y **términos suma**. Un ejemplo de un término producto es $\overline{A}BC$. Esto es un producto lógico formado por una operación AND de tres literales. Un ejemplo de un término suma es $\overline{A} + B + \overline{C}$, que representa una suma lógica formada por una operación OR entre los tres literales. Hay que tener en cuenta de que las palabras **«producto»** y **«suma»** no implican operaciones aritméticas en el **álgebra de Boole;** sino que especifican las operaciones lógicas AND y OR, respectivamente.

3.3.1. Minitérminos y maxitérminos

Se ha mostrado que la tabla de verdad define una función booleana. Una expresión algebraica que represente la función se puede derivar de la tabla buscando la suma lógica de todos los términos producto para los que la función asume el **valor binario 1**. A un término producto donde todas las variables aparecen exactamente una vez, sean complementadas o no Complementadas, se le llama ***minitérmino***. Su principal característica es que representa exactamente una combinación de las variables binarias en la tabla de verdad, teniendo el valor 1 para esta combinación y 0 para el resto. Hay 2^n minitérminos diferentes para *n* variables.

Los ocho minitérminos correspondiente a las tres variables *A*, *B*, y *C* se muestran en la tabla 16. Como puede apreciarse, para cada combinación binaria hay un minitérmino asociado. Cada minitérmino es un producto de exactamente tres literales. Se muestra también un símbolo m_j para cada minitérmino de la tabla, donde el subíndice *j* denota el equivalente decimal de la combinación binaria para la que el minitérmino tiene el valor 1. En la parte derecha de la tabla se muestra además la tabla de verdad para cada minitérmino. Estas tablas de verdad serán útiles al usar minitérminos para formar expresiones booleanas.

Por otra parte, a un **término suma** que contiene todas las variables de forma complementada o no complementada se le llama ***maxitérmino***. Otra vez es posible formular 2^n maxitérminos con *n* variables. Los ocho maxitérminos derivados para tres variables se muestran en la tabla 17 de la página siguiente. Cada maxitérmino representa una suma lógica de tres variables, donde cada variable se complementa si el bit correspondiente del número binario es 1 y no se complementa si es 0. El símbolo para un maxitérmino es M_j, donde *j* denota el equivalente decimal de una combinación binaria para la que el maxitérmino tiene el valor 0.

A	*B*	*C*	Término producto	Símbolo	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{A}\,\overline{B}\,\overline{C}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{A}\,\overline{B}\,C$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{A}\,B\,\overline{C}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{A}\,B\,C$	m_3	0	0	0	1	0	0	0	0
1	0	0	$A\,\overline{B}\,\overline{C}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$A\,\overline{B}\,C$	m_5	0	0	0	0	0	1	0	0
1	1	0	$A\,B\,\overline{C}$	m_6	0	0	0	0	0	0	1	0
1	1	1	$A\,B\,C$	m_7	0	0	0	0	0	0	0	1

Tabla 16. Minitérminos para tres variables. Fuente: elaboración propia.

A	B	C	Término producto	Símbolo	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$A+B+C$	M_0	0	1	1	1	1	1	1	1
0	0	1	$A+B+\overline{C}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$A+\overline{B}+C$	M_2	1	1	0	1	1	1	1	1
0	1	1	$A+\overline{B}+\overline{C}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{A}+B+C$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{A}+B+\overline{C}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{A}+\overline{B}+C$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{A}+\overline{B}+\overline{C}$	M_7	1	1	1	1	1	1	1	0

Tabla 17. Maxitérminos para tres variables. Fuente: elaboración propia.

Ahora está claro de dónde salen los términos «**minitérmino**» y «**maxitérmino**»: un **minitérmino** es una función, no igual a 0, que tiene el menor número de 1s en su tabla de verdad, mientras que un **maxitérmino** es una función, no igual a 1, que tiene el mayor número de 1s en su tabla de verdad. Véase de la tabla 16 y 17 que los minitérminos y maxitérminos con los mismos subíndices son complementos entre sí; o sea, $M_j = m_j$. Por ejemplo, para j=3, tenemos:

$$\overline{m}_3 = \overline{\bar{A}BC} = A + \bar{B} + \bar{C} = M_3$$

Una función booleana puede ser representada algebraicamente por una tabla de verdad dada formando la suma lógica de todos los minitérminos que producen un 1 en el valor de la función. Esta expresión se llama una **suma de minitérminos**.

Veamos la siguiente función booleana *F* de la tabla 18. La función toma valor 1 para las combinaciones binarias 000, 010, 101 y 111, de las variables A, B, y C. Esas combinaciones corresponden a los minitérminos j = 0, 2, 5 y 7. Si observamos la tabla 18 y las tablas de verdad para éstos minitérminos de la tabla 16, es evidente que la función *F* puede expresarse algebraicamente como la suma lógica de los minitérminos:

$$F = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}C\ ABC = m_0 + m_2 + m_5 + m_7$$

X	Y	Z	F	$\overline{F}$
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Tabla 18. Tabla de verdad de la función booleana F. Fuente: elaboración propia.

La expresión anterior se puede abreviar aún *más* si enumeramos los subíndices decimales de los minitérminos dentro de una suma lógica (OR booleana), empleando el símbolo Σ y colocando entre paréntesis después de *F* la lista de las variables involucradas, en el mismo orden de conversión de los minitérminos:

$$F(A,B,C) = \Sigma m(0,2,5,7)$$

Si ahora consideramos el complemento de la función anterior $\overline{F}$, los valores binarios de $\overline{F}$ de la tabla 18 se obtendrán intercambiando los 1s por 0s y viceversa en los valores de F. Por tanto, partiendo de la suma lógica de los minitérminos de $\overline{F}$, tendríamos:

$$\overline{F} = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C} = m_1 + m_3 + m_4 + m_6$$

siendo la forma abreviada,

$$\overline{F}(A,B,C) = \Sigma m(1,3,4,6)$$

Nótese que la lista de números asociados a los minitérminos de la función $\overline{F}$ son los que no están en la lista de números de los minitérminos de la función $\overline{F}$.

Si ahora quisiéramos expresar la función booleana F como **producto de maxitérminos**, entonces tomaremos el complemento de $\overline{F}$ para obtener F:

$$F = \overline{m_1 + m_3 + m_4 + m_6} = \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_6}$$

$$= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \text{ debido a que } \overline{m_j} = M_j$$

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

cuya forma abreviada seria:

$$F(A,B,C) = \prod \mathrm{M}(1,3,4,6)$$

y donde el símbolo $\prod$ denota el **producto lógico** (AND booleana) de los maxitérminos cuyos números se listan entre paréntesis. Nótese que los números decimales incluidos en la lista de maxitérminos son los mismos que los de la lista de minitérminos de la función complementada (1, 3, 4, 6) del ejemplo anterior. En general, cuando se trata con funciones booleanas, se suele usar con frecuencia la lista de minitérminos de $\overline{F}$ en vez de los maxitérminos.

3.3.2. Suma de productos

La de suma de minitérminos que se obtiene directamente de una tabla de verdad contiene el máximo número de literales en cada término y, normalmente contiene más productos de los necesarios. Esto se debe a que cada minitérmino incluye todas las variables de la función, complementada o no. Una vez obtenida la suma de minitérminos a partir de la tabla de verdad, el siguiente paso consiste en simplificar la expresión para reducir el número de productos y de literales en los términos. El resultado será una expresión simplificada en la forma de **suma de productos**, que es una forma canónica

alternativa de expresión que contiene productos con uno, dos, o cualquier número de literales. Un ejemplo de una función booleana expresada como suma de productos es la siguiente:

$$F = \bar{B} + \bar{A}B\bar{C} + AB$$

Como puede verse, la expresión contiene tres productos, cada uno con diferente número de literales. El diagrama lógico asociado a esta función estaría formado por un grupo de compuertas AND, seguido de una única compuerta OR. Excepto para el término con un único literal, cada producto requiere una compuerta AND. La suma lógica estaría formada por unas puertas OR cuyas entradas serían literales únicas y las salidas de las puertas AND. Este tipo de configuración para un circuito lógico se le denomina **implementación de dos niveles** o **circuito de dos niveles**.

3.3.3. Producto de sumas

Otra **forma canónica** en la que podemos expresar funciones booleanas algebraicamente es el **producto de sumas**, la cual se forma se obtiene formando un **producto lógico de sumas**. Cada término de la suma lógica puede tener cualquier número de literales diferentes. Como ejemplo de una suma de productos tenemos:

$$F = A(\bar{B} + C)(A + B + \bar{C})$$

La expresión anterior contiene sumas de uno, dos y tres literales. Los términos de suma realizan una operación OR y el producto es una operación AND. La estructura de las puertas de la expresión de productos de suma estaría formada por un grupo de puertas OR para las sumas (excepto para el término con un único literal), seguido de una puerta AND. Similar al caso de **suma de productos**, este tipo de expresión canónica está formada por una **estructura de dos niveles** de puertas.

3.4. Mapas de Karnaugh

El **mapa de Karnaugh** es un diagrama hecho de cuadros, donde cada cuadro está asociado una combinación binaria de entrada o minitérmino y su contenido representa el valor que toma la función para dicha combinación. La estructura del mapa depende del número de variables que definen la función. A continuación veremos cómo se representan los K-mapas de 2, 3 y 4 variables.

Como se ha visto en las secciones anteriores, las expresiones booleanas pueden simplificarse mediante manipulación algebraica aplicando los diferentes postulados y teoremas del **álgebra de Boole**. Sin embargo, este procedimiento de simplificación no permite determinar si se ha conseguido la expresión más sencilla posible. Por otro lado, el método conocido como **Mapa de Karnaugh** o K-mapa provee un procedimiento directo para optimizar funciones booleanas con un máximo de varias variables. El **mapa de Karnaugh** es un diagrama hecho de cuadrados, donde cada cuadrado representa un minitérmino de la función y su contenido representa el valor que toma la función para la correspondiente combinación de entrada. Por tanto, el mapa de Karnaugh presenta un diagrama visual de todos los caminos posibles que pueden tomarse para expresar una función en forma canónica. La estructura del mapa depende del número de variables que definen la función, de manera que a continuación veremos cómo se representan los K-mapas de 2, 3 y 4 variables.

3.4.1. Mapas de dos variables

El mapa de dos variables está formado por cuatro cuadros o celdas organizados en dos filas y dos columnas, uno por cada minitérmino. Los nombres de las variables de entrada se sitúan en la parte superior izquierda del mapa, separadas por una línea diagonal (ver figura 12). Los valores que puede tomar la variable B se sitúan en la cabecera de las columnas del K-mapa, mientras que los asociados a la otra variable, A, se sitúan a la izquierda de las filas del mapa. El nombre de la función representada se sitúa debajo del mapa.

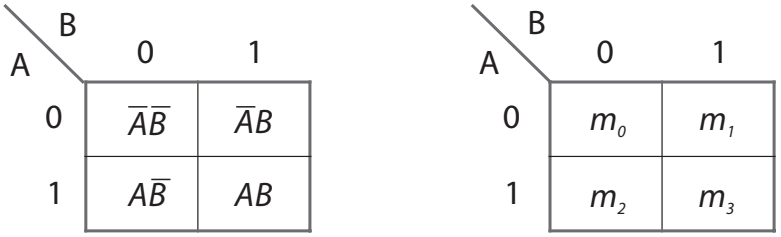


Figura 12. K-mapa de dos variables. Fuente: elaboración propia.

La entrada asociada a una determinada celda del K-mapa está formada por los valores lógicos situados en la cabecera de la columna y fila a las que pertenece dicha celda. Por ejemplo, la celda situada en la parte inferior derecha del K-mapa tiene la entrada asociada AB = (11), el de la celda situada en la parte inferior izquierda AB = (10), la de la parte superior derecha, AB = (01) y la de la parte superior izquierda la AB = (0, 0). En el interior de cada celda del K-mapa se puede representar, en decimal, la entrada asociada al mismo.

Una función de dos variables puede representare en un mapa marcando las celdas que corresponden a los **minitérminos** de la función. En la figura 13 se muestran dos ejemplos para ilustrar el proceso. En el lado izquierdo tenemos la función $f = AB$, equivalente al **minitérmino** m_3, y donde se pone un 1 dentro de la celda que pertenece a m_3. En el lado derecho, se muestra el mapa para la función representada por la suma lógica de tres **minitérminos** según:

$$m_1 + m_2 + m_3 = \bar{A}B + A\bar{B} + AB$$

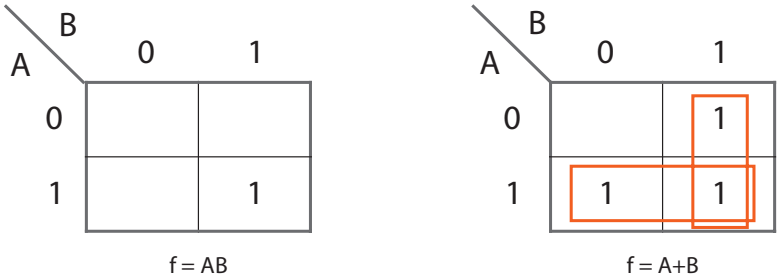


Figura 13. Funciones algebraicas representadas en el mapa de Karnaugh. Fuente: elaboración propia.

La expresión optimizada $A + B$ se determina a partir de los grupos formados por las celdas de la segunda fila para la variable A y por las celdas de la segunda columna para la variable B. Estos dos

grupos juntos incluyen las tres celdas pertenecientes a las variables A o B y si puede justificarse mediante la simplificación algebraica de la siguiente forma:

$$\bar{A}B + A\bar{B} + AB = \bar{A}B + A(\bar{B} + B) = \overline{(A} + A)(B + A) = A + B$$

Más adelante veremos el procedimiento a seguir para combinar celdas en el mapa, el cual iremos aclarando por medio de algunos ejemplos.

3.4.2. Mapas de tres variables

El **mapa de Karnaugh** de 3 variables es una tabla con un número de celdas equivalente a 2^3 celdas, donde 3 es el número de variables del mapa obteniéndose 8 **minitérminos**. En este caso, las variables A, B y C se utilizan para denominar a las variables. Los valores binarios de A se especifican en la parte lateral izquierda mientras que los valores de B y C en parte superior de la tabla. Nótese que los números binarios situado en las columnas no siguen la secuencia de la cuenta binaria. Esto es porque en la secuencia enumerada solamente cambia su valor un bit de una columna hacia la adyacente, lo cual corresponde al **Código Gray** introducido anteriormente. En el esquema de la derecha de la figura 14 se han señalado cuales son las variables que cambian de una celda a otra, así como se han añadido también en cada celda el número decimal asociado a cada **minitérmino**.

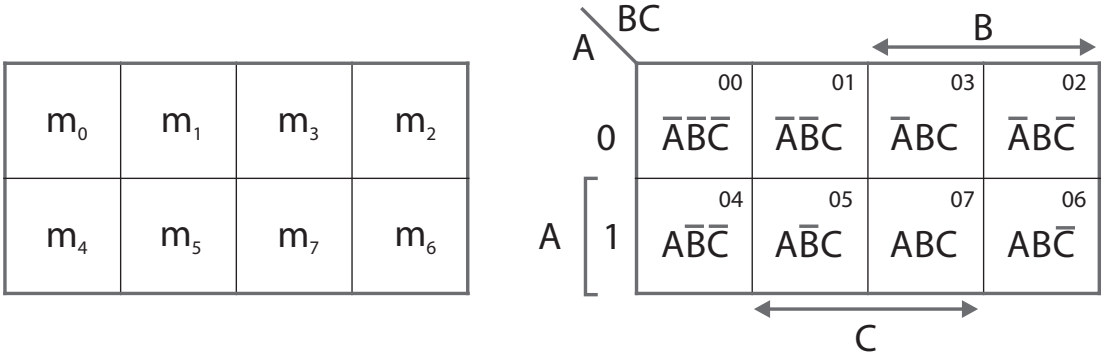


Figura 14. K-mapa de tres variables. Fuente: elaboración propia.

El valor asociado a cada celda corresponde a la combinación definida en el mapa de Karnaugh anterior. En la primera columna, se puede apreciar que las variables B y C son negadas, debido a que sus respectivos valores en la parte superior son 00. Este mismo concepto aplica para todas las celdas del mapa.

Veamos los siguientes ejemplos de funciones con 3 variables y su simplificación usando el mapa de Karnaugh:

$$F_1(A,B,C) = \sum m(3,4,6,7)$$

$$F_2(A,B,C) = \sum m(0,2,4,5,6)$$

El mapa de la función F_1 se muestra en la figura 15 (a). Hay cuatro celdas marcadas con 1s, uno para cada **minitérmino** de la función. Se agrupan las celdas adyacentes en la tercera columna para llegar

al término de dos literales BC. Las dos celdas sobrantes, también con 1s, son adyacentes por la definición basada en el cilindro, y se muestran en el diagrama con sus valores incluidos en medios rectángulos. Cuando se combinan, estos dos cuadrados construyen el término de dos literales $A\,\overline{C}$, de manera que la función optimizada pasa a ser:

$$F_1 = BC + A\bar{C}$$

El mapa para la función F_2 se muestra en la figura 15 (b). Primeramente se combinan las celdas adyacentes en las primeras y últimas columnas, basándonos en el cilindro, para llegar al término de un literal $\overline{C}$. Luego la última celda sobrante que representa el **minitérmino** 5, se combina con la celda adyacente usada anteriormente. Esta opción no solo es posible, sino que es deseable, ya que las dos celdas adyacentes conducen al término de dos literales $A\,\overline{B}$, con la celda simple representando al **minitérmino** de tres literales $A\,B\,\overline{C}$, quedando entonces la función optimizada como:

$$F_2 = \bar{C} + A\bar{B}$$

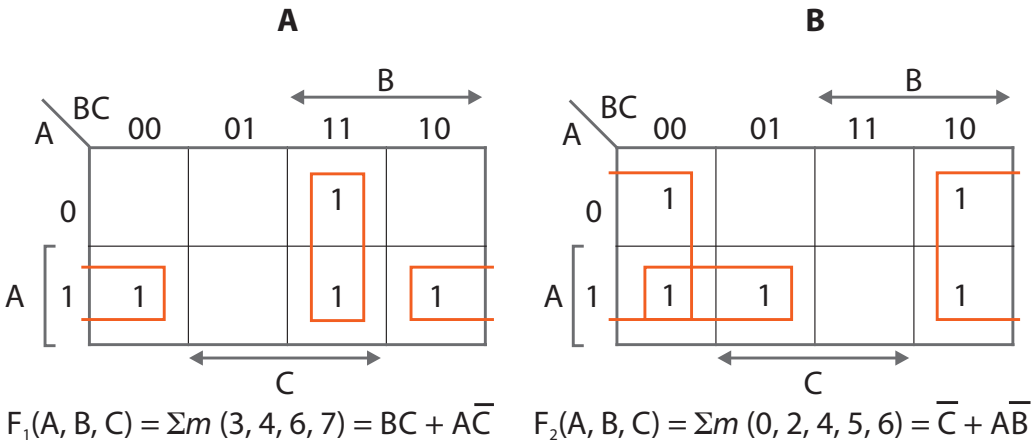


Figura 15. Mapas de ejemplos para tres variables. Fuente: elaboración propia.

3.4.3. Mapas de cuatro variables

En el caso de cuatro variables, el **mapa de Karnaugh** es una tabla con un número de celdas equivalente a 2^4 (16) celdas, siendo 4 el número de variables. En este caso, las letras A, B, C y D se utilizan para denominar a las variables. Los valores binarios de A y B se especifican en la parte lateral izquierda mientras que los valores de C y D en parte superior de la tabla.

El valor asociado a cada celda corresponde a la combinación especificada en el mapa de Karnaugh de la figura 16 (b). En la primera fila A y B son negados ya que el valor de A y B en la parte de la izquierda son 00. En la primera columna son negados C y D, ya que sus valores en la parte superior son 00. Este concepto se aplica para todas las celdas del mapa. Nótese además que al igual que en el mapa de tres variables, donde las columnas no siguen la secuencia binaria, la secuencia de las filas en el mapa de 4 variables también se rige por el **código de Gray**, logrando de esta forma que solo cambie una sola variable entre las celdas adyacentes de las diferentes filas.

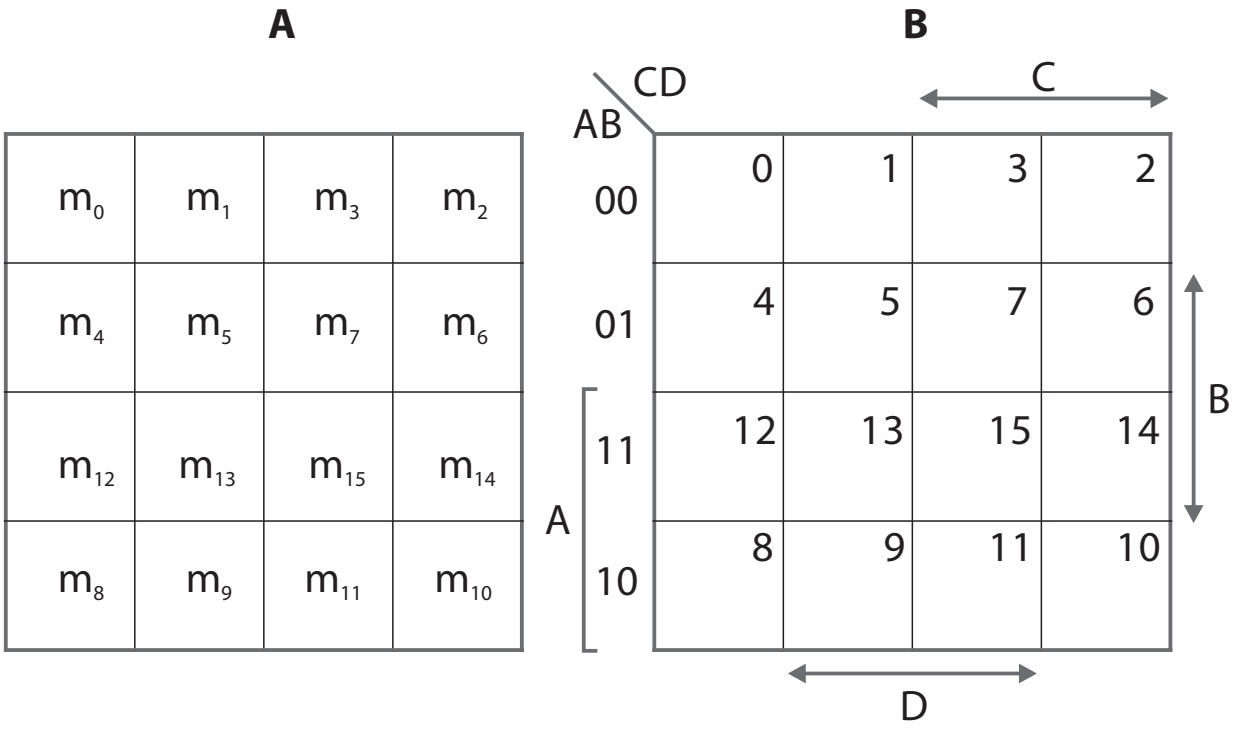


Figura 16. K-mapa de cuatro variables. Fuente: elaboración propia.

Veamos un ejemplo donde se muestre el procedimiento de simplificar una función booleana de 4 variables:

$$F(A,B,C,D) = \sum m\,(0,1,2,4,5,6,8,9,12,13,14)$$

Los **minitérminos** de la función anterior se han marcado con 1 en el mapa de la figura 17. Las 8 celdas de las dos columnas de la izquierda se combinan para formar un grupo con el término literal único, $\overline{C}$.

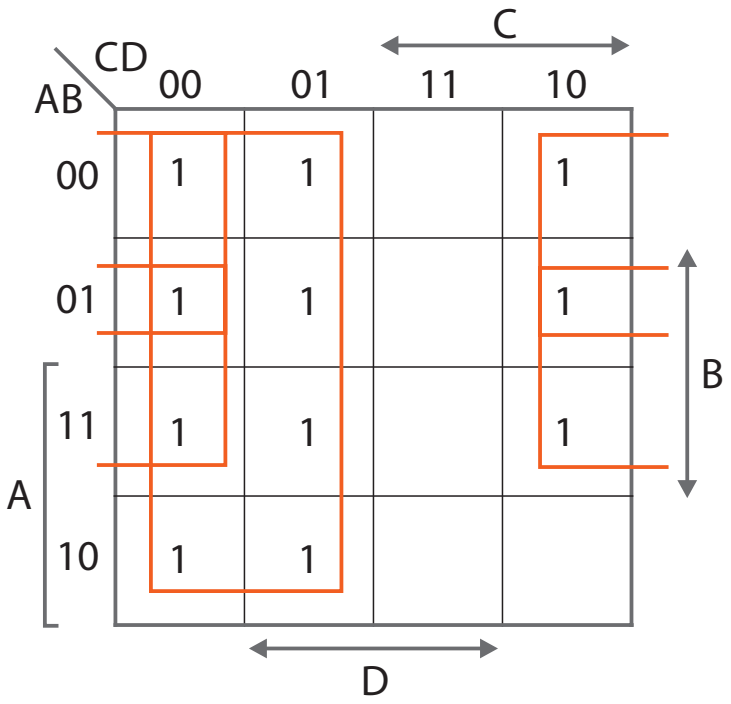


Figura 17. Ejemplo de simplificación de una función de 4 variables usando el mapa de Karnaugh. Fuente: elaboración propia.

Los tres 1s restantes no pueden combinarse dentro un solo grupo, por lo que debe hacerse formando grupos de 2 o 4 celdas. Los dos 1s superiores de la derecha se combinan con los dos 1s superiores de la izquierda dando lugar al término $\overline{A}\,\overline{D}$. Finalmente, la celda marcada con 1 en la tercera fila y cuarta columna (minitérmino 1110), en vez de tomarse sola, se combina con otras celdas ya usadas para formar un grupo de cuatro celdas incluyendo las dos filas intermedias y las dos columnas finales, resultando en el término $B\,\overline{D}$. La expresión optimizada sería entonces la suma lógica de los tres términos:

$$F = \bar{C} + \overline{A}\,\overline{D} + B\overline{D}$$

3.4.4. Simplificación de funciones haciendo uso de los mapas de Karnaugh

Después de ver varios ejemplos, podemos resumir, siguiendo una serie de pasos, como simplificar cualquier función usando los mapas de Karnaugh. A continuación, veremos cuáles son estos pasos:

- Se colocan 1s en las celdas correspondientes a las combinaciones de entrada donde la expresión a simplificar toma valor 1 en la salida.
- Se **agrupan** las celdas adyacentes. Las celdas adyacentes son aquellas donde sólo cambia una variable. Los grupos formados deben contener 2, 4, 8 o 16 celdas. Si un mapa no contiene 1s, entonces el resultado de la función es 0. Si por el contrario se agrupan todas las celdas, entonces su resultado es 1. Si un mapa tiene la mitad de celdas en 1 y la otra mitad en 0 y no se puede hacer ninguna agrupación, la solución será la **XOR** o **XNOR** de las entradas.
- Cada grupo formado debe contener el mayor número posible de celdas.
- Se deben agrupar todos los unos contenidos en el mapa, sin importar que una celda que contenga un 1 se agrupe varias veces.
- No se deben agrupar las celdas que contienen 0s.
- Se analiza cada grupo individualmente. Se elimina aquella variable que cambia dentro del grupo. Si una variable se conserva, se retiene su valor para ese **minitérmino**.

3.4.5. Condiciones de indiferencia

En determinadas aplicaciones, una función puede no estar especificada para ciertas combinaciones de entrada. Un ejemplo es el caso del código BCD, el cual es un código de cuatro bits donde sólo se consideran 10 combinaciones, mientras que las seis combinaciones restantes no se usan o no se esperan que ocurran. En este caso, a las celdas cuyas combinaciones no se consideran, se coloca una "X", indicando que no importa que valor tome la salida para tales combinaciones de entrada. En el momento de formar los grupos, se puede considerar estos valores como 1 o 0, según convenga en la solución del mapa. Por esta razón, es muy común llamar a los minitérminos no especificados **condiciones de indiferencia**.

Para entender el procedimiento a la hora de manejar las condiciones de indiferencia, consideraremos la siguiente función F, la cual no está completamente definida, y contiene tres **minitérminos de indiferencia** d:

$$F(A,B,C,D) = \Sigma m(1,3,7,11,15)$$

$$d(A,B,C,D) = \Sigma m(0,2,5)$$

Los minitérminos de F son las combinaciones de variables que igualan la función a 1. Los minitérminos de d son minitérminos de indiferencia. La optimización de la función haciendo uso del mapa se muestra en la figura 18 Los minitérminos de F están marcados con 1, los de d están marcados con X, y las celdas sobrantes se han completado con 0s. Para simplificar la función en forma de suma de productos, debemos incluir los cinco 1s en el mapa, pero podemos incluir o no alguna de las X, dependiendo de cuales producen la expresión más sencilla de la función. El término CD incluye los cuatro minitérminos en la tercera columna. Los minitérminos sobrantes en la celda 0001, se puede combinar con la celda de la derecha (0011) para dar lugar a un término de tres literales. Sin embargo, incluyendo una (la de abajo) o dos (las de los extremos de la primera fila) X adyacentes, podemos combinar cuatro celdas en un grupo obteniéndose un término de dos literales. En la parte (a) de la figura, los minitérminos de indiferencia 0 y 2 están incluidos con los 1s, lo cual da lugar a la función simplificada:

$$F = CD + \bar{A}\bar{B}$$

En la parte (b), el minitérmino de indiferencia 5 está incluido con los 1s, y la función simplificada sería entonces:

$$F = CD + \bar{A}D$$

Las dos expresiones anteriores representan a dos funciones algebraicamente diferentes. Ambas funciones incluyen los minitérminos especificados en la función original incompletamente especificada, pero cada una incluye diferentes minitérminos de indiferencia. Por lo que respecta a la función incompletamente especificada, ambas expresiones son correctas. La única diferencia está en el valor que toma la función F para los minitérminos no especificados.

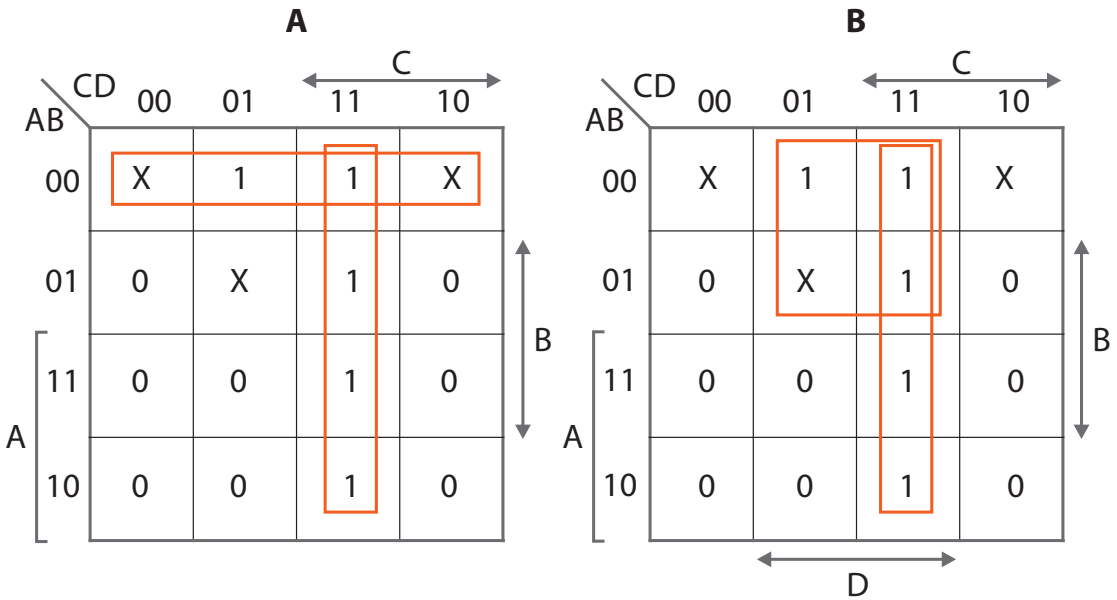


Figura 18. Ejemplo de simplificación de una función de 4 variables usando el mapa de Karnaugh. Fuente: elaboración propia.

Tema 4.
Bloques combinacionales básicos

Los circuitos lógicos utilizados en los sistemas digitales pueden ser de dos tipos, **combinacionales** o **secuenciales**. Un **circuito combinacional** está formado por compuertas lógicas cuyas salidas están determinadas, en cualquier instante del tiempo, por operaciones lógicas realizadas sobre las entradas en ese mismo instante. Un **circuito combinacional** realiza una operación lógica que puede especificarse lógicamente mediante un conjunto de ecuaciones booleanas.

Un circuito combinacional está formado por variables de entrada, de salida, compuertas lógicas e interconexiones. Las compuertas lógicas interconectadas entre si reciben señales procedentes de las entradas y generan señales de salidas. En la figura 19 de la página siguiente se muestra el diagrama de un circuito combinacional típico. Las *n* variables de entrada provienen del entorno del circuito, y las *m* variables de salida están disponibles para ser usadas por dicho entorno. Cada variable de entrada y de salida representa físicamente una señal binaria que toma los valores lógicos 0 y 1.

Figura 19. Diagrama general de un sistema digital combinacional. Fuente: elaboración propia.

4.1. Circuitos aritméticos

4.1.1. Sumador básico.

Los sumadores lógicos son circuitos muy importantes en diferentes tipos de sistemas digitales donde se procesen datos numéricos. Para diseñar un sumador básico, es necesario conocer las reglas de la suma binaria. Simplemente se tienen cuatro posibilidades:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

Por tanto, siguiendo las reglas anteriores se puede construir la siguiente **tabla de verdad.**

Entradas		Salidas	
A	**B**	Σ	C_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabla 19. Tabla de verdad del sumador básico. Fuente: elaboración propia.

A partir de la tabla anterior, podemos entonces obtener las siguientes funciones para cada una de las dos salidas.

$$\Sigma = \bar{A}B + A\bar{B} = A \oplus B$$
$$C_0 = AB$$

El circuito lógico que se obtiene en base a las ecuaciones anteriores quedaría como el de la figura 20 de la página siguiente.

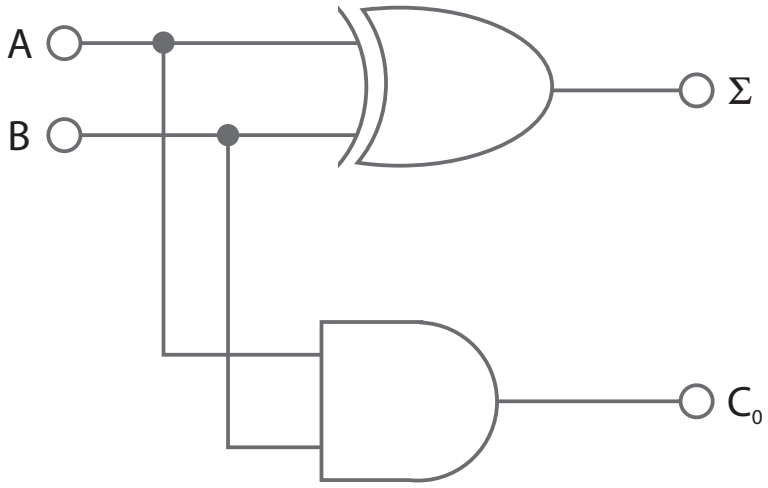


Figura 20. Implementación del sumador básico. Fuente: elaboración propia.

4.1.2. Sumador completo

El sumador completo se implementa a partir de tres entradas de un bit. Estas entradas representan dos variables de un bit (A y B) más una tercera variable (C_{IN}) que representa un acarreo de entrada, proveniente por ejemplo, de la salida C_0 del sumador básico anterior. Teniendo en cuenta lo anterior, la tabla de verdad sería la siguiente.

Entradas			Salidas	
A	**B**	**C_{IN}**	**Σ**	**C_0**
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

Tabla 20. Tabla de verdad del sumador completo. Fuente: elaboración propia.

Las expresiones Booleanas correspondientes a las salidas de la tabla de verdad serían las siguientes:

$$\begin{aligned}\Sigma &= \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \\ &= \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC) \\ &= \bar{A}(B \oplus C) + A(\overline{B \oplus C}) \\ &= A \oplus (B \oplus C)\end{aligned}$$

$$C_0 = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$
$$= C(\bar{A}B + A\bar{B} + AB(\bar{C} + C)$$
$$= C(A \oplus B) + AB$$

Basándonos en las ecuaciones anteriores, se obtiene entonces la siguiente implementación:

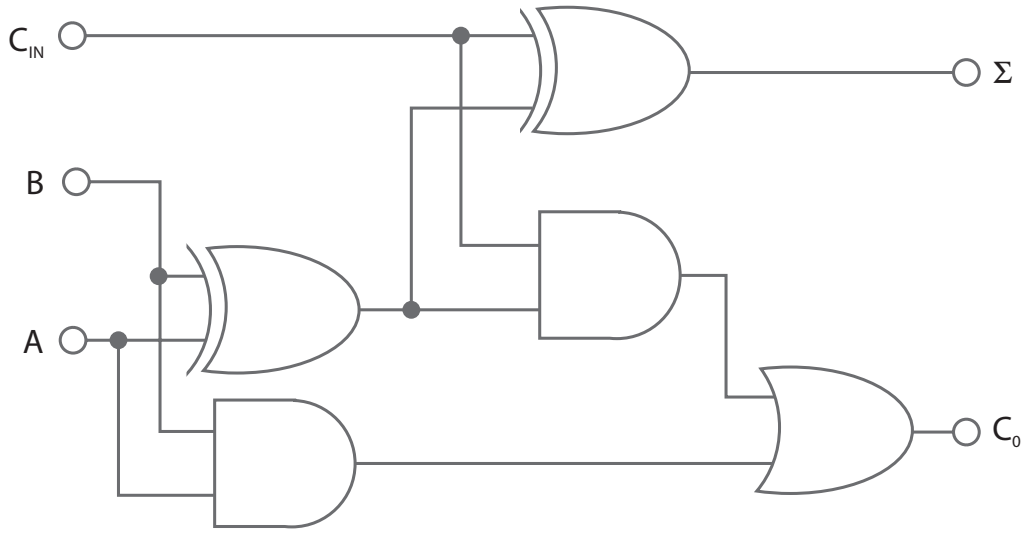


Figura 21. Implementación del sumador completo. Fuente: elaboración propia.

4.1.3. Sumador con acarreo en serie

Para implementar un sumador binario paralelo de dos números de 4 bits, se deben emplear 4 sumadores completos. La figura 22 muestra la interconexión de cuatro sumadores completos para formar un sumador de 4 bits con acarreo serie. Los bits de los sumando *A* y *B* son designados mediante subíndices en orden creciente de derecha a izquierda, donde el subíndice 0 denota el bit de menor peso o menos significativo. La salida de acarreo de cada sumador se conecta a la entrada de acarreo del siguiente sumador de orden superior. Estos acarreos se denominan **acarreos internos**. El acarreo de entrada del sumador paralelo es *C*0, y el acarreo de salida es *C*4.

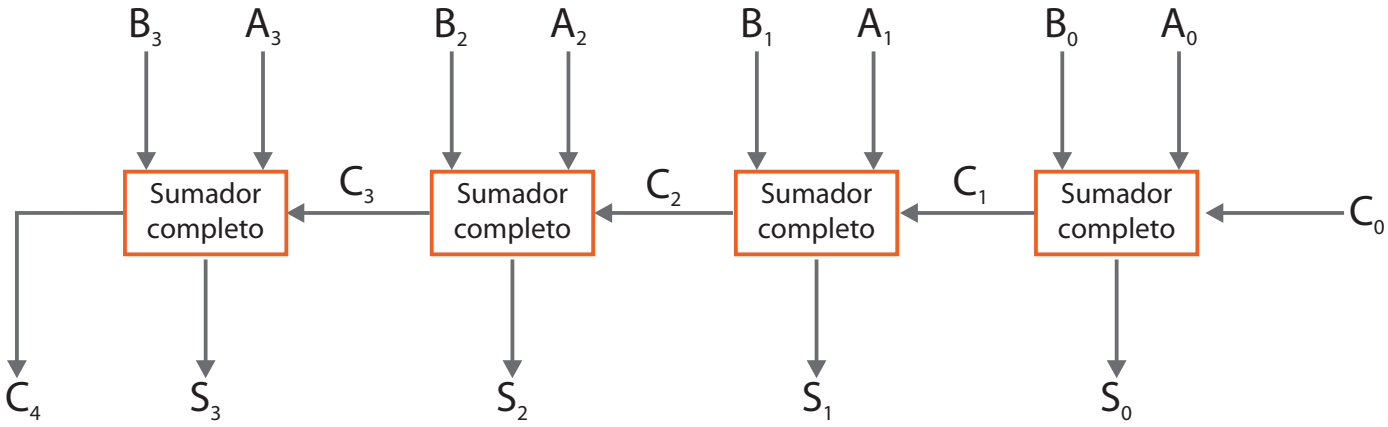


Figura 22. Esquema de un sumador en paralelo con acarreo serie de 4 bits. Fuente: elaboración propia.

El acarreo de entrada C_0 para la pareja de bits menos significativa A_0 y B_0 es 0. Cada sumador completo recibe como entrada los bits correspondientes de los números A y B, así como la entrada de acarreo , y genera como salidas el bit de suma y la salida de acarreo C_{I+0}. El acarreo de salida de cada posición es el acarreo de entrada de la próxima posición de orden superior.

El sumador lógico de 4 bits es el típico ejemplo de un componente digital que puede usarse como un bloque prediseñado. Éste puede emplearse en diversas aplicaciones que impliquen operaciones aritméticas. Si quisiéramos diseñar este circuito por el método usual, necesitaríamos una tabla de verdad con 512 filas debido a que tendríamos 9 entradas binarias. Por tanto, colocando directamente cuatro de los **sumadores completos conocidos** en cascada, se puede obtener una aplicación simple y directa salvando este gran problema.

4.1.4. Semirestador y restador completo

Un **semirestador** es un circuito combinacional con dos entradas A y B y dos salidas R y C_0. La salida R representa el resultado de la resta aritmética de las $A - B$, mientras que la salida C_0 indica el acarreo de la resta. El símbolo asociado al semirestador se ha representado en la figura 23, mientras que la tabla de verdad correspondiente se muestra a continuación (ver tabla 21).

Entradas		Salidas	
A	**B**	R	**C_0**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Tabla 21. Tabla de verdad del semirestador de dos entradas. Fuente: elaboración propia.

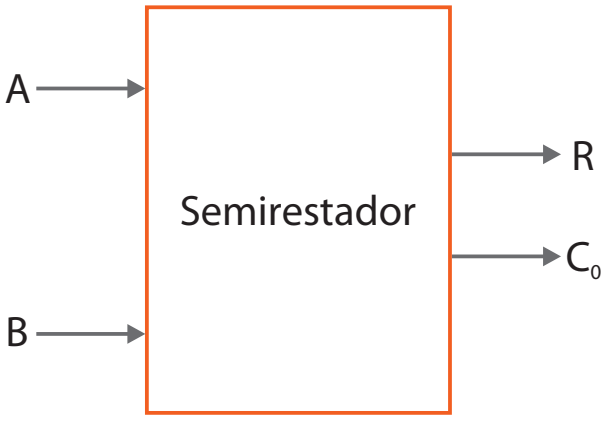


Figura 23. Esquema representativo de un semirestador. Fuente: elaboración propia.

A partir de la tabla de la verdad anterior se obtienen directamente las siguientes expresiones para las salidas R y C_0:

$$R = A \oplus B$$
$$C_0 = \bar{A}B$$

Al igual que el sumador completo, el restador completo contiene tres entradas, A, B y C_{IN}, y dos salidas, que en este caso son R y C_0. La salida representa el resultado de la resta aritmética de las entradas A, B y C_{IN} ($A - B - C_{IN}$) y la salida C_0, el arrastre. El esquema y la tabla de verdad del restador completo se muestran en la figura 24 y en la tabla 22, respectivamente.

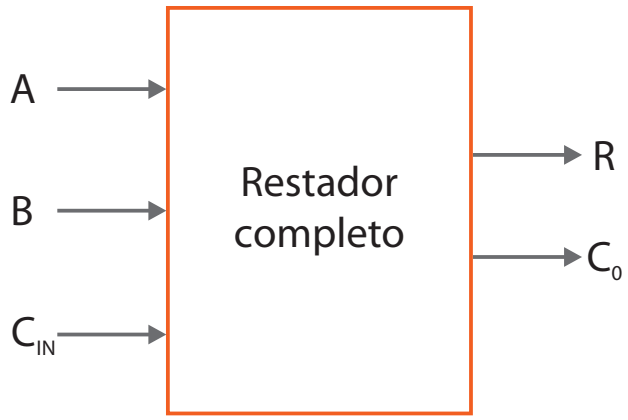


Figura 24. Esquema general de un restador completo. Fuente: elaboración propia.

Entradas			Salidas	
A	**B**	**C_{IN}**	R	**C_0**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Tabla 22. Tabla de verdad del restador completo de dos entradas. Fuente: elaboración propia.

4.1.5. Restador de 4 bits

Basándonos en el sumador completo de 4 bits descrito anteriormente, es posible aplicar el algoritmo de la resta que explicaremos a continuación e implementarlo en un circuito lógico.

Por tanto, si nos regimos por el algoritmo anterior, la resta entre los números binarios 1101 y 1000 (figura 25 (a)) y entre los números 1001 –1100 (ver figura 25 (b) de la página siguiente) sería como se ilustra a continuación.

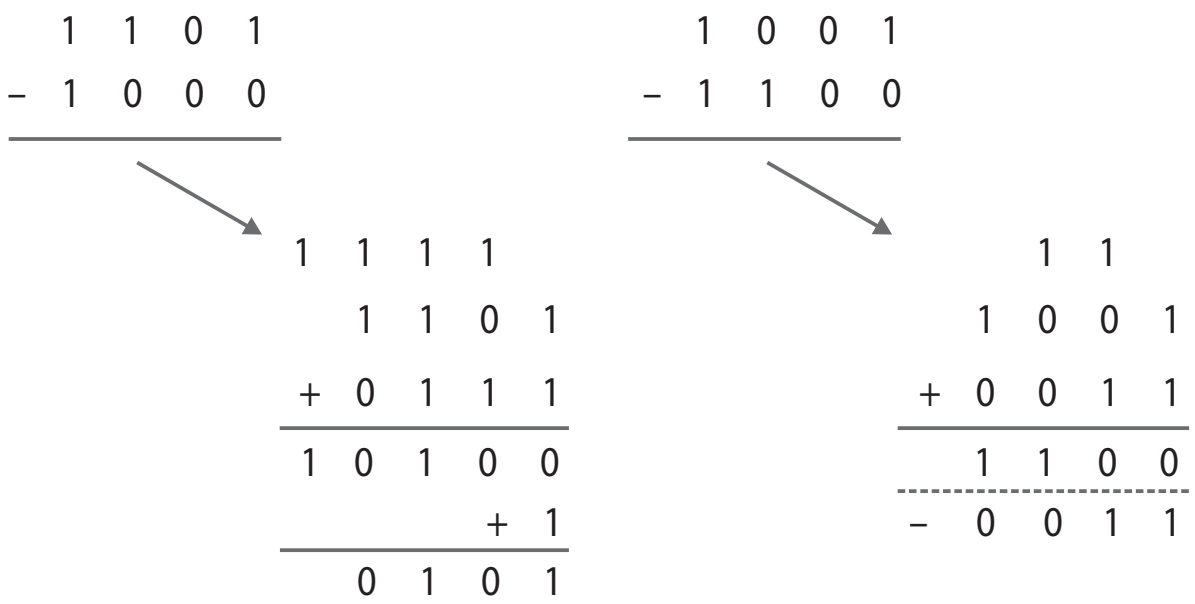


Figura 25. Ejemplos de restas binarias con números de 4 bits. Fuente: elaboración propia.

El resultado para la primera resta sería 0101 positivo y para la segunda 0011 negativo.

Entonces el algoritmo nos indica los siguientes pasos:

1) Definir el complemento del sustraendo a 1. Esto se puede conseguir haciendo uso de compuertas NOT.

2) Sumar el minuendo y el sustraendo complementado en el paso anterior. La figura 26 de la página siguiente muestra estos dos pasos.

Si el acarreo de salida es 1, entonces éste se suma al resultado de la suma y el resultado es positivo. Si el acarreo de salida es 0, se debe complementar el resultado de la suma y el resultado es positivo. Seguidamente se coloca otro sumador completo donde las entradas de A serían las salidas del sumador de la figura anterior. Las entradas de B se fijan con el valor 0000 y el acarreo de entrada se conecta al acarreo de salida del primer sumador. En el caso en que el acarreo de salida del primer sumador sea 1, se sumará ese valor al resultado, si es 0 se sumará 0, manteniéndose por tanto el valor. Si el acarreo es 0, se debe complementar a 1 la salida. Para ello se define la siguiente tabla de verdad.

C_0	X_i	Y_i
0	0	1
0	1	0
1	0	0
1	1	1

Tabla 23. Tabla de verdad del complemento de la resta binaria. Fuente: elaboración propia.

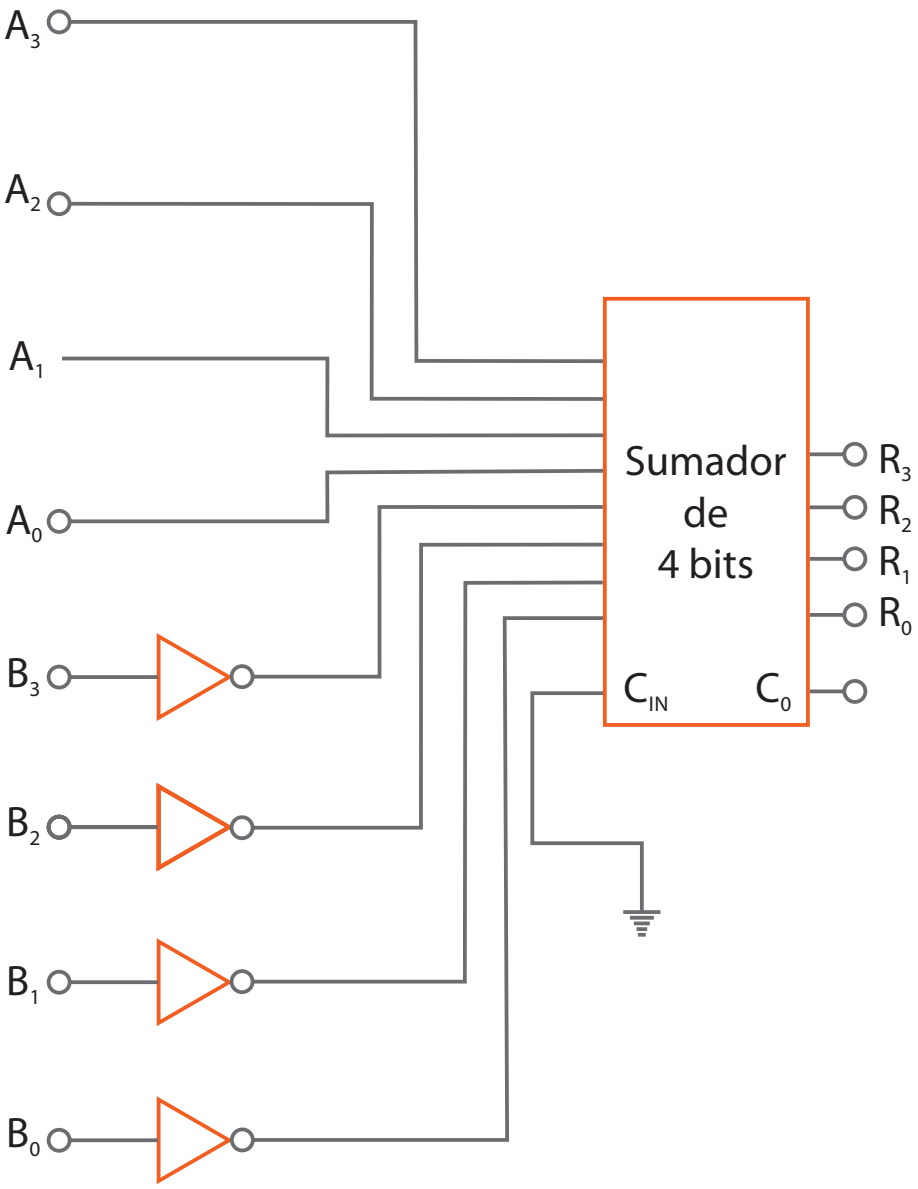


Figura 26. Complemento a 1 y suma para la resta binaria. Fuente: elaboración propia.

donde C_0 es el acarreo de salida de la primer sumador, X_i (compuesta por 4 bits) es la salida del segundo sumador y es la salida final del restador (compuesta por 4 bits). De la tabla anterior se obtiene la siguiente expresión Booleana:

$$Y_i = X_i \oplus C_0$$

La implementación definitiva del restador de 4 bits se representa en la figura 27 de la página siguiente.

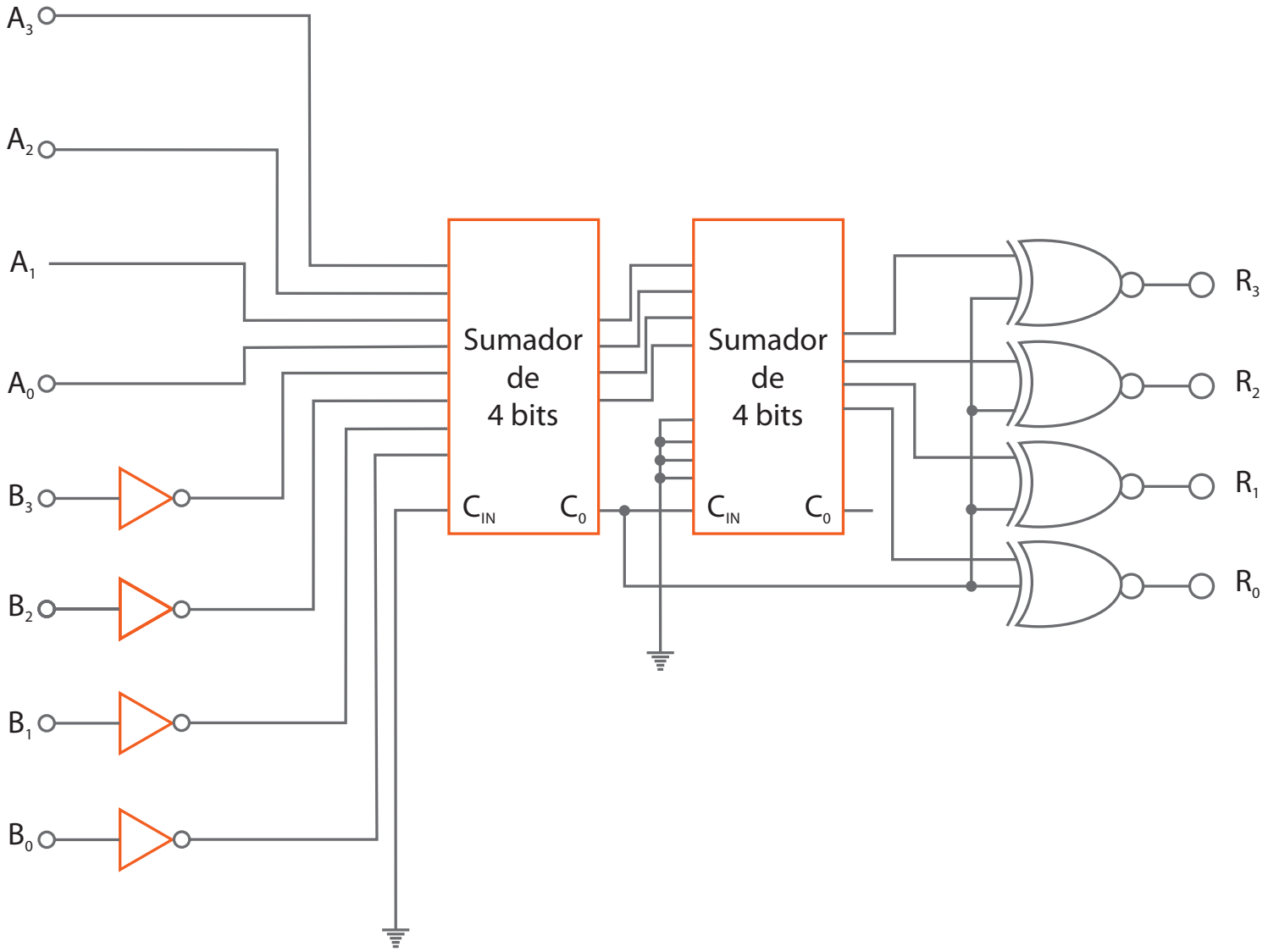


Figura 27. Diagrama completo del restador de 4 bits. Fuente: elaboración propia.

4.1.6. Multiplicador binario

La multiplicación de números binarios se realiza de forma similar a la de dos números decimales. El **multiplicando** se multiplica por cada uno de los bits del **multiplicador**, comenzando por el bit menos significativo (LSB). Las distintas multiplicaciones generadas forman una serie de **productos parciales**, donde cada uno de ellos se desplaza un bit a la izquierda, siendo el producto final la suma de los productos parciales.

Considere la multiplicación de dos números A y B de 2 bits cada uno, tal y como se muestra en la figura 28. Los bits del multiplicando son B_1 y B_0, los bits del multiplicador son A_1 y A_0, y el producto final es C_3, C_2, C_1 y C_0. El primer producto parcial se forma multiplicando B_1 y B_0 por A_0. La multiplicación de los bits A_0 y B_0 produce un 1 si ambos bits son 1, de lo contrario el resultado es un 0, similar al funcionamiento de una **compuerta AND**. El segundo **producto parcial** se obtiene multiplicando los bits B_1 y B_0 por A_1 y desplazándolo una posición a la izquierda. Los dos productos parciales se suman seguidamente con circuitos semisumadores. Por lo general suelen haber muchos más bits en los productos parciales, de ahí que será necesario el uso de sumadores completos para efectuar la suma de los productos parciales.

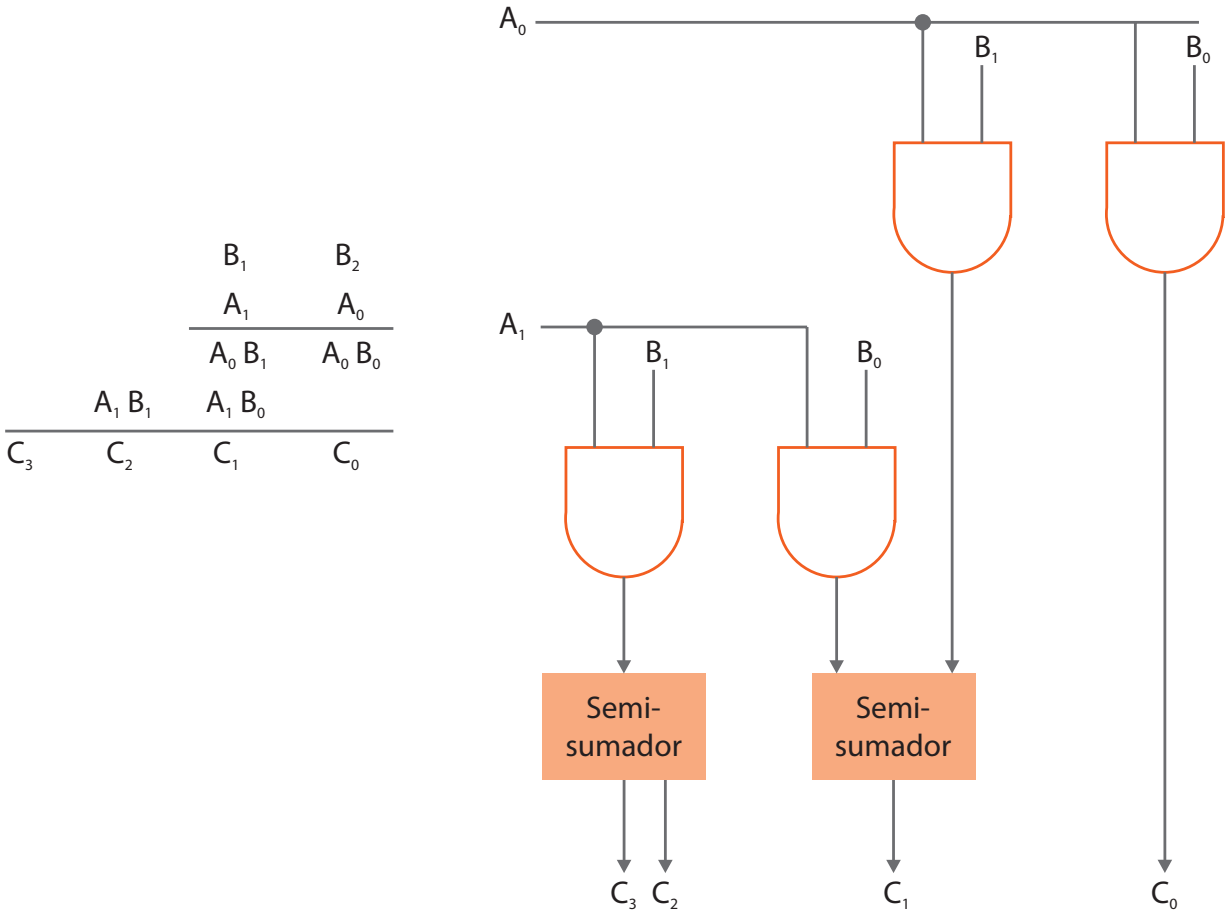


Figura 28. Multiplicador binario de dos bits. Fuente: Mano, M. & Kime, C. (2005). Multiplicador binario de 2#2 bits [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

El multiplicador binario con más de dos bits puede construirse de manera similar. Se realiza la operación AND de un bit del multiplicador con cada bit del multiplicando en tantos niveles como bits

tenga el multiplicador. En cada nivel de puertas AND, la salida binaria se suma en paralelo con el producto parcial del nivel anterior, formando un nuevo producto parcial. Finalmente, el último nivel genera el producto final.

4.1.7. Comparador digital

Un comparador digital consiste en comparar dos números binarios para determinar su relación, es decir, si son iguales o no. Se puede decir que la **compuerta XOR** es un comparador básico, ya que su salida devuelve un 1 si sus dos bits de entrada son diferentes y un 0 si son iguales. Si queremos comparar números binarios de varios bits, entonces se debe usar una compuerta XOR por cada bit que contengan los números a comparar. Un comparador puede verificar también si un número es mayor que otro usando la **compuerta AND**. Si se aplica la siguiente expresión Booleana a un circuito:

$$X = A\bar{B}$$

Si $A = 1$ y $B = 0$, entonces X sería 1 indicando que $A > B$. Para cualquier otra combinación X toma el valor 0 indicando que $A <= B$. Escalando estos conceptos, se puede implementar un circuito comparador de 2 bits, o sea, que tenga dos entradas de 2 bits cada una tal y como se representa en la figura 29.

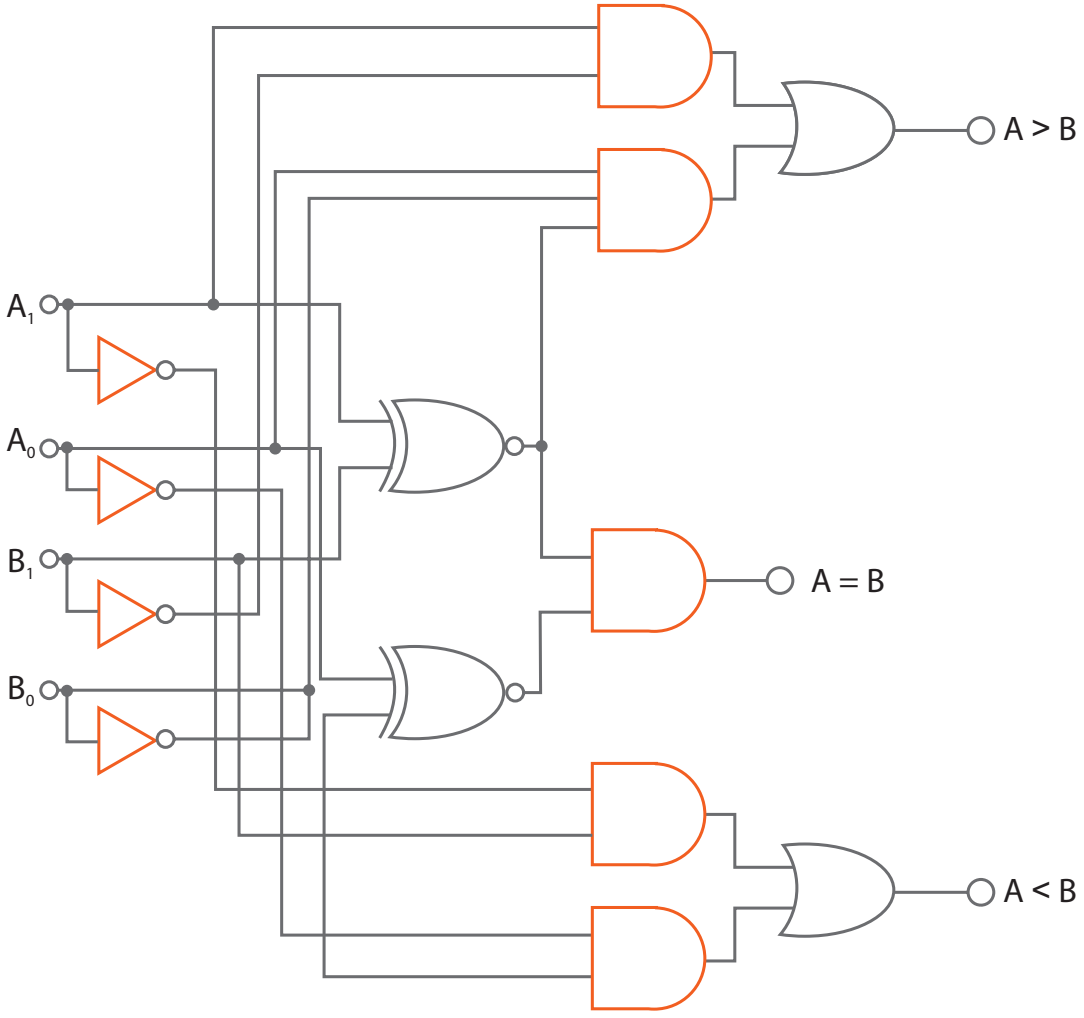
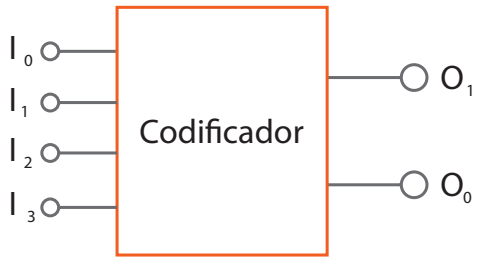


Figura 29. Comparador de números con 2 bits. Fuente: Flórez, H. (2000). Comparador de dos cantidades de dos bits [Figura]. Recuperado de Diseño lógico (1st ed.). Ediciones de la U.

4.2. Codificadores

Los **codificadores**, como bien su nombre lo indica, realizan la función inversa de un decodificador. En un codificador existe un conjunto de entradas, de las cuales sólo se activa una de ellas en cada momento, y un conjunto de salidas que codifican el valor de la entrada activa en ese momento. No existe una relación directa entre el número de entradas y el de salidas, aunque por lo general 2^n entradas producen n salidas. En este último caso, se dice que el codificador es un **codificador completo.**

La figura 30 muestra la tabla de verdad que representa el funcionamiento de un codificador de 4 entradas a binario natural, en el que tanto las entradas como las salidas son activas en alto o con un 1.



				O_1	O_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Figura 30. Esquema y tabla de verdad de un codificador de 4 entradas a números binarios de 2 bits. Fuente: elaboración propia.

Los codificadores se construyen básicamente mediante el uso compuertas OR. La estructura interna del codificador del ejemplo anterior se muestra en la figura 31, la cual puede deducirse directamente de la tabla de verdad de la figura 29, y donde puede observarse que, por ejemplo, la salida O_1 toma el valor 1 cuando las entradas I_2 o I_3 son 1, mientras que O_0 lo hace cuando toman valor 1 las entradas I_1 o I_2.

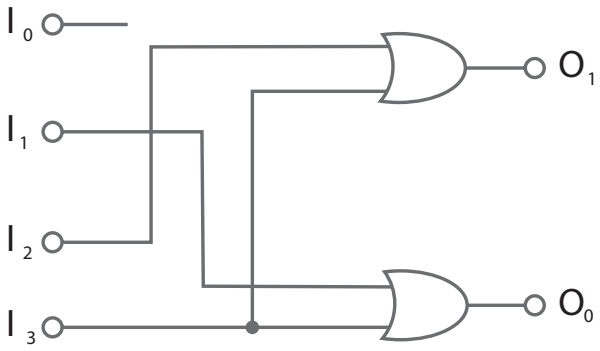


Figura 31. Esquema lógico de un codificador de 4 entradas a un número binario de 2 bits. Fuente: elaboración propia.

De forma similar al codificador anterior, se pueden implementar codificadores de 3 y 4 salidas, que tendrán 8 y 16 entradas, respectivamente. Sin embargo, estos codificadores presentan la limitación de que sólo una de las entradas puede estar activa al mismo tiempo. Por tanto, si dos entradas se activaran simultáneamente, la salida presentaría una combinación incorrecta. Para resolver este problema, algunos diseños de codificadores establecen una prioridad en las entradas, de manera que se asegure que sólo una de ellas sea codificada.

4.2.1. Codificador con prioridad

Como se ha mencionado en el párrafo anterior, la función del codificador con prioridades consiste en definir, para el caso en el que dos o más entradas sean iguales a 1 simultáneamente, cuál será la que tenga mayor prioridad entre ellas. Para un codificador de prioridad de cuatro entradas, la tabla de verdad sería como se muestra en la tabla 24. Haciendo uso de las X, la tabla queda reducida entonces a cinco filas, representando la misma información que la tabla de verdad habitual de 16 filas. Mientras que las X en las columnas de salida representan **condiciones indiferentes**, las X en las columnas de entrada se usan para representar productos de términos que no son minitérminos. Por ejemplo, la combinación de entrada 001X, representaría el producto de términos $\overline{D}_3\overline{D}_2D_1$. Es decir, si un determinado bit en la combinación de entrada es una X, entonces la variable asociada a ese bit no aparece en el producto de términos.

Entradas				Salidas		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Tabla 24. Tabla de verdad para el codificador de prioridad. Fuente: elaboración propia.

Para construir la **tabla de verdad resumida** de un codificador de prioridad, debemos incluir cada minitérmino en al menos una de las filas, de modo que cada uno de ellos pueda obtenerse reemplazando por 1 y 0 las X. Asimismo, un minitérmino nunca debe incluirse en más de una fila, evitando que existan conflictos entre las salidas de varias filas. La tabla 24 se forma entonces de la siguiente manera: la entrada D_3 se define como la entrada de mayor prioridad; es decir, no importa el valor de las otras entradas cuando esta esté a 1, la salida para A_1A_0 es 1 (el equivalente binario de 3). Así, hemos obtenido la última fila de la tabla 24. El siguiente nivel de prioridad lo tiene D_2, donde las salidas A_1A_0 serán 10 si D_2 es 1, y siempre que D_3 sea 0, obviando los valores de las entradas de menor prioridad. De este modo, obtenemos la cuarta fila de la tabla. La salida D_1 para se genera sólo si todas las entradas con mayor prioridad a ella están a 0, sin tener en cuenta los niveles de prioridad que estén por debajo. Así obtenemos las restantes filas de la tabla. La **salida de validación** designada como V se pone a 1 únicamente cuando al menos una de las entradas sea igual a 1. Si todas son 0, entonces $V = 0$ y las dos salidas del circuito A_1A_0 no se emplearán, siendo referidas como indiferencias en la tabla marcándolas con una X.

Los mapas de Karnaugh empleados para simplificar las salidas A_1 y A_0 se muestran en la figura 32 de la página siguiente. A partir de la tabla 24 se han generado los minitérminos de ambas funciones. La versión optimizada de cada función se ha colocado debajo de su mapa correspondiente. La ecuación para la salida V es una función OR de todas las variables de entrada $D_3D_2D_1$ y D_0. Finalmente, el

codificador con prioridad se implementa tal y como muestra la figura 33, de acuerdo con las siguientes funciones booleanas:

$$A_0 = D_3 + D_1\overline{D}_2$$
$$A_1 = D_2 + D_3$$
$$V = D_0 + D_1 + D_2 + D_3$$

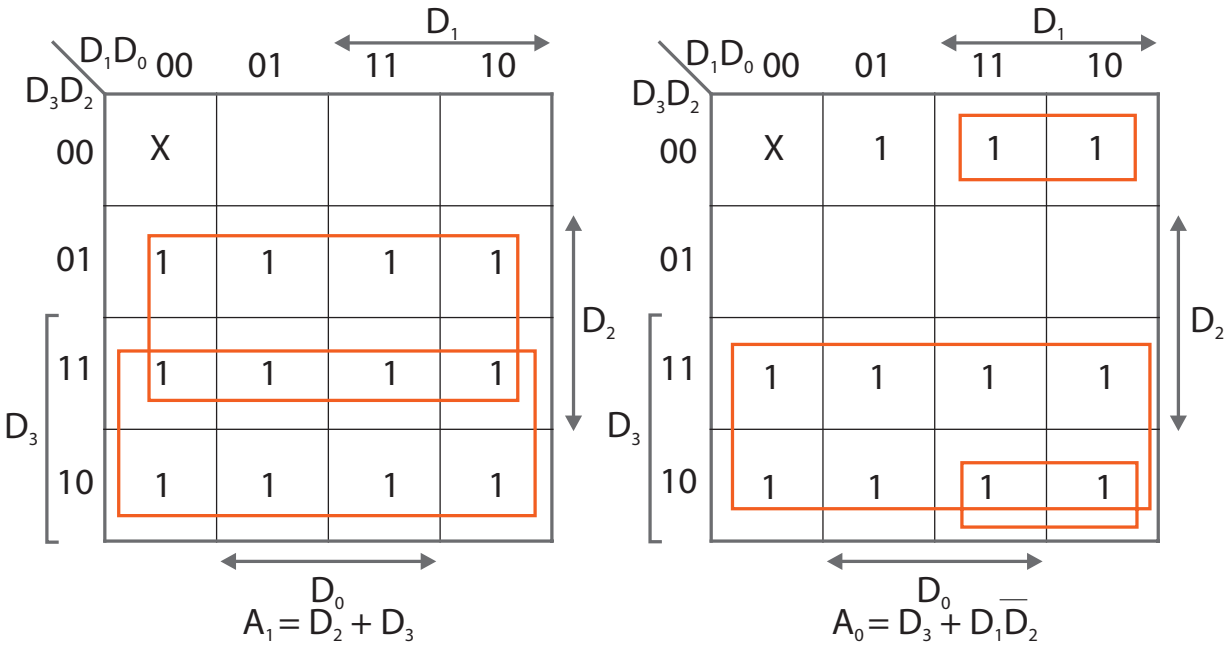


Figura 32. Mapa de Karnaugh asociado al codificador de prioridad. Fuente: elaboración propia.

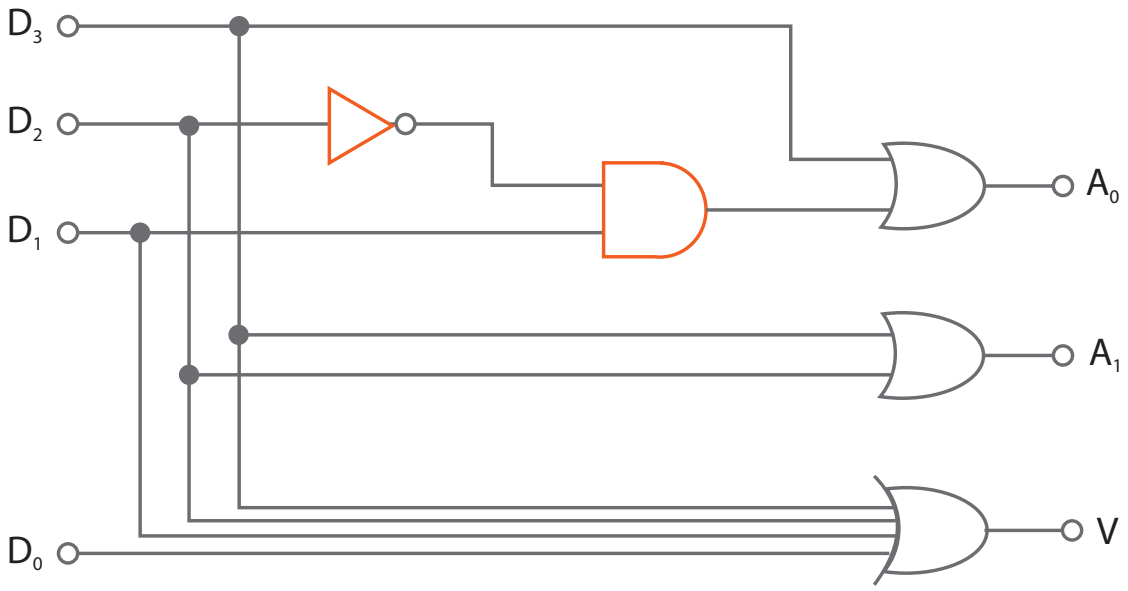


Figura 33. Esquema lógico que representa al codificador de prioridad de la Tabla 24. Fuente: Mano, M. & Kime, C. (2005). Diagrama lógico para un codificador con prioridad de 4 entradas [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

4.3. Decodificadores

Un **decodificador** tiene como función principal convertir un código de entrada de n bits en un código de salida m de bits, tal que $n << m << 2^n$, y donde para cada palabra válida codificada en la entrada, exista un único código de salida. Por tanto, un decodificador es un **circuito combinacional** al que se aplica un código binario de n bits en sus entradas, y genera un código binario de m bits a través de sus salidas. Puede darse el caso en que para ciertas combinaciones de entrada no usadas, el decodificador no genere ningún código por las salidas.

Los bloques funcionales que implementan la decodificación se denominan decodificadores de n a m líneas, donde $n \leq 2^n$. Su propósito consiste en generar 2^n (o menos) minitérminos a partir de las variables de entrada. Para el caso en que $n = 1$ y $m = 2$, obtendríamos el decodificador de 1 a 2 líneas con una entrada A y las dos salidas D_0 y D_1. La figura 34 (a) muestra la tabla de verdad de esta función decodificadora. Si $A = 0$, entonces $D_0 = 1$ y $D_1 = 0$. Si $A = 1$, entonces $D_0 = 0$ y $D_1 = 1$. A partir de esta tabla de verdad, se obtiene que $D_0 = \overline{A}$ y $D_1 = A$, obteniéndose el circuito de la figura 34 (b).

A

A	D_0	D_1
0	1	0
1	0	1

B

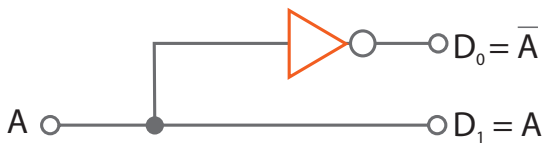


Figura 34. Decodificador digital de 1 a 2 líneas. Fuente: elaboración propia.

Siguiendo la misma lógica del ejemplo anterior, la figura 35 (a) de la página siguiente muestra la tabla de verdad de la función decodificadora para $n = 2$ y $m = 4$, en la cual se ilustra de forma más clara la naturaleza general de los decodificadores. Las salidas de la tabla son minitérminos de dos variables, donde cada fila contiene un valor en la salida igual a 1 y 3 valores iguales a 0. La salida Di es igual a 1 siempre que los dos valores de entrada A_1 y A_0 representen el código binario para el número i. Por lo tanto, el circuito digital estará formado por cuatro posibles minitérminos de dos variables, uno para cada salida.

El circuito lógico de la figura 35 (b) muestra que los minitérminos se implementan usando compuertas AND de 2 entradas. Estas compuertas AND están conectadas a 2 decodificadores de 1 a 2 líneas, uno por cada línea conectada a las entradas de la puerta AND.

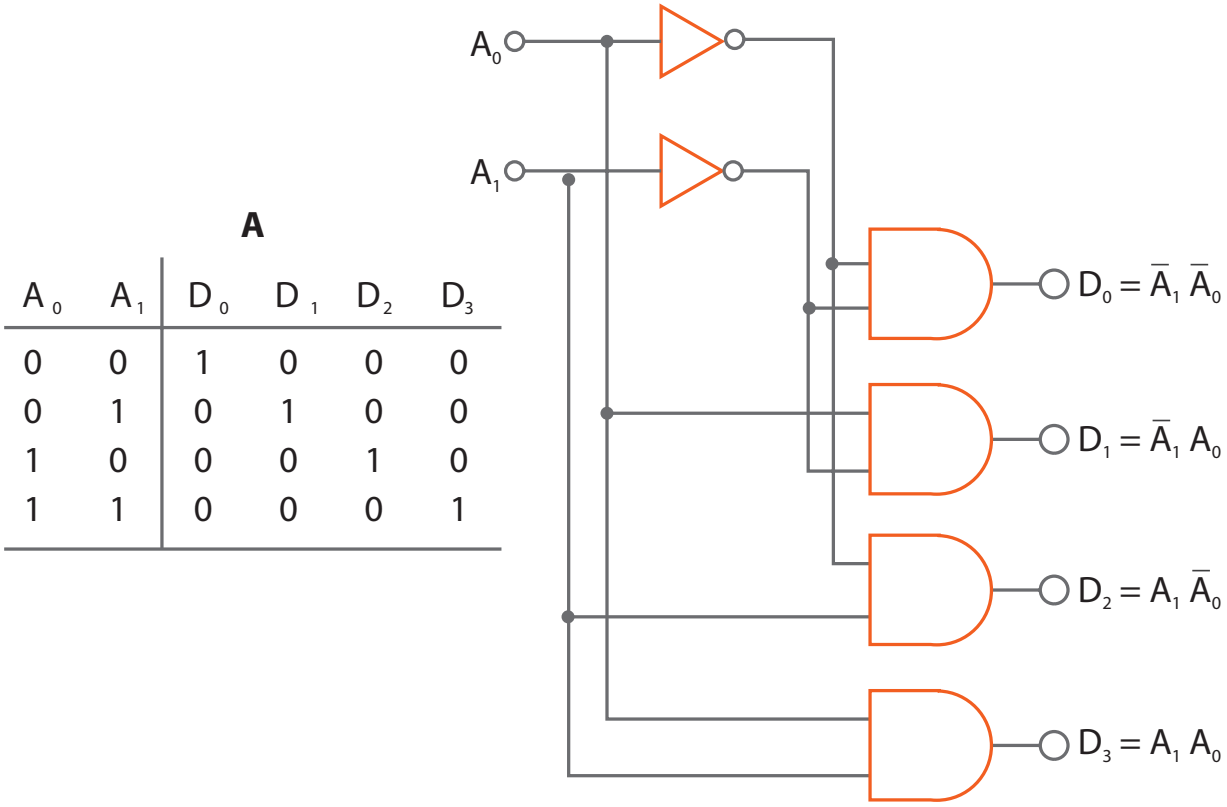


Figura 35. Decodificador digital de 2 a 4 líneas. Fuente: elaboración propia.

4.3.1. Decodificadores con entrada de habilitación

La **decodificación** de n a m líneas con habilitación se puede implementar conectando m circuitos habilitadores a las salidas del decodificador deseado. De esta manera, la **señal de habilitación EN** se conectará con las m copias de la entrada del control de habilitación de los circuitos habilitadores. Para $n = 2$ y $m = 4$, el decodificador con señal de habilitación se muestra en la figura 36 de la página siguiente junto a su tabla de verdad. Cuando EN = 0, todas las salidas del decodificador serán también 0. Cuando EN = 1, solamente una de las salidas del decodificador tendrá valor 1, en función de los valores de A1 y A0, y el resto 0.

A

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	1	0	0	0	1

B

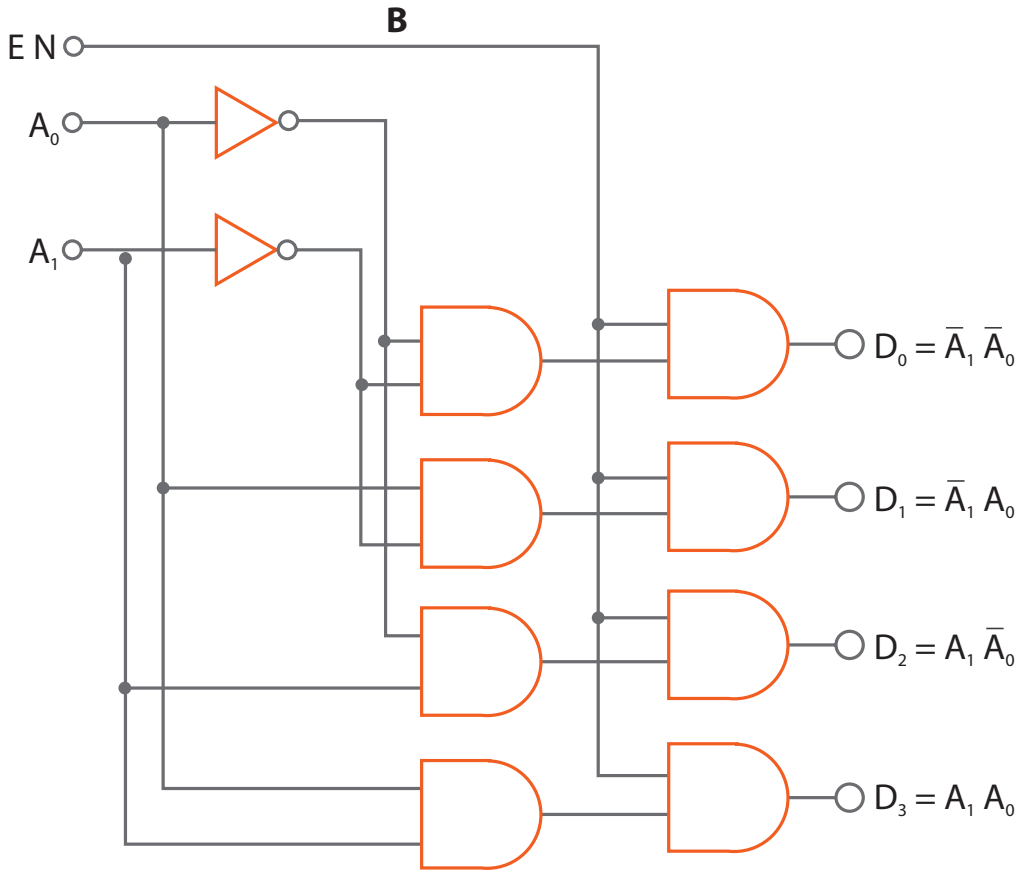


Figura 36. Decodificador digital de 2 a 4 líneas con señal de habilitación. Fuente: elaboración propia.

4.4. Conversores de código

Los conversores de códigos son circuitos combinacionales que transforman una palabra de un código a una palabra perteneciente a otro código, siendo en definitiva traductores de código. Existen varios tipos de conversores, entre los que se pueden destacar los siguientes:

- Binario/Gray.
- BCD/Gray.
- BCD/7-segmentos.

4.4.1. Convertidor Binario/Gray

Para implementar el **convertidor Binario-Gray,** en este caso de 4 bits, es necesario plantear la tabla de verdad correspondiente. A partir de la tabla, se elaboran los mapas de Karnaugh por cada una de las variables de salida.

Código Binario	Código Gray
$\mathbf{B_3B_2B_1B_0}$	$G_3G_2G_1G_0$
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001

Tabla 25. Tabla de verdad para el convertidor Binario/Gray. Fuente: elaboración propia.

Luego de obtener las funciones simplificadas a partir de los mapas de Karnaugh de cada variable de salida, la implementación del circuito lógico sería como el de la figura 37, basado en las funciones siguientes:

$$G_0 = B_1 \oplus B_0$$
$$G_1 = B_2 \oplus B_1$$
$$G_2 = B_3 \oplus B_2$$
$$G_3 = B_3$$

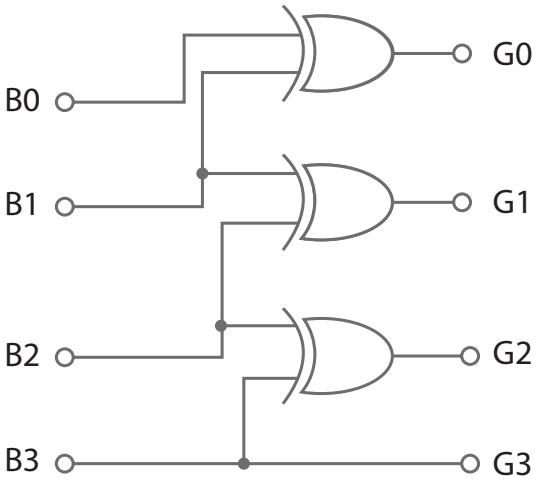


Figura 37. Circuito lógico del converso Binario-Gray. Fuente: elaboración propia.

4.4.2. Conversor Binario-BCD

Para implementar el conversor binario-BCD plantearemos la siguiente tabla de verdad, en la cual tendremos 5 salidas. Los 4 bits menos significativos $C_3C_2C_1C_0$ corresponden al dígito menos significativo de un número decimal de dos dígitos (del 0 a 9), mientras que el bit menos significativo C_4 corresponde al segundo digito del número decimal, que en este caso siempre será 1 para los número del 10 al 15.

Binario				BCD				
B_3	B_2	B_1	B_0	C_4	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0

Binario				BCD				
1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

Tabla 26. Tabla de verdad del converso binario-BCD. Fuente: elaboración propia.

Una vez que se han construido los mapas de Karnaugh a partir de la tabla anterior y para cada una de las salidas, nos quedamos con las siguientes funciones simplificadas que servirán para diseñar el circuito lógico del conversor binario-BCD (ver figura 38).

$$C_0 = B_0$$
$$C_1 = \bar{B}_3 B_1 + B_3 B_2 \bar{B}_1$$
$$C_2 = B_2(\bar{B}_3 + B_1)$$
$$C_3 = B_3 \bar{B}_2 \bar{B}_1$$
$$C_4 = B_3(B_2 + B_1)$$

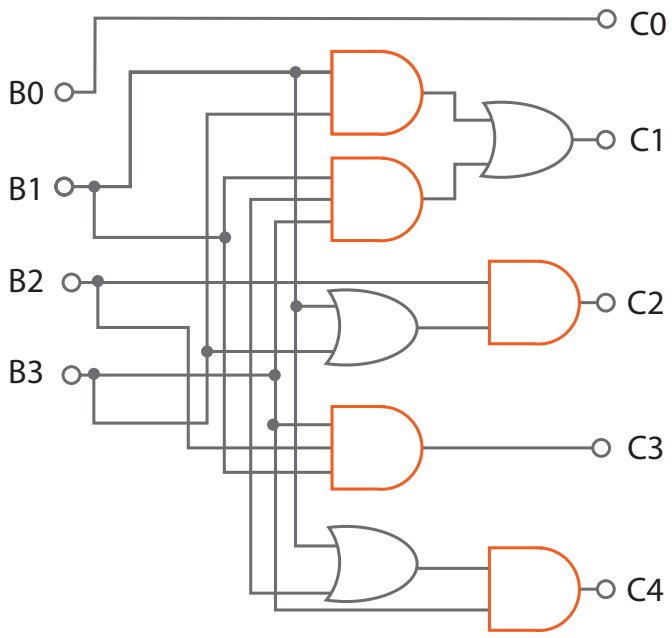


Figura 38. Conversor digital Binario-BCD de dos dígitos. Fuente: elaboración propia.

4.4.3. Conversor BCD - 7 segmentos

Un tipo de conversor muy empleado en el diseño de sistemas digitales es el convertidor de código BCD a código 7-segmentos. Básicamente, el código 7-segmentos es un código que se utiliza para iluminar los distintos segmentos de un *display* numérico que sirve para representar generalmente los números decimales del 0 al 9, tal y como se muestra en la figura 39 de la página siguiente.

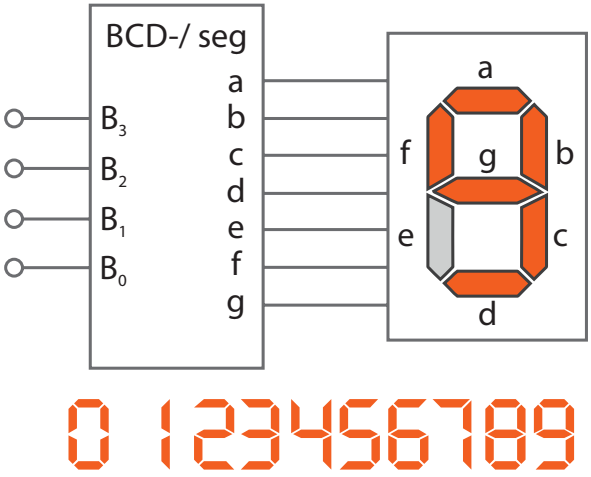


Figura 39. Conversor BCD-7 segmentos de dos dígitos y representación de los números del 0 al 9. Fuente: elaboración propia.

Por ejemplo, si se desea visualizar el número 9 en el *display* 7 segmentos, el segmento **e** debe estar apagado, es decir, la entrada **e** debe estar en cero (0) y las restantes en uno (1). Si queremos visualizar el 8, entonces todos los segmentos deben estar encendidos (1). Por tanto, podemos deducir fácilmente la tabla de verdad asociada al conversor de la figura 39 la cual se expone en la tabla 27. Algunos *displays* de 7 segmentos activan sus segmentos de forma contraria por lo que estos se iluminan poniendo las señales a 0 lógico. En estos casos, las siete salidas del codificador se deben invertir.

Código BCD	Código 7 segmentos
$B_3B_2B_1B_0$	**a b c d e f g**
0 0 0 0	1 1 1 1 1 1 0
0 0 0 1	0 1 1 0 0 0 0
0 0 1 0	1 1 0 1 1 0 1
0 0 1 1	1 1 1 1 0 0 1
0 1 0 0	0 1 1 0 0 1 1
0 1 0 1	1 0 1 1 0 1 1
0 1 1 0	0 0 1 1 1 1 1
0 1 1 1	1 1 1 0 0 0 0
1 0 0 0	1 1 1 1 1 1 1
1 0 0 1	1 1 1 0 0 1 1

Tabla 27. Tabla de verdad del conversor BCD-7 segmentos. Fuente: elaboración propia.

La información de la tabla de verdad anterior la utilizamos en los Mapas de Karnaugh, a partir de los cuales obtenemos las optimizaciones para las funciones de salida. Una posible solución luego de simplificar las siete funciones quedaría de la forma siguiente:

$$a = \bar{A}C + \bar{A}BD + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}$$
$$b = \bar{A}\bar{B} + \bar{A}\bar{C}\bar{D} + \bar{A}CD + A\bar{B}\bar{C}$$
$$c = \bar{A}B + \bar{A}D + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}$$
$$d = \bar{A}C\bar{D} + \bar{A}\bar{B}C + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C} + \bar{A}B\bar{C}D$$
$$e = \bar{A}C\bar{D} + \bar{B}\bar{C}\bar{D}$$
$$f = \bar{A}B\bar{C} + \bar{A}\bar{C}\bar{D} + \bar{A}B\bar{D} + A\bar{B}\bar{C}$$
$$g = \bar{A}C\bar{D} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$$

Como puede verse, la implementación de estas siete funciones de manera independiente exige emplear 27 compuertas AND y 7 puertas OR. Sin embargo, es posible reducir el circuito final compartiendo aquellos productos que son comunes a las diferentes funciones de salida, consiguiéndose un importante ahorro en cuanto al número total de entradas. Un ejemplo sería el término $\overline{B}\,\overline{C}\,\overline{D}$, el cual aparece en las expresiones de a, c, d y e. La salida de la compuerta AND que implementa este producto se conecta directamente a las entradas de las compuertas OR de estas cuatro funciones.

Muchas veces en la literatura a este conversor de BCD a siete segmentos se le denomina **decodificador**, porque decodifica, a partir de un dígito decimal, un código binario. Sin embargo, lo cierto es que se trata de un **conversor de código** ya que convierte un código decimal de 4 bits en un código de siete bits. La palabra «decodificador» está normalmente reservada para otro tipo de circuitos como los ya visto en el apartado anterior.

4.4.4. Multiplexores

Un **multiplexor** es un circuito combinacional que permite seleccionar, a partir de varias entradas binarias, solamente una de ellas, para direccionar dicha información hacia una única salida del circuito. La selección de una de las entradas se controla mediante variables de entrada, denominadas **líneas de selección**. Si hay 2^n entradas, entonces habrá n entradas de selección, cuya combinación binaria determinará cuál de las entradas será seleccionada. Con $n = 1$, el multiplexor sería de 2 a 1, con dos entradas de información, I_0 e I_1, y una única variable de selección S. La tabla de verdad para este circuito se muestra en la tabla 28 de la página siguiente. Básicamente, si la entrada de selección es $S = 1$, la salida del multiplexor Y tomará el valor de la entrada I_1, sino, si la entrada de selección es $S = 0$, entonces la salida del multiplexor Y tomará el valor de la entrada I_0. Finalmente, la expresión para la salida del multiplexor sería:

$$Y = \bar{S}I_0 + SI_1$$

La ecuación anterior se puede obtener utilizando un Mapa de Karnaugh de 3 variables. El circuito digital obtenido para el multiplexor de 2 a 1 se muestra en la figura 40 de la página siguiente, el cual se puede descomponer en un decodificador de 1 a 2 líneas, dos circuitos de habilitación y una puerta OR de 2 entradas.

Selección	Entradas		Salida
S	I_1	I_0	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabla 28. Tabla de verdad del multiplexor de 2 a 1. Fuente: elaboración propia.

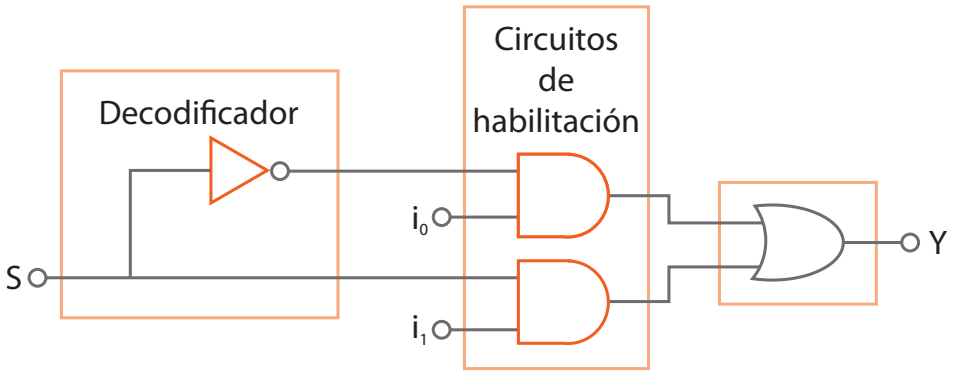


Figura 40. Circuito digital asociado al multiplexor de 2 entradas y 1. Fuente: elaboración propia.

Si ahora quisiéramos diseñar un multiplexor de 4 a 1 líneas, la función Y dependería de cuatro entradas I_0, I_1, I_2 e I_3 de datos y dos entradas de selección S_0 y S_1. Poniendo en la columna los valores desde I_0 hasta I_3, podemos construir la tabla 29 que representa la tabla de verdad resumida para este multiplexor. La información de las variables de entrada no aparece en las columnas de entrada de la tabla, pero si en la columna de salida. Por cada fila de la tabla resumida tendríamos varias filas de la tabla completa. La siguiente la ecuación sería la expresión lógica para la salida :

$$Y = (\bar{S}_1\bar{S}_0)I_0 + (\bar{S}_1 S_0)I_1 + (S_1\bar{S}_0)I_2 + (S_1 S_0)I_3$$

Selección		Salida
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Tabla 29. Tabla de verdad del multiplexor de 4 a 1. Fuente: elaboración propia.

Esta implementación de multiplexor se puede realizar combinando un decodificador de 2-4 líneas, cuatro compuertas AND empleadas como circuitos de habilitación y una puerta OR de 4 entradas, tal y como muestra la figura 41 b). El circuito resultante cuenta con 22 entradas de puertas, lo que incrementa su coste. Sin embargo, éste sería la base estructural para diseñar por la vía de expansión, grandes multiplexores de n a 2^n líneas.

A un multiplexor también se denomina **selector de datos**, ya que éste se encarga de seleccionar solo una de sus muchas entradas, para llevar la información binaria a la salida del mismo. El término «multiplexor» es a menudo es abreviado como «MUX».

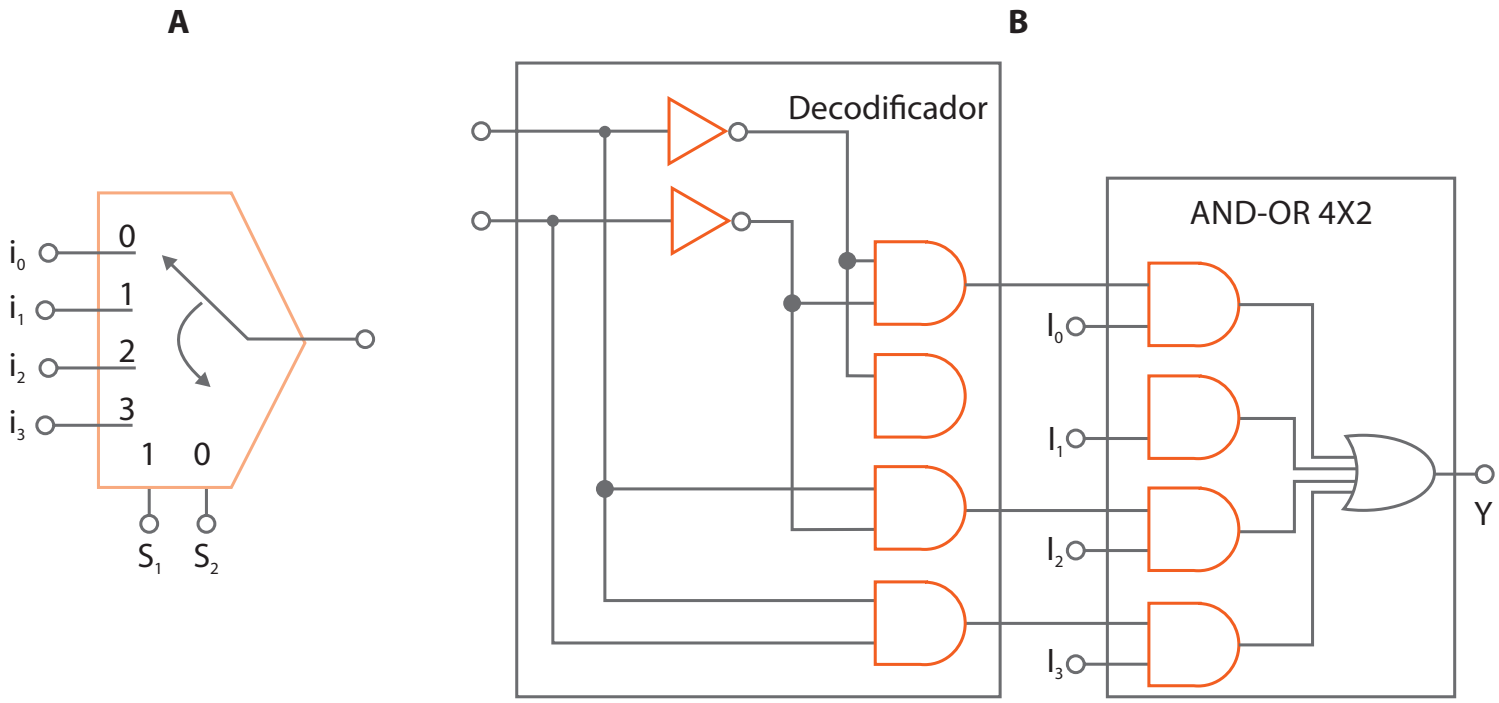


Figura 41. a) Estructura externa de un multiplexor de 4 y 1 líneas; b) circuito digital interno del multiplexor. Fuente: elaboración propia.

4.4.5. Demultiplexores

Lo contrario a la selección es la **distribución**, en la cual la información recibida proviene de una única línea que es transmitida a cada una de las 2^n posibles líneas de salida. El circuito digital que implementa esta distribución se denomina **demultiplexor**. Para controlar cuál de las señales de entrada será transmitida a la salida, se emplea una combinación de bits sobre las n **líneas de selección**. El decodificador 2 a 4 líneas con habilitación de la figura 33 es una implementación de un demultiplexor de 1 a 4 líneas. En este caso, la entrada EN proporciona los datos, mientras que las otras entradas (A1 y A0) actúan como variables de selección. Aunque ambos circuitos tienen aplicaciones diferentes, sus diagramas lógicos son exactamente iguales. De ahí que a un decodificador con entrada de habilitación, se denomina también **decodificador/demultiplexor**. La entrada de datos EN se conecta con las cuatro salidas, pero la información sólo es direccionada hacía una de ellas, en función de la combinación presente en las líneas de selección A1 y A0. La tabla de verdad siguiente muestra el funcionamiento de un demultiplexor de 4 canales.

Selección			Salidas			
EN	A_1	A_0	O_3	O_2	O_1	O_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Tabla 30. Tabla de verdad del demultiplexor de1 a 4. Fuente: elaboración propia.

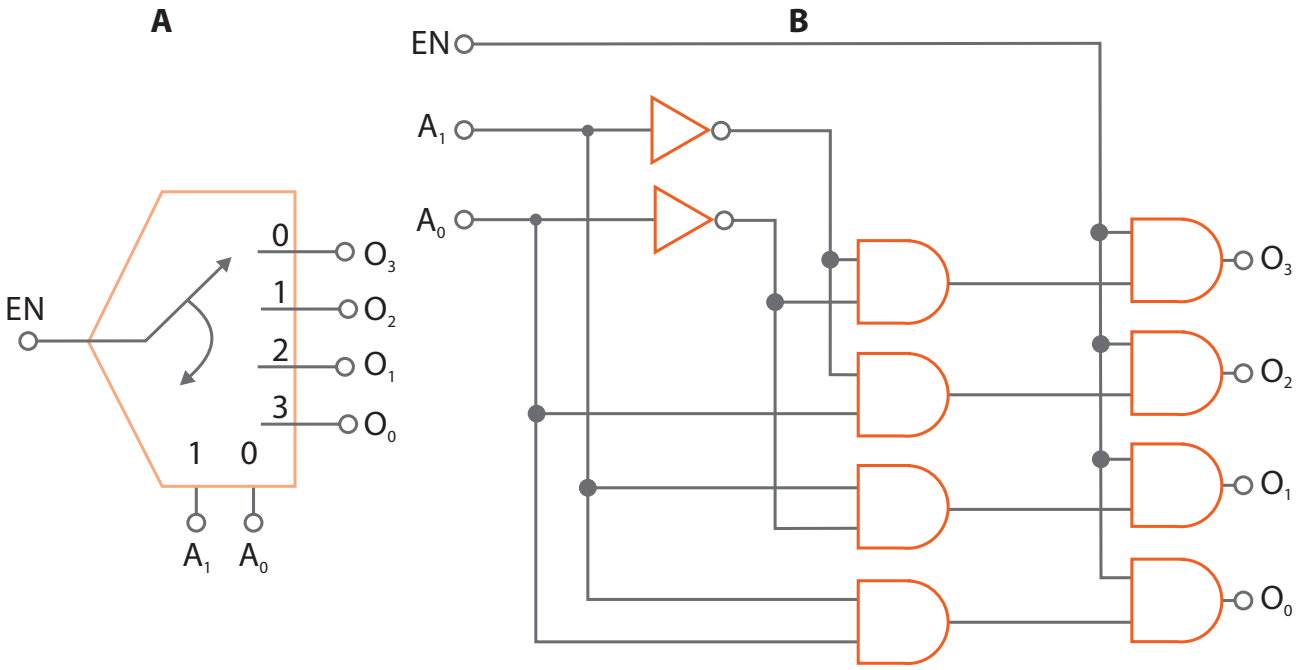


Figura 42. a) Estructura externa del demultiplexor de 1 y 4 líneas; b) Circuito lógico del demultiplexor. Fuente: elaboración propia.

Tema 5.
Circuitos secuenciales. Biestables

Hasta ahora hemos visto los distintos tipos de circuitos combinacionales más usados en el diseño de sistemas digitales. Sin embargo, muchos de los sistemas digitales que se diseñan hoy día contienen elementos de memoria, convirtiéndolos de esta forma en **sistemas secuenciales**. En este capítulo comenzaremos el estudio de los circuitos secuenciales y de los elementos básicos que se utilizan para almacenar información binaria, denominados *latches* y *flip*-flops. Veremos cuáles son las diferencias entre los latches y los *flip-flops* y estudiaremos varios tipos de ambos circuitos.

En la figura 43 de la página siguiente se muestra el diagrama de un sistema secuencial, donde se puede apreciar la interconexión de un circuito combinacional con elementos de almacenamiento o memoria. Los elementos de memoria son circuitos capaces de almacenar información binaria. La información binaria guardada en estos elementos en un determinado momento define el estado actual del circuito secuencial. El circuito secuencial recibe información binaria desde su entorno a través de las entradas que, junto a la información almacenada en los elementos de memoria, determinan el valor binario de las salidas. También determinan cuáles serán los valores para el próximo estado de los elementos de almacenamiento. El diagrama de bloques demuestra que las salidas de un circuito secuencial no sólo son función de las entradas, sino también del estado actual de los elementos de memoria. El siguiente estado de los elementos de memoria también estará en función de las entradas así como del estado presente. De este modo, un **circuito secuencial** se puede definir como una sucesión en el tiempo de entradas, estados interiores, y salidas.

Existen dos tipos de circuitos secuenciales, dependiendo del momento en que se observan sus entradas y de los cambios de su estado interior. El comportamiento de un circuito secuencial **síncrono** se define a partir del conocimiento de sus señales en instantes discretos de tiempo. El comportamiento de un circuito secuencial asíncrono depende tanto de las entradas en cualquier instante de tiempo como del orden en que cambian las entradas a lo largo del tiempo.

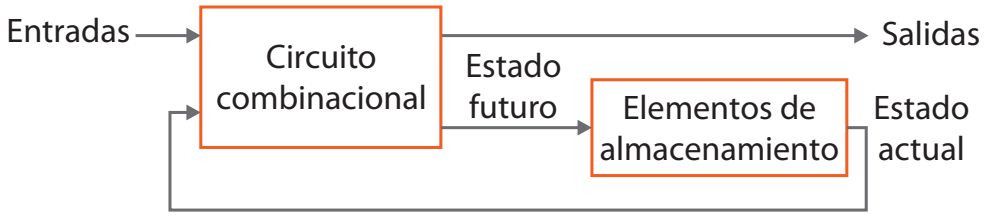


Figura 43. a) Diagrama en bloques de un sistema digital secuencial. Fuente: elaboración propia.

5.1. Latches

Un **latch** es un dispositivo de almacenamiento de dos estados binarios. Este dispositivo posee además una realimentación interna, que consiste en conectar una salida con la entrada opuesta. Con esto se consigue mantener siempre un estado a la salida del circuito.

5.1.1. Latch *S* – *R*

En la figura 44 se muestra el circuito lógico para un elemento sencillo de memoria denominado **latch SR**. Este circuito posee dos entradas (S y R, correspondientes a los estados "*set*" y "*reset*", respectivamente) y al menos una salida (Q). Muchas veces incluyen también una salida negada ($\overline{Q}$) tal y como se muestra en la figura. Como se puede ver, la señal Q se invierte lógicamente y se conecta a la compuerta G2, produciendo la salida $\overline{Q}$. Esta señal, a su vez, se invierte lógicamente y se acopla de regreso a la entrada de la compuerta G1 generando la salida Q, completando así el lazo. Este arreglo se puede utilizar para almacenar un solo bit de información cuando ambas entradas, R y S, son 0s lógicos. Cuando $Q = 1$ y $\overline{Q} = 0$, se dice que el biestable está en el estado *SET*, y cuando $Q = 0$ y $\overline{Q} = 1$, está en el estado *RESET*. Las salidas Q y $\overline{Q}$ normalmente son complementarias. En el caso de que ambas entradas sean iguales a 1 simultáneamente, se produce un estado indefinido con ambas salidas igual a 0.

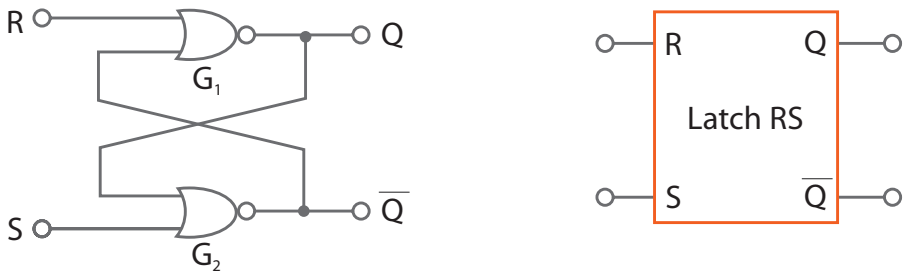


Figura 44. Circuito lógico asociado a un latch del tipo SR. Fuente: elaboración propia.

En la tabla 31 se muestra el funcionamiento del *latch* RS de la figura 44. Se puede apreciar que hay dos condiciones de entrada que causan el estado *set*. La condición inicial es $S = 1$, $R = 0$, la cual pone al

circuito en estado *set*. Aplicar un 0 en la entrada S con $R = 0$ mantiene al circuito en el mismo estado set. Una vez que ambas entradas están en 0, se puede pasar al el estado *reset* poniendo un 1 a la entrada R. Si posteriormente retiramos el 1 de R, el circuito permanecerá en el estado *reset*. De este modo, cuando ambas entradas son 0, el *latch* puede estar en *set* o en *reset*, dependiendo de cuál fue la última entrada que se puso un 1. En condiciones normales, las dos entradas del *latch* permanecen a 0 a menos que deseemos que el estado de éste cambie.

Si se aplica un 1 en ambas entradas del *latch*, las dos salidas Q y $\overline{Q}$ serían 0. Esto produciría un estado indefinido ya que violaría la condición de que las salidas son complementarias entre ellas. Cuando ambas entradas devuelven un 0 simultáneamente también se produce un próximo estado indeterminado o imprevisible. En el funcionamiento normal, estos problemas se evitan asegurándonos que no se aplican 1s simultáneamente en ambas entradas.

Entradas		Salidas		Estado
S	R	Q	$\overline{Q}$	
1	0	1	0	*SET*
0	0	1	0	
1	0	0	1	*RESET*
0	0	0	1	
1	1	0	0	*INDEFINIDO*

Tabla 31. Tabla de verdad del latch SR. Fuente: elaboración propia.

5.1.2. Latch $\overline{S} - \overline{R}$

EL *latch* $\overline{S} - \overline{R}$ posee las mismas características que el *latch* $S - R$ considerando que sus entradas son activas a nivel bajo (0). Este se compone de 2 compuertas NAND acopladas como se muestra en la figura 45. Normalmente funciona con ambas entradas a 1, a menos que el estado del *latch* tenga que cambiar. La aplicación de un 0 en S provoca que en la salida Q se genere un 1, poniendo el *latch* en el estado *set*. Cuando la entrada S se vuelve 1, el circuito permanece en el estado *set*. Con ambas entradas a 1, el estado del *latch* se puede cambiar poniendo un 0 en la entrada R. Esto lleva al circuito al estado *reset* permaneciendo así aun después de que ambas entradas vuelvan a retomar el valor 1. Para este *latch* con compuertas NAND, la condición indefinida se produce cuando ambas entradas son 0 simultáneamente, combinación de entradas que debe evitarse.

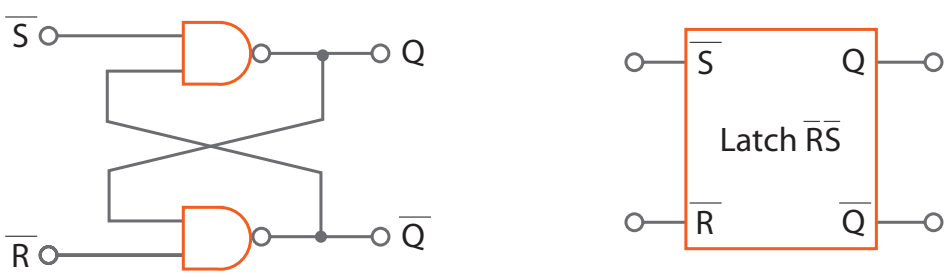


Figura 45. Circuito lógico asociado a un latch del tipo $\overline{S} - \overline{R}$. Fuente: Fuente: elaboración propia.

Entradas		Salidas		Estado
$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
0	1	1	0	*SET*
1	1	1	0	
1	0	0	1	*RESET*
1	1	0	1	
0	0	1	1	*INDEFINIDO*

Tabla 32. Tabla de verdad del latch $\overline{S} - \overline{R}$. Fuente: elaboración propia.

5.1.3. Latch SR con habilitación

El funcionamiento de los *latches* básicos $S - R$ (compuertas NOR) y $\overline{S} - \overline{R}$ (compuertas NAND) se puede modificar añadiendo una entrada de control/habilitación adicional que determina cuándo puede cambiar el estado del *latch*. En la figura 46 se muestra un latch SR con una entrada de control, el cual consiste en un latch NAND básico con dos puertas NAND adicionales, encargadas de habilitar las otras dos entradas S y R a través de la entrada de control *E*.

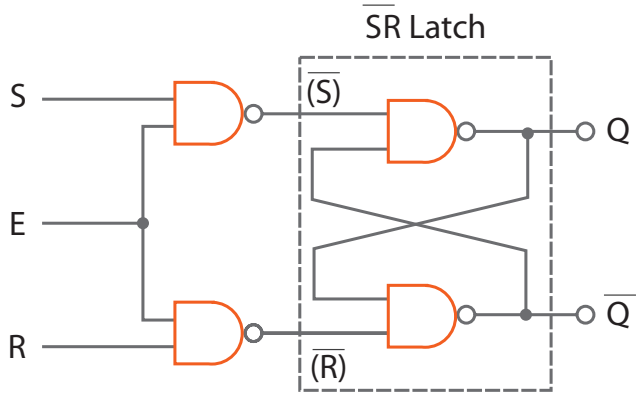


Figura 46. Circuito lógico asociado a un latch del tipo $\overline{S} - \overline{R}$ con entrada de control. Fuente: pulse aquí para ver la fuente.

El circuito anterior trabaja de forma similar al latch $S - R$ pero teniendo una función adicional. Si la entrada E = 0, la salida siempre será "No cambio". Este comportamiento describe el almacenamiento de un bit de información. La tabla de verdad que sigue éste circuito sería la siguiente:

Entradas			Salida
E	S	R	Q_{t+1}
0	X	X	No cambió
1	0	0	No cambió
1	0	1	$Q = 0$ (*Reset*)
1	1	0	$Q = 1$ (*Set*)
1	1	1	Indefinido

Tabla 33. Tabla de verdad del latch $\overline{S} - \overline{R}$ con entrada se control. Fuente: elaboración propia.

Cuando la entrada de control *E* es igual a 0, las salidas de las puertas NAND permanecen en 1. Ésta es la condición de mantenimiento del dato para el *latch* compuesto por dos puertas NAND. Cuando $E = 1$, la información de las entradas S y R afectan al *latch*. El estado *set* se alcanza con $S = 1$, $R = 0$, y $E = 1$. Para cambiar al estado reset, las entradas deben ser $S = 0$, $R = 1$ y $E = 1$. Para cualquier otro caso, cuando *E* vuelve a 0, el circuito permanece en su estado actual. Cuando $E = 0$, el circuito se desactiva para que el estado de las salidas no cambie, sin importar los valores de *S* y *R*. Igualmente, cuando $E = 1$ y las entradas S y R son 0, el estado del circuito no cambia. Cuando las tres entradas son 1 se produce un estado indefinido. Esta condición pone ambas entradas $\overline{S} - \overline{R}$ del *latch* básico a 0, forzando un estado indefinido. Cuando *E* vuelve a ser 0, el siguiente estado no se puede determinar, ya que el *latch* ve las entradas (0, 0) seguidas por (1, 1).

El *latch* $S - R$ con entrada de control es un circuito de gran importancia ya que muchos otros *latches* y *flip-flops* se construyen a partir de él. Muchas veces al *latch* SR con entrada de control se le denomina *flip-flop* RS (o SR), sin embargo, el circuito no cumple los requisitos de los **biestables** que veremos en la próxima sección.

5.1.4. Biestable D

Para eliminar el **estado indefinido** no deseable en el *latch* $S - R$, se debe asegurar que las entradas S y R nunca sean iguales a 1 simultáneamente. Esto se consigue con el *latch* D de la figura 47. Este *latch* sólo contiene dos entradas, la entrada D (de dato) y la entrada C (de control). El complemento de la entrada D va conectado a la entrada $\overline{S}$, y se aplica D a la entrada $\overline{R}$. Si $C = 0$, el *latch* tiene ambas entradas en 1, y el circuito no podrá cambiar de estado independientemente del valor de D. La entrada D se muestrea solo cuando $C = 1$. Si $D = 1$, la salida Q se pone a 1, situando al circuito en el estado set. Si $D = 0$, la salida Q se pone a 0, llevando al circuito al estado *reset*.

La denominación de *latch* D viene por su capacidad para retener el dato (D) en su interior. La información binaria presente en la entrada D es transferida a la salida Q cuando $C = 1$. Mientras C esté habilitada, la salida Q será sensible a los cambios en la entrada D. Cuando $C = 0$, la información binaria que estaba presente en la entrada de datos D en el momento de la transición se retendrá en la salida Q, hasta que C vuelva a habilitarse. La tabla 27 resume el funcionamiento del latch D a partir de su tabla de verdad.

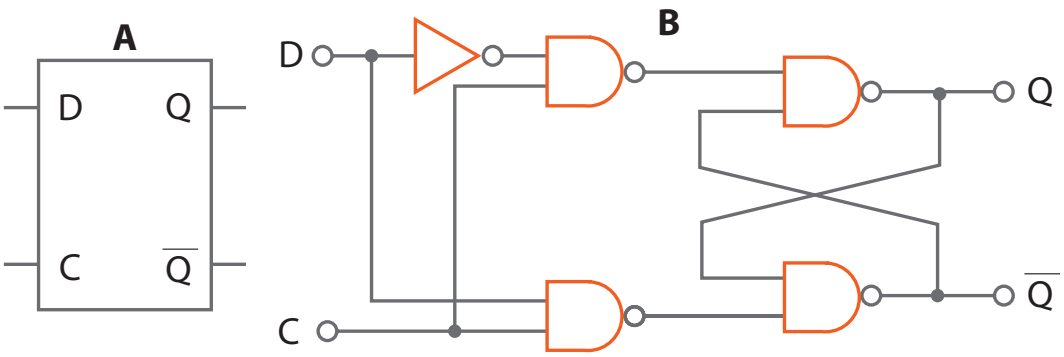


Figura 47. a) Representación externa de un latch D; b) Circuito lógico asociado al latch D. Fuente: pulse aquí para ver la fuente.

Entradas		Salida
C	*D*	Q_{t+1}
0	X	No cambió
1	0	$Q = 0$ (*Reset*)
1	1	$Q = 1$ (*Set*)

Tabla 34 Tabla de verdad del latch. Fuente: elaboración propia.

5.2. Flip-flops

Los *Flip-Flops* se clasifican como circuitos **biestables** síncronos. Esta clasificación viene dada por que la salida del mismo solamente varía en un instante de tiempo específico determinado por una entrada denominada **reloj.** Ese instante de tiempo se denomina **flanco,** y puede ser un **flanco de subida** o **flanco de bajada.** El flanco de subida hace referencia a la transición del estado 0 al 1 que se produce en la entrada de reloj. Por su lado, el flanco de bajada hace referencia a la transición del estado 1 a 0. El tiempo de duración de un flanco es de aproximadamente 30 nanosegundos. La salida de un *Flip-Flop* solamente cambia de estado cuando se presenta un flanco en su entrada de reloj. Los *Flip-Flops* pueden activarse con un flanco de subida o de bajada.

El símbolo de un circuito con presencia de una entrada de reloj, ya sea con flanco de subida o flanco de bajada, se puede identificar con las letras CP (Clock Pulse). La siguiente figura muestra estos símbolos respectivamente.

5.2.1. Flip-flop D

El *Flip-Flop* D es exactamente igual al *latch* D, con la diferencia de que la entrada de habilitación se reemplaza por un reloj. Para ello, solo es necesario agregar al *latch* D un circuito **detector de flancos.** La implementación sería la de un flip-flop D **maestro-esclavo,** que consiste en la conexión de un *latch* D, y un *latch* SR o *latch* D, tal y como se muestra en la figura 48. Este circuito cambia su valor cuando se produce un flanco negativo o de bajada del pulso de reloj. Así, un flip-flop D maestro-esclavo construido tal y como se ha indicado, también es un flip-flop activo por flanco o disparado por flanco.

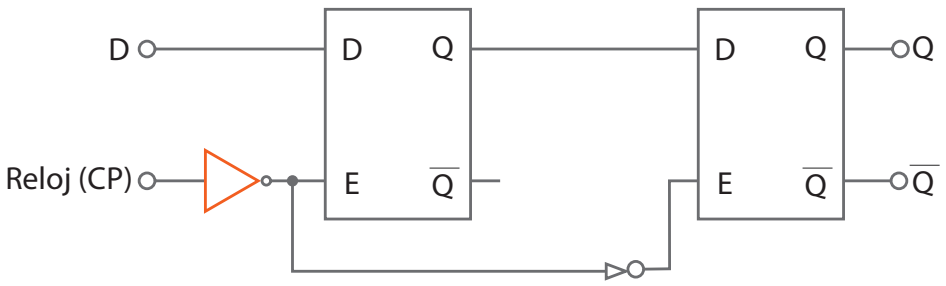


Figura 48. a) Flip-flop D activo con flanco de bajada. Fuente: pulse aquí para ver la fuente.

Como hemos mencionado anteriormente, algunos *flip-flops* se activan con el flanco positivo (transición de 0 a 1). El diagrama lógico de un *flip-flop* tipo D disparado por flanco positivo se muestra en la

figura 49. Este flip-flop toma exactamente la forma de un *flip-flop* maestro-esclavo de la figura 46, pero añadiendo un inversor en la entrada del reloj.

Entradas		Salida
EN	*D*	Q_{t+1}
↑	0	0 (*Reset*)
↑	1	1 (*Set*)

Tabla 35. Tabla de verdad del flip-flop D. Fuente: elaboración propia.

Cuando la entrada de reloj es igual a 0, el **maestro** (primer *latch* D) se habilita y se hace transparente de modo que D sigue al valor de la entrada. El **esclavo** (segundo *latch* D) es inhabilitado y mantiene el estado anterior del *flip-flop* fijo. Cuando se produce un flanco positivo, la entrada de reloj cambia a 1, lo que desactiva al maestro fijando su valor y habilita al esclavo para que copie el estado del maestro en la salida. Cuando la entrada de reloj es igual a 1, el maestro es deshabilitado y no puede cambiar, de modo que el estado del maestro y del esclavo permanezca inalterados. Finalmente, cuando el reloj cambia de 1 a 0, el maestro se habilita y comienza siguiendo al valor de D. Pero durante la transición de 1 a 0, el esclavo se deshabilita antes de que pueda alcanzar cualquier cambio del maestro. Así, el valor almacenado en el esclavo permanece inalterado durante esta transición.

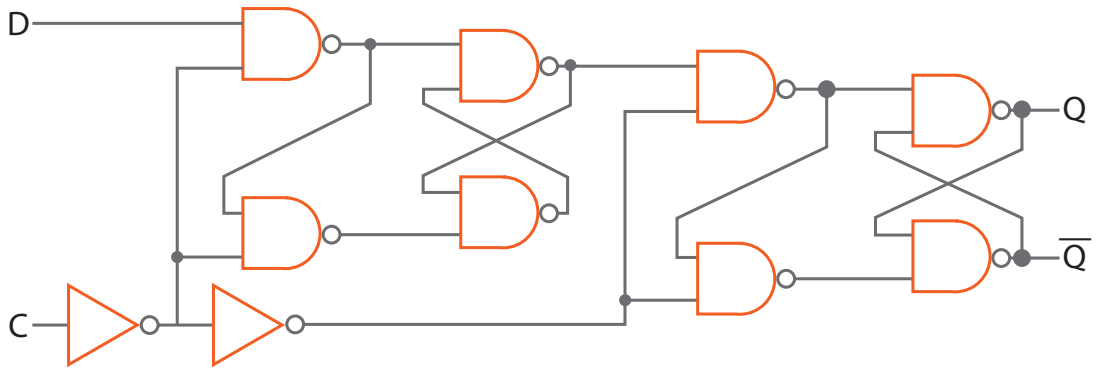


Figura 49. a) Flip-flop D activo con flanco de subida. Fuente: pulse aquí para ver la fuente.

5.2.2. Flip-flop J-K

El *Flip-Flop* J-K es un circuito secuencial cuyo comportamiento es idéntico al *latch* S-R en las operaciones de *Set*, *Reset* y No cambio. La ventaja del Flip-Flop J-K es que es disparado por flanco y además no presenta el estado No válido. Mientras el flip-flop SR produce salidas indefinidas y un comportamiento indeterminado cuando $S = R = 1$, el *flip-flop* J-K genera por su salida el complemento de su valor actual para $J = K = 1$. Los *flip-flops* J-K disparados por flanco se construyen a partir de *flip-flops* D, también disparados por flanco como su diagrama lógico empleando un *flip-flop* D activo por flanco positivo. La tabla característica que describe el comportamiento del *flip-flop* J-K se muestra en la tabla 36. La entrada *J* se comporta similar a la entrada *S* para poner en el estado *Set* al *flip-flop*. Por su lado, la entrada *K* se comporta similar a la entrada *R* para pone el estado de *Reset* al *flip-flop*. La única diferencia entre los *flip-flops* SR y los *flip-flops* JK es su respuesta cuando ambas entradas son iguales a 1. Como puede verificarse a partir del diagrama lógico, esta condición complementa el estado del *flip-flop* J-K.

Cuando J = 1 y Q = 0, entonces D = 1, complementando las salidas del *flip-flop* J-K. Cuando K = 1 y Q = 1, entonces D = 0, complementando las salidas del *flip-flop* J-K. Esto demuestra que, sin tener en cuenta el valor de Q, la condición J = 1 y K =1 provoca la complementación de las salidas del *flip-flop* en respuesta al pulso de reloj. El comportamiento del siguiente estado se resume en la columna de la tabla característica de la tabla 36.

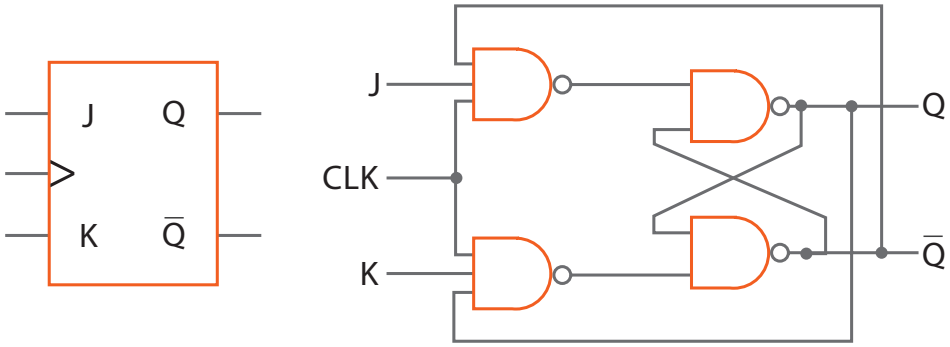


Figura 50. Flip-flop J-K activo con flanco de subida. Fuente: pulse aquí para ver la fuente.

Tabla de funcionamiento				Tabla de excitación				
J	***K***	**Estado**	***Qn + 1***	***Q***	***Qn+ 1***	**Estado**	***J***	***K***
0	0	No cambió	0	0	0	No cambió		
0	1	*Reset*	0	0	1	*Reset*		
1	0	*Set*	1	1	0	*Set*		
1	1	Complemento	1	1	1	No cambió		

Tabla 36. Tabla de funcionamiento y de excitación del flip-flop JK. Fuente: elaboración propia.

5.2.3. Flip-flop T

El *flip-flop T* se obtiene de unir las entradas J y K del flip-flop J-K para generar así la entrada *T*. Con esta unión, sólo pueden aplicarse las combinaciones de entrada *J = 0, K = 0* y *J = 1, K = 1*. Si tomamos la ecuación característica para el flip-flop J-K y hacemos esta conexión, la ecuación se vuelve:

$$Q(t+1) = T\bar{Q} + \bar{T}Q = T \oplus Q.$$

El símbolo para el *flip-flop* T así como su circuito lógico (ver figura 51) se basan en la ecuación anterior y que se obtiene de la tabla 36. La ecuación característica para el *flip-flop T* es simplemente la dada, y la tabla característica muestra que para *T=0*, las salidas del *flip-flop T* permanecen inalteradas, mientras que para *T = 1*, las salidas se complementan. Debido a que el *flip-flop T* sólo puede mantener su salida inalterada o complementar su estado, no existe ninguna manera de establecer un estado inicial haciendo uso únicamente de la entrada *T* sin añadir circuitería externa que indique el valor actual de la salida. Así, el *flip-flop T* normalmente se inicializa a un estado específico en su salida por medio de entradas asíncronas de *set* o un *reset*.

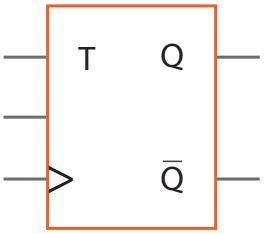


Figura 51. Representación y circuito lógico de Flip-flop T. Fuente: pulse aquí para ver la fuente.

5.2.4. Entradas asíncronas

Muchas veces, los *flip-flops* poseen **entradas especiales** asíncronas de *set* y *reset* (independientes de la entrada de reloj C). Estas entradas ponen de forma asíncrona el *flip-flop* en los estados de *set* o *reset*, y se denominan set asíncrono (o preset) y reset asíncrono (o clear), respectivamente. Si se aplica un 1 lógico (o un 0 lógico si la entrada es negada) en estas entradas, esto afecta a la salida del *flip-flop* si no hay señal de reloj. En el momento en que se activa un sistema digital, los estados iniciales de sus *flip-flops* podrían ser cualquiera. Las entradas asíncronas son útiles para inicializar los *flip-flops* de un sistema digital en un estado inicial conveniente para el funcionamiento con reloj.

En la figura 52 se muestra el símbolo gráfico para un *flip-flop D* disparado por flanco positivo y con entradas directas de *set* y *reset*. Las entradas S y R presentan círculos en sus líneas de entrada indicando que son activas a nivel bajo (es decir, un 0 aplicado producirá la acción de *set* o *reset*). La posición superior e inferior de S y R, respectivamente, implica que los cambios de salida resultantes no se controlan por la señal reloj C. La tabla del funcionamiento del circuito se muestra en la tabla 37 de la página siguiente. Las primeras tres filas de la tabla especifican el funcionamiento de las entradas asíncronas S y R. Estas entradas se comportan como las entradas del *latch* $\overline{S} - \overline{R}$ NAND estudiado anteriormente, operando de forma independientemente al reloj. Las dos últimas filas de la tabla especifican el funcionamiento síncrono para los valores de D. La señal de reloj C se muestra con una flecha ascendente indicando que el flip-flop se dispara por flanco positivo. Los efectos sobre la entrada D son controlados por el reloj de la manera habitual.

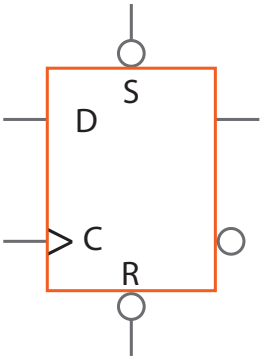


Figura 52. Representación de un flip-flop tipo D con entradas asíncronas. Fuente: pulse aquí para ver la fuente.

S	R	C	D	Estado	Q	$\overline{Q}$
0	1	X	X	*Set*	1	0
1	0	X	X	*Reset*	0	1
0	0	X	X	Indefinido	Indefinido	
1	1	↑	0	Normal	0	1
1	1	↑	1	Normal	1	0

Tabla 37. Tabla de funcionamiento y de excitación del flip-flop D. Fuente: elaboración propia.

5.2.5. Tiempos de los *flip-flops*

Tanto los *flip-flops* disparados por pulso como los activos por flanco (de subida o bajada), llevan asociados a su funcionamiento determinados parámetros de tiempo. Por tanto, a la hora de utilizar los *flip-flops* en un sistema digital, se deben tener en cuenta los tiempos de respuesta a los valores de sus entradas y a la entrada del reloj.

Para ambos tipos de *flip-flops*, se definen los siguientes parámetros:

- Tiempo de *setup* (*ts*): tiempo mínimo previo a la ocurrencia de la transición del reloj que provoca un cambio en la salida, durante el cual las entradas (por ejemplo, S, R o D) deben mantener un valor constante.

- Tiempo de mantenimiento o de *hold* (t_h): es el tiempo posterior a la transición del reloj que genera la salida, durante el cual las entradas del flip-flop tampoco deben cambiar.

- Anchura del pulso de reloj (t_w): duración mínima para la anchura del pulso de reloj que asegura el tiempo suficiente para capturar el valor de la entrada.

La mayoría de los *flip-flops* disparados por pulsos y activos por flanco difieren en el tiempo de *setup ts*. Por lo general, los *flip-flops* disparados por pulso tienen un *ts* igual a la anchura de pulso de reloj t_w, de modo que, considerando que t_s puede ser mucho menor en los *flip-flops* disparados por flanco, estos últimos tienden a ofrecer diseños más rápidos debido a que sus entradas pueden cambiar más tarde con respecto al siguiente flanco del reloj activo.

En los *flip-flops,* los **tiempos de retardo de propagación** se definen como t_{PHL}, t_{PLH} o T_{pd}, que no son más que el intervalo de tiempo que transcurre entre el flanco activo del reloj y la estabilización de la salida hacia un nuevo valor. En la figura 53 de la página siguiente se han designados estos parámetros como t_p– y se proporcionan además sus valores mínimos y máximos. Debido a que los cambios de las salidas de los *flip-flops* van aparte de las entradas de control, el **tiempo de retardo de propagación** t_p mínimo debe ser mayor que el tiempo de mantenimiento para que el funcionamiento sea correcto.

Los parámetros de tiempos se pueden definir de forma similar para los *latches* y para las entradas asíncronas, los cuales incluirían retardos de propagación adicionales necesarios para modelar el comportamiento transparente de los *latches*.

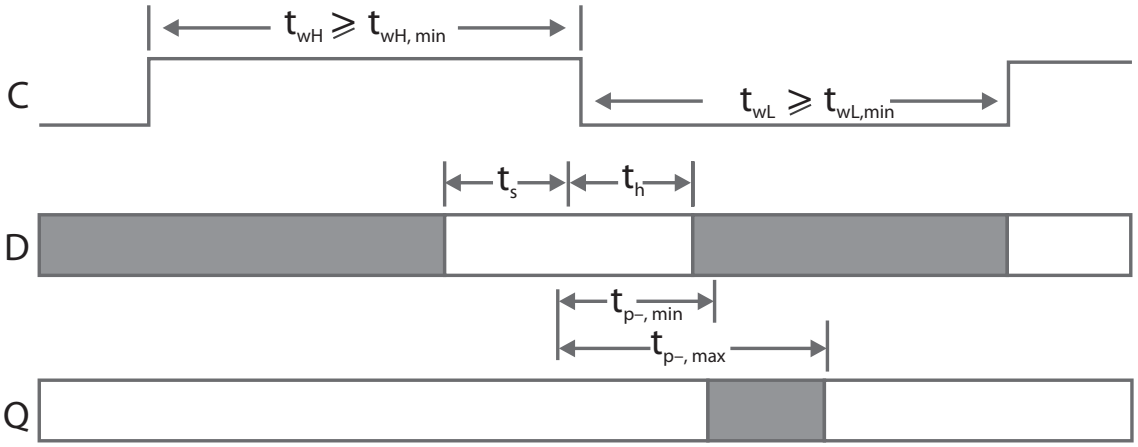


Figura 53. Parámetros de tiempo de un flip-flop. Fuente: Mano y Kime (2005). Parámetros de tiempo de un flip-flop. [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

Tema 6.
Diseño y análisis de circuitos secuenciales síncronos

El funcionamiento de un circuito secuencial está determinado por el estado se sus entradas, salidas, y por el estado actual del circuito. Las salidas y su estado futuro dependen tanto de su estado actual como del estado de las entradas. Cuando hacemos el análisis de un circuito secuencial, estamos describiendo de forma convenientemente cuál será la evolución a lo largo del tiempo de sus entradas, salidas y estados.

Por lo general, se dice que un circuito secuencial es síncrono si su diagrama lógico está constituido por *flip-flops* cuyas entradas de reloj están conectadas directamente o indirectamente a una señal de reloj, y si las entradas directas de *Set* y *Reset* no se utilizan durante el funcionamiento normal del circuito. Los *flip-flops* pueden ser de cualquier tipo del anteriormente visto, y el diagrama lógico puede también incluir puertas combinacionales. En este apartado veremos una representación algebraica para especificar el diagrama lógico de un circuito secuencial. Se verán además tanto la tabla como el diagrama de estado que describen el comportamiento global del circuito.

6.1. Análisis de circuitos secuenciales

6.1.1. Ecuaciones de entrada

Para obtener el diagrama lógico de un circuito secuencial es necesario conocer el tipo de *flip-flops* a emplear y una lista de funciones booleanas para el circuito combinacional. El conjunto de funciones

booleanas asociada al circuito combinacional que genera los valores de entrada de los *flip-flops* se denomina *ecuaciones de entrada a los flip-flops*. Las *ecuaciones de entrada de los flip-flops* son una expresión algebraica conveniente para especificar el diagrama lógico de un circuito secuencial. Indican el tipo de *flip-flop* a partir del símbolo o letra empleado, y especifican completamente el circuito combinacional que controla los *flip-flops*. Aunque el tiempo no se incluye en estas ecuaciones, éste va implícito en el reloj y en la entrada *C* de los *flip-flops*. La figura 51 muestra un circuito secuencial que contiene dos *flip-flops* tipo *D*, una entrada *X* y una salida *Y*, que puede especificarse por las siguientes ecuaciones:

$$D_A = AX + BX$$
$$D_B = \bar{A}X$$
$$Y = (A + B)\bar{X}$$

Como puede apreciarse, las dos primeras ecuaciones representan las entradas de los dos *flip-flops* mientras que la tercera especifica la salida *Y*. Nótese como las dos ecuaciones asociadas a las entradas utilizan el símbolo *D*, coincidiendo con el símbolo de la entrada de los *flip-flops*. Los subíndices *A* y *B* designan las respectivas salidas de los *flip-flops*.

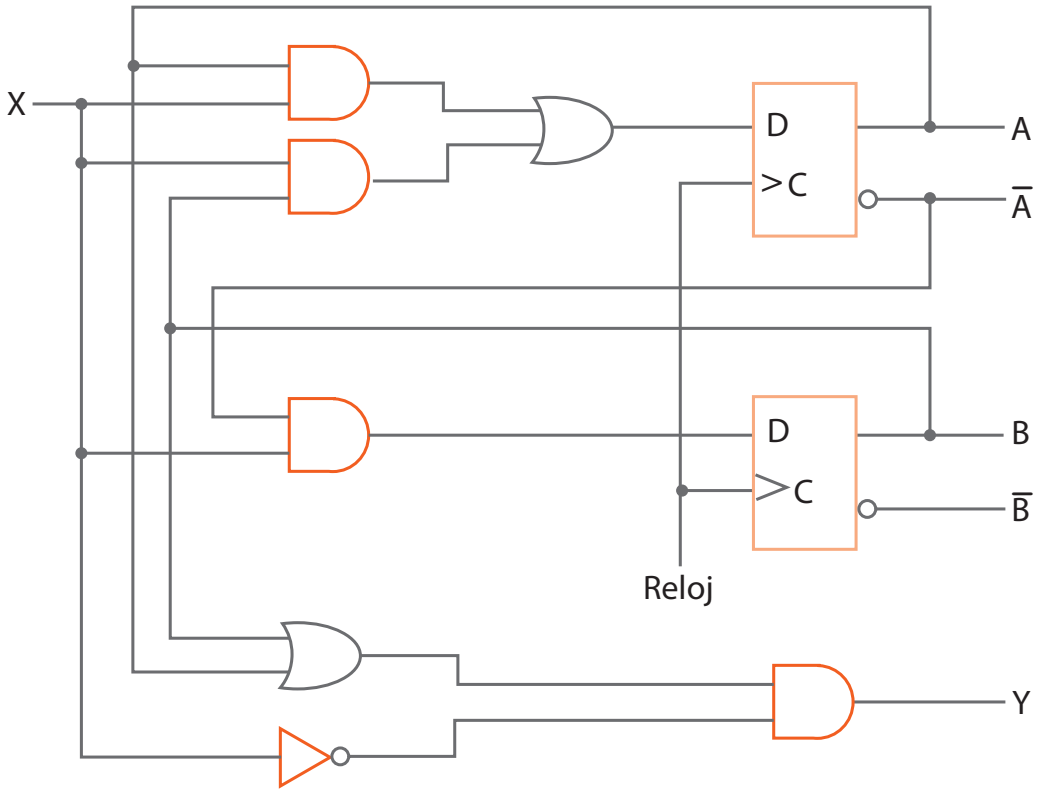


Figura 54. Ejemplo de un circuito secuencial. Fuente: Mano y Kime (2005). Parámetros de tiempo de un flip-flop. [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

6.1.2. Tabla de estados

Las relaciones funcionales entre las entradas, salidas, y los estados de los *flip-flops* de un circuito secuencial pueden especificarse en una tabla de estados (o de transiciones). En la tabla 38 de la página siguiente se muestra la tabla de estados para el circuito de la figura 54. Dicha tabla contiene cuatro

secciones denominadas *estado actual*, *entradas*, *estado futuro*, y *salida*. En la sección *estado actual* se muestran los estados de los *flip-flops* A y B para cualquier instante de tiempo t. La sección *entrada* especifica cada valor de X para cada posible *estado actual*. Note que los *estados actuales* aparecen repetidos para cada posible combinación de la entrada. Por otro lado, la sección *estado futuro* define el estado de los *flip-flops* después de pasar un periodo de reloj, en el momento, es decir, en $t + 1$. La sección *salida* especifica el valor de Y en el instante de tiempo t para cada combinación de *estado actual* y *entrada*.

Estado actual		Entrada	Estado futuro		Salida
A	***B***	***X***	***A***	***B***	***Y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Tabla 38. Tabla de estado del circuito de la figura 54. Fuente: elaboración propia.

Para obtener una tabla de estado primeramente se deben listar todas las posibles combinaciones binarias de estados actuales y entradas. Como puede verse, en la tabla 38 hay ocho combinaciones binarias, desde 000 hasta 111. Posteriormente se determinan los estados futuros en función del diagrama lógico o de las ecuaciones de entrada de los *flip-flops*. Para un *flip-flop* D, se mantiene la relación $A\ (t + 1) = D_A\ (t)$. Esto indica que el estado futuro del *flip-flop* A será el valor actual de la entrada D. Dicho valor está especificado en la ecuación de entrada del *flip-flop* como una función del estado actual de A y B y una entrada X. Además, el estado futuro del *flip-flop* A debe satisfacer la ecuación:

$$A(t + 1) = D_A = AX + BX$$

La sección *estado futuro* de la tabla, si observamos la columna A, contiene tres filas con 1 lógico donde el estado actual y el valor de la entrada satisfacen las condiciones $(A, X) = 11$ o $(B, X) = 11$. Igualmente, el estado futuro del *flip-flop* B se obtiene de la ecuación de entrada:

$$B(t + 1) = D_B = \bar{A}X$$

La cual toma valor 1 cuando el estado actual de A es 0 y la entrada X es igual a 1. La columna de salida Y se obtiene de la ecuación de salida:

$$Y = A\bar{X} + B\bar{X}$$

Para cualquier circuito secuencial diseñado con *flip-flops* del tipo D, *su tabla de estado* se obtendrá como hemos visto anteriormente. En general, un circuito secuencial con m *flip-flops* y n entradas necesita 2^{m+n} filas en su *tabla de estado* y se constituye de la siguiente manera:

- Se listan las combinaciones binarias de 0 hasta $2^{m+n} - 1$ combinando las columnas de *entrada* y de *estado actual*.
- La sección *estado futuro* tiene m columnas, una para cada *flip-flop* empleado. Los valores binarios para el estado futuro se obtienen a partir de las ecuaciones de entrada de cada *flip-flop* tipo D.
- La sección *salida* contiene tantas columnas como variables de salida. Sus valores binarios se obtienen del circuito o de las funciones booleanas de la misma manera que en una tabla de verdad.

Hasta ahora hemos visto cómo definir una tabla de estado **unidimensional**, es decir, donde tanto el estado actual como las combinaciones de entrada se combinan en una única columna. Muchas veces se emplea una tabla de estado **bidimensional**, en la cual el estado actual se coloca en la columna izquierda y las entradas en la fila superior. El estado futuro se coloca en cada celda de la tabla según la combinación correspondiente del estado actual y de la entrada. Si las salidas dependen de las entradas, entonces se emplea una **tabla bidimensional** similar. La tabla 39 muestra cómo quedaría la tabla de estado bidimensional para el circuito de la figura 54.

Estado actual		Estado futuro				Salida	
		$X = 0$		$X = 1$		$X = 0$	$X = 1$
A	***B***	***A***	***B***	***A***	***B***	***Y***	***Y***
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

Tabla 39. Tabla de estado bidimensional del circuito de la figura 54. Fuente: elaboración propia.

Los circuitos secuenciales se pueden clasificar en dos tipos en función de la dependencia de sus salidas como veremos a continuación.

- **Autómatas o Máquinas de Mealy:** son aquellos circuitos secuenciales en los que las salidas dependen de las entradas y de los estados actuales de los *flip-flops*.

$$Z_k = Z_k(x_1, x_2, \dots, q_1, q_2, \dots)$$

  Las salidas de una **máquina de Mealy** pueden cambiar cuando lo hagan las salidas q_i de los *flip-flops*, o cuando lo hagan las entradas al circuito x_i. Debido a que las entradas pueden cambiar en cualquier momento independientemente de la señal de reloj, las salidas del autómata pueden hacer lo mismo en cualquier momento.

- **Autómatas o Máquinas de Moore:** son aquellos circuitos secuenciales donde sus salidas sólo dependen de los estados actuales, es decir, de los valores almacenados en los *flip-flops*, de ahí que bastaría con una única columna unidimensional.

$$Zk = Z_k(q_1, q_2, \dots)$$

Las salidas de una **máquina de Moore** cambian de valor únicamente cuando lo hacen las salidas q_i de los *flip-flops*, o sea, cuando se producen los flancos activos (ascendentes o descendentes), mientras que en el resto del ciclo permanecen constantes.

6.1.3. Diagrama de estados

El **diagrama de estado** puede ser un punto opcional en el análisis de circuitos secuenciales. Simplemente consiste en trasladar la información contenida en la tabla de estados a una representación gráfica. En este tipo de diagrama, los estados se representan con un círculo, mientras que las transiciones entre los diferentes estados se indican mediante flechas que conectan los círculos, orientadas en la dirección apropiada. En la figura 55 se muestra el diagrama de estado asociado al circuito secuencial de la figura 54 y su tabla de estados definida en la tabla 38. Los números binarios que se encuentran en el interior de cada círculo identifican los estados de los *flip-flops*. En los **Autómatas de Mealy**, las flechas se etiquetan con dos números binarios separados por una barra (/), siendo el número de la izquierda el valor de entrada durante el estado actual, y el de la derecha el valor de la salida durante el estado actual aplicando dicha entrada. Por ejemplo, si observamos la figura 55, la flecha que va del estado 00 al 01 se etiqueta 1/0, lo que indica que cuando el circuito secuencial se encuentra en el estado actual 00 y la entrada es 1, su salida es 0. Llegada la próxima transición del reloj, el circuito pasa al siguiente estado, en este caso 01. Si la entrada cambiara a 0, entonces la salida se pone a 1, pero si la entrada permaneciese en 1, la salida seguiría en 0. Esta información se obtiene a partir del diagrama de estados, durante las transiciones de las dos flechas que parten del círculo con estado 01. Cuando una flecha conecta un círculo consigo mismo, indica que no se produce ningún cambio de estados.

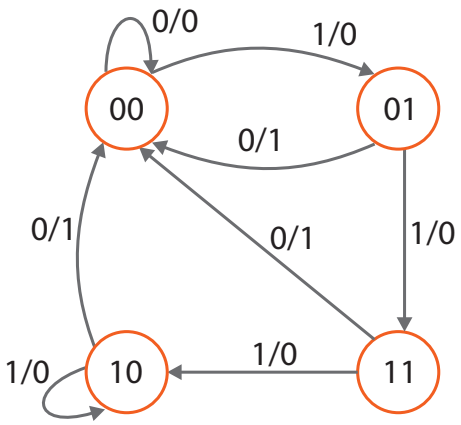


Figura 55. Diagrama de estados para el circuito de la figura 54. Fuente: Mano y Kime (2005). Parámetros de tiempo de un flip-flop. [Figura]. Recuperado de Fundamentos de diseño lógico y computadoras (3rd ed.). Madrid: Pearson Prentice-Hall.

6.2. Diseño de circuitos secuenciales

El proceso para el diseño de circuitos secuenciales síncronos requiere de un proceso inverso al del análisis visto en el apartado anterior. Primero se define una serie de especificaciones que modelen su funcionamiento y se culmina con un diagrama lógico o una lista de funciones booleanas a partir de las cuales se puede obtener el diagrama lógico. A diferencia de un circuito combinacional que se especifica completamente por medio de una tabla de verdad, un circuito secuencial necesita de una tabla de estados para su especificación. Por lo tanto, el primer paso en el diseño de un circuito secuencial es obtener su tabla de estados o en su lugar, el diagrama de estados.

Como hemos visto hasta ahora, el diseño de un circuito secuencial síncrono se realiza a partir de *flip-flops* y de compuertas lógicas. El diseño se basa en elegir tanto los *flip-flops* como la circuito combinacional adecuados que, en su conjunto, generen un circuito que cumpla las especificaciones requeridas. Una vez determinado el tipo y el número de *flip-flops*, el proceso del diseño pasa de ser un problema de circuitos secuenciales a un problema de circuitos combinacionales, pudiendo entonces aplicarse las técnicas de diseño de circuitos combinacionales. Los pasos a seguir para el diseño de un circuito secuencial síncrono son similares a los de un circuito combinacional, pero con algunos pasos adicionales que se resume a continuación:

- **Especificación:** definir la especificación del circuito.
- **Formulación:** obtener el diagrama de estados o tabla de estados a partir de la especificación del problema.
- **Asignación de estados:** si sólo se dispone del diagrama de estados, se debe obtener la tabla de estados y asignar los códigos binarios a los estados de la tabla.
- **Determinar la ecuación de entrada al *flip-flop*:** seleccionar el tipo de *flip-flop* a emplear y a partir de la tabla de estados, obtener las ecuaciones de entrada de los *flip-flops* de las entradas codificadas del estado futuro.
- **Determinar de la ecuación de salida:** obtener las ecuaciones de salida a partir de las salidas especificadas en la tabla de estados.
- **Optimización:** optimizar las ecuaciones de entrada y salida de los *flip-flops*.
- **Comprobación:** verificar el funcionamiento correcto del diseño final del circuito.

A la hora de diseñar un circuito secuencial síncrono, el paso más difícil y sobre el que no existe ninguna metodología específica a aplicar es el primero, es decir, obtener un diagrama de estados que modele el funcionamiento de la máquina secuencial. Una vez que se obtiene dicho diagrama, los siguientes pasos son más o menos "mecánicos". De todas maneras, durante el proceso de diseño, siempre es deseable que el circuito final tenga un coste reducido, por lo que tendremos que aplicar métodos de simplificación que permitan obtener un circuito óptimo. Se deja al estudiante profundizar en este tema a través del seguimiento de varios ejemplos explicados paso a paso en (Mano & Kime, 2005) y (Molina, Díaz, & Escudero, 2004).

Tema 7.
Introducción al lenguaje ensamblador

Hasta ahora hemos visto como diseñar sistemas digitales sencillos, ya sean sistemas combinacionales o secuenciales. En este capítulo veremos específicamente la arquitectura de conjunto de instrucciones de procesadores de propósito general. Estudiaremos las operaciones que realizan las distintas instrucciones, centrándonos particularmente en cómo se obtienen los operandos y dónde se almacenan los resultados. Las instrucciones básicas la clasificaremos en tres categorías: transferencia de datos, manipulación de datos y control de programa. Para cada categoría detallaremos las instrucciones básicas comúnmente empleadas.

7.1. Lenguaje máquina

El **lenguaje máquina** o código máquina no es más que el lenguaje binario en el que se definen y almacenan las **instrucciones** en la memoria, es decir, operaciones expresadas mediante la codificación binaria de cadenas de 1's y 0's. El lenguaje máquina es distinto para cada computador, excepto cuando existe compatibilidad entre las familias de procesadores. Por otro lado, el lenguaje que sustituye a los códigos de operación binarios y las direcciones con **nombres simbólicos** o **mnemónicos** y que proporciona otras características útiles al programador se le denomina **lenguaje ensamblador.**

Habitualmente un procesador tiene una gran variedad de instrucciones y formatos de éstas. La función de la **unidad de control** es descodificar cada instrucción y proporcionar las señales de control necesarias para que estas sean ejecutadas.

7.2. Lenguaje ensamblador

7.2.1. Formato de Instrucciones

El formato de una instrucción se representa en forma de rectángulo simbolizando los bits de la instrucción en código binario (ver figura 56). Los bits se dividen en grupos llamados campos. Los campos típicos que se encuentran en los formatos de las instrucciones son los siguientes:

- Campo de **código de operaciones:** especifica la operación a realizar, ya sea aritmética, lógica, de transferencia, etc.
- Campo de **direcciones:** proporciona direcciones de la memoria o direcciones para un registro del procesador.
- Campo de **modo:** especifica la forma en que se interpreta el campo de direcciones.

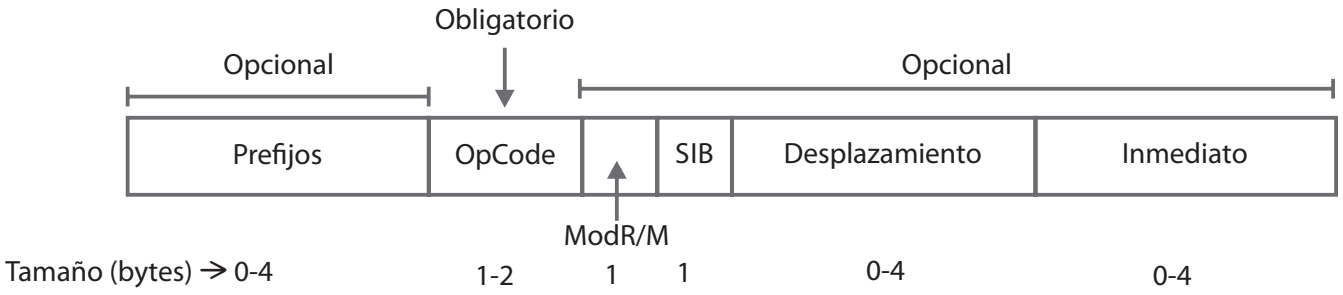


Figura 56. Diferentes campos que conforman una instrucción. Fuente: pulse aquí para ver la fuente.

Las instrucciones pueden diferir en función del número de operandos que estas direccionan explícitamente en una operación de datos, de ahí que sea un factor importante a la hora de definir la arquitectura de conjunto de instrucciones de un procesador. Cada instrucción contiene explícitamente o implícitamente toda la información que necesita para ejecutarse tal y como se mostró en la figura 56.

Muchas veces se emplean campos especiales, por ejemplo, para especificar el número de posiciones que se desplazará un dato en una instrucción de desplazamiento, o un campo de operando para las instrucciones de operando inmediato. La unidad de control de un procesador está diseñada para ejecutar cada una de las instrucciones de un programa por medio de los siguientes de pasos:

1) Lectura de memoria de la instrucción por un registro de control.
2) Interpretación (descodificación) de la instrucción.
3) Ejecución de la operación en los registros del procesador (bajo las señales generadas por la unidad de control).

4) Almacenamiento del resultado.

5) Retorno al paso 1 para traer la siguiente instrucción, actualizar el contador de programa (CP).

En el procesador existe un registro que se denomina **contador de programa** (CP), el cual se encarga de seguir la pista de las instrucciones del programa almacenado en la memoria cuando éste se está ejecutando. El PC contiene la dirección de la instrucción que se va a ejecutar a continuación y se incrementa en uno cuando se lee una nueva palabra o instrucción del programa almacenado en la memoria. Los procesadores tienen además del PC, el llamado banco de registros que incluye **registros de datos**, **punteros** y **registros de segmentos de memoria**. Los más comunes se listan a continuación.

Registros de datos:

- AX (AH, AL) → Acumulador, operaciones de E/S y aritmética.
- BX (BH, BL) → Base, registro de propósito general (indexación y cálculos).
- CX (BH, BL) → Contador: puede contener cuantas veces se repite un ciclo.
- DX (BH, BL) → Datos: algunas operaciones de E/S y en la multiplicación y división de números grandes (trabajando junto con AX).

Punteros:

- SP → puntero de pila (*stack pointer*).
- BP → puntero base de pila (facilita la referencia de datos y direcciones vía pila).
- SI → registro índice (operaciones con cadenas de caracteres asociados con DS).
- DI → registro índice (operaciones con cadenas de caracteres asociados con ES).

Registros de segmentos:

- SS → Segmento de pila.
- DS → Segmento de datos.
- ES → Segmento extra de datos.
- CS → Segmento de código.

Por ejemplo, en el caso de la familia de microprocesadores 80 x 86/88, se puede direccionar 1MB con 20 líneas de dirección (2^{20} = 1.048.576 B), pero sus registros internos tan solo son de 16 bits, de ahí que la solución consiste en hacer una **segmentación de la memoria**.

Las direcciones se generan combinando una base y un desplazamiento, cada uno de 16 bits:

Dirección física = base x 10h + desplazamiento

Cada base genera una página o segmento de memoria de 64 Kb, cada una con funciones específicas:

Base Función

CS → Contiene el código ejecutable.

SS → Se reserva para la pila (*stack*).

DS → Contiene los datos.

ES → Segmento extra de datos.

La figura 57 muestra el diagrama general del hardware real de un procesador de la familia 80x86/88 con los registros mencionados anteriormente.

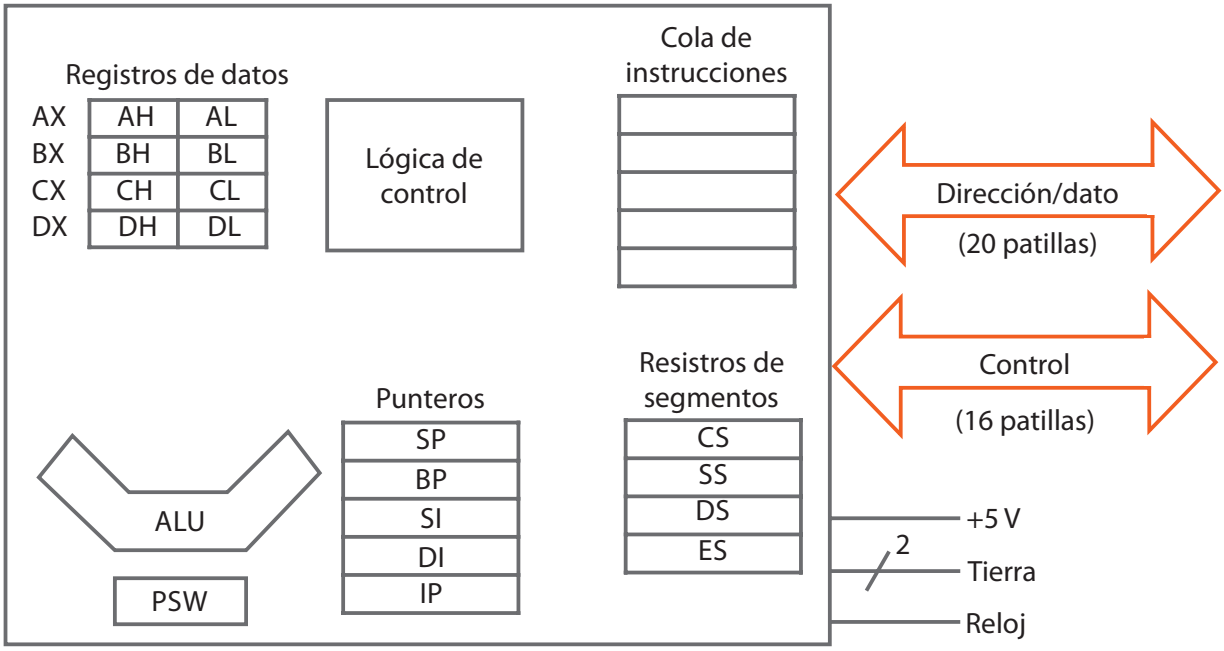


Figura 57. Ejemplo del hardware real de un microprocesador 80 x 86/88. Fuente: pulse aquí para ver la fuente.

7.2.2. Juego de instrucciones

En función de la operación que realicen, las instrucciones se pueden clasificar en varios tipos según la operación que realizan. A continuación veremos algunas de las instrucciones más comunes para cada uno de los tipos de instrucciones que existen.

Instrucciones de transferencia: copian en el operando destino la información del operando fuente sin modificarla. Además no modifican el estado de los *flags* (códigos de condición o banderas) y aunque generalmente transfieren palabras, pueden también mover fracciones de ellas o bloques enteros. En la tabla 40 se listan las ocho transferencias típicas utilizadas en muchos procesadores. A cada instrucción le acompaña su símbolo mnemónico o **nombre simbólico**, abreviatura de lenguaje ensamblador recomendada por el estándar IEEE. Sin embargo, no todos los procesadores emplean los mismos mnemónicos para la misma instrucción, pudiendo cambiar entre diferentes familias.

Nombre	Mnemónico	Operación
Load	LD	Transferencia desde la memoria a un registro del procesador.
Store	ST	Transferencia desde un registro del procesador a una posición de memoria.
Move	MOVE	Se emplea en procesadores con varios registros para indicar una transferencia desde un registro del procesador a otro, o para la transferencia de datos entre registros y memoria y entre dos posiciones de memoria.
Exchange	XCH	Intercambio de datos entre dos registros, entre un registro y una posición de memoria, o entre dos posiciones de memoria.
Push	PUSH	Coloca un nuevo objeto en la posición más alta de la pila.
Pop	POP	Borra un objeto de la pila de forma que el contenido de la pila sube.
Input	IN	Transfieren datos entre los registros de procesador y los dispositivos de E/S. Incluyen un campo de direcciones en su formato para especificar el puerto seleccionado para la transferencia de datos.

Tabla 40. Instrucciones de transferencia de datos. Fuente: elaboración propia.

Instrucciones de manipulación de datos: en un procesador típico, las instrucciones de manipulación de datos se agrupan en tres tipos básicos:

1. Instrucciones aritméticas.
2. Instrucciones lógicas y de manipulación de bits.
3. Instrucciones de desplazamiento.

Las **operaciones aritméticas** básicas son la suma, la resta, la multiplicación y la división. Muchos procesadores poseen instrucciones para realizar estas cuatro operaciones. Sin embargo, existen algunos procesadores pequeños que sólo tienen instrucciones de suma y resta, por lo que la multiplicación y la división se deben implementar mediante programas.

En la tabla 41 de la página siguiente se resume la lista de las instrucciones aritméticas típicas. En el caso de las operaciones INC y DEC, cuando se ejecutan en una palabra del procesador, si el número binario tiene todos sus bits a 1 o todos a 0, se produce como resultado un número con todos sus bits a 0 o todos 1, respectivamente.

Nombre	Mnemónico	Operación
Incremento	INC	Incremento en 1 del valor almacenado en el registro o en una posición de la memoria.
Decremento	DEC	Decremento en 1 del valor almacenado en el registro o en una posición de la memoria.
Suma	ADD	Las operaciones suman, resta, sustracción, multiplicación y división pueden estar disponibles para diferentes tipos de datos. El tipo de dato involucrado se supone que se incluye en la definición del código de operación.
Resta	SUB	
Multiplicación	MUL	
División	DIV	
Suma con acarreo	ADDC	Realiza la operación suma de dos operandos más el valor del acarreo del cálculo anterior.
Resta con acarreo	SUBB	Realiza la sustracción de dos operandos más el valor del acarreo generado en la operación anterior.
Resta Inversa	SUBR	Se invierte el orden de los operandos, realizando la operación B-A en lugar de A-B.
Negación	NEG	Complemento a 2 de un número con signo, que es equivalente a multiplicar el número por -1.

Tabla 41. Instrucciones aritméticas. Fuente: elaboración propia.

Las **instrucciones lógicas** realizan operaciones con las palabras almacenadas en registros o en la memoria. Permiten tanto la manipulación individual como de grupos de bits que representan información codificada en binario. Las instrucciones tratan cada bit por separado como si fuesen variables booleanas. La tabla 42 presenta algunas funciones lógicas típicas.

Nombre	Mnemónico	Operación
Clear o puesta a 0	CLR	Hace que el operando especificado sea remplazado por 0's.
Set o puesta a 1	SET	Hace que el operando especificado sea remplazado por 1's.
Complemento	NOT	Invierte el valor de todos los bits del operando.
AND	AND	Realizan la operación lógica correspondiente en los bits del operando de manera individual según se especifique.
OR	OR	
OR exclusiva	XOR	
Pone acarreo en 0	CLRC	Puesta a 0, 1 o complemento del bit de acarreo, son operaciones que afectan los bits de estados.
Pone acarreo en 1	SETC	
Complemento del acarreo	COMC	

Tabla 42. Instrucciones lógicas. Fuente: elaboración propia.

Cuando las instrucciones lógicas se efectúan en palabras, se suelen considerar como operaciones de manipulación de bits. Hay tres posibles operaciones para manipular bits: poner un bit seleccionado a 0, ponerlo a 1 o complementarlo. Por ejemplo, la instrucción AND se usa para poner un bit o un grupo de bits a 0, debido a que cualquier variable booleana multiplicada (AND) por 0 produce un 0. Por tanto, la instrucción AND se utiliza para poner los bits de un operando a 0 de forma selectiva, empleando una palabra que contenga 0 en las posiciones concretas que se desean poner a 0, y 1 en el resto de posiciones o bits que se desean mantener intactas. Muchas veces a la operación AND se le denomina operación máscara, debido a que enmascara una parte del operando. Al contrario de la instrucción AND, la instrucción OR se utiliza para poner un bit o un grupo de bits de un operando a 1 de forma selectiva, usando una palabra que contenga 1's en la posiciones de los bits que se quieren poner a 1 y 0's en el resto. Por su parte, la instrucción XOR se utiliza para complementar selectivamente los bits de un operando, debido a que la operación XOR con 1 hace que cambie el valor de las posiciones de los bits especificadas, pero no cambia si la operación XOR se hace con 0. A veces, a la instrucción XOR se le denomina instrucción de complemento de bit.

Las instrucciones de desplazamiento permiten desplazar el contenido de un operando de diversas maneras. Los desplazamientos no son más que operaciones donde los bits del operando se desplazan hacia la izquierda o hacia la derecha. Estas operaciones pueden especificar desplazamientos **lógicos**, **aritméticos** u operaciones de **rotación**. A continuación se enumeran los distintos tipos de instrucciones de desplazamiento en la tabla 43.

Mnemónico	Operación
SHR	Desplazamiento lógico a la derecha.
SHL	Desplazamiento lógico a la izquierda.
SHRA	Desplazamiento aritmético a la derecha.
SHLA	Desplazamiento aritmético a la izquierda.
ROR	Rotación a la derecha.
ROL	Rotación a la izquierda.
RORC	Rotación a la derecha con acarreo.
ROLC	Rotación a la izquierda con acarreo.

Tabla 43. Instrucciones de desplazamiento. Fuente: elaboración propia.

Cuando se realiza un desplazamiento lógico, se introduce un 0 en la posición del bit entrante una vez hecho el desplazamiento. En el caso de desplazamiento aritmético hacia la derecha, el bit de signo se mantiene en la posición más a la izquierda. Este valor se desplaza hacia la derecha junto con el resto de dígitos pero manteniendo el bit de signo sin cambiar. El desplazamiento aritmético a la izquierda es idéntico a la instrucción de desplazamiento lógico hacia la izquierda, e introduce un 0 como bit entrante en la posición más a la derecha. La diferencia entre las dos instrucciones se basa en que, en el caso del desplazamiento aritmético a la izquierda, el bit de *status overflow V* se puede poner a 1, mientras que el desplazamiento lógico a la izquierda V no queda afectado.

Las instrucciones de rotación ejecutan un desplazamiento circular, es decir, el valor correspondiente al bit de salida de la palabra no se pierde, sino que se introducen por el bit entrante. Las instrucciones de rotación con acarreo tratan al bit de acarreo como una extensión del registro cuya palabra está siendo rotada. De esta forma, una rotación a la izquierda con acarreo transfiere el bit de acarreo al bit de entrada a la posición más a la derecha del registro, transfiere el bit saliente desde el bit más a la izquierda del registro al acarreo y desplaza el contenido completo del registro a la izquierda. Algunos procesadores tienen un formato con varios campos para la instrucción de desplazamiento. Un campo contiene el código de operación y el resto especifica el tipo de desplazamiento y el número de posiciones que el operando se va a desplazar.

Instrucciones de control de programa: afectan el valor de la dirección del PC y pueden alterar el flujo de control. Un cambio en el PC debido a la ejecución de una instrucción de control de programa, provoca una ruptura o salto en la secuencia de ejecución de las instrucciones. Esta funcionalidad en los procesadores digitales es muy importante ya que proporciona un control sobre el flujo de ejecución del programa, pudiéndose desviar o bifurcarse hacia distintos segmentos del programa según se requiera.

En la tabla 44 se listan algunas instrucciones de control de programa típicas. La **bifurcación** o ramificación (del término inglés branch) y el **salto** (del término inglés jump) significan la misma cosa, aunque a veces se usan con modos de direccionamiento diferentes. Estas a su vez pueden ser condicionales o incondicionales. La instrucción de **salto** implícito (*skip*) no necesita campo de direcciones. Una instrucción de salto implícito condicional saltará a la siguiente instrucción de la secuencia si se cumple una determinada condición, quedándose ésta sin ejecutar. Si no se cumple, el control prosigue con la siguiente instrucción de la secuencia, donde el programador puede insertar una instrucción de ramificación incondicional. De esta forma, una instrucción de salto implícito seguida de una instrucción de bifurcación provoca una ramificación si la condición no se cumple.

Nombre	Mnemónico
Bifurcación	BR
Salto	JMP
Salto implícito (*Skip*)	SKP
Llamada a subrutina	CALL
Retorno de subrutina	RET
Comparación (con substracción)	CMP
Test (mediante AND)	TEST

Tabla 44. Instrucciones de control de programa. Fuente: elaboración propia.

Las instrucciones de **llamada** y de **retorno** se emplean para usar subrutinas. Estas instrucciones de salto a la subrutina y posterior regreso al programa principal van emparejadas entre sí. Una **subrutina** no es más que una secuencia de instrucciones que realizan una tarea de cálculo concreta. Cuando se ejecuta un programa, se puede llamar a una subrutina concreta varias veces desde distintos puntos del programa. Siempre que se llama a una subrutina se produce una bifurcación al comienzo de ésta.

Una vez ejecutada en su totalidad, se realiza una nueva bifurcación para volver al programa principal justo donde se quedó cuando se hizo la llamada.

La instrucción de **comparación** realiza una resta para comparar pero sin almacenar el resultado, pero en cambio, realiza una bifurcación condicional cambiando el contenido de un registro, o poniendo a 0 o a 1 los bits de *status*. Igualmente, la instrucción de **test** efectúa una multiplicación lógica (operación AND) entre dos operandos sin almacenar el resultado, y ejecuta una de las acciones enumeradas en la instrucción de comparación.

Existen también **instrucciones de bifurcación condicional**, la cuales se reagrupan de la siguiente manera:

- Relacionadas con los bits de status del PSR.
- Condicional para números sin signo.
- Condicional para números con signo.

En el caso del primer grupo, cada instrucción de bifurcación condicional comprueba una combinación específica de los bits de *status* según la condición. Si la condición es verdadera, el control se transfiere a la dirección efectiva. Si es falsa, el programa continúa con la próxima instrucción. En la tabla 45 se listan las instrucciones condicionales que dependen directamente de los bits del registro de *status* del procesador (PSR). El mnemónico se constituye normalmente con la letra **B** (*branch*) y una letra asociada al bit de *status* involucrado. La letra **N** (de *not*) se incluye si se comprueba la condición igual a 0 para el bit de *status* involucrado. Los bits de status del PSR típicamente son: el acarreo (**C**), cero (**Z**), *overflow* o desbordamiento (**V**) y signo (**N**). De esta forma, BC efectúa una ramificación si el acarreo es igual a 1, y BNC hace una ramificación si el bit de acarreo es igual a 0.

Condición de bifurcación	Mnemónico	Condición de test
Bifurcación si es cero.	BZ	Z = 1
Bifurcación si no es cero.	BNZ	Z = 0
Bifurcación si hay acarreo.	BC	C = 1
Bifurcación si no hay acarreo.	BNC	C = 0
Bifurcación si negativo.	BN	N = 1
Bifurcación si positivo.	BNN	N = 0
Bifurcación si hay *overflow*.	BV	V = 1
Bifurcación si no hay *overflow*.	BNV	V = 0

Tabla 45. Instrucciones de bifurcación relacionadas con los bits de status. Fuente: elaboración propia.

Las instrucciones de bifurcación condicional con **números sin signo** se listan en la tabla 46 de la página siguiente. Las palabras **mayor**, **menor** e **igual** se usan para indicar la diferencia entre dos números sin signo. Si dos números son iguales, es decir A = B, se podrá determina a partir del bit de status Z, donde Z será 1 si A – B = 0. Si A es menor que B y el acarreo C = 1, entonces A < B. Para que

se cumpla que A sea menor o igual que B ($A \leq B$), los bits C o Z deben ser 1 ($C = 1$ o $Z = 1$). La relación $A > B$ es lo contrario de $A \leq B$ y se detecta complementando la condición de los bits de status. Igualmente, la condición $A \geq B$ es la inversa de $A < B$, y la condición $A \neq B$ es la inversa de $A = B$.

Condición de bifurcación	Mnemónico	Condición	Bits de status
Bifurcación si es mayor.	BH	$A > B$	$C + Z = 0$
Bifurcación si es mayor o igual.	BHE	$A \geq B$	$C = 0$
Bifurcación si es menor.	BL	$A < B$	$C = 1$
Bifurcación si es menor o igual.	BLE	$A \leq B$	$C + Z = 1$
Bifurcación si es igual.	BE	$A = B$	$Z = 1$
Bifurcación si no es igual.	BNE	$A \neq B$	$Z = 0$

Tabla 46. Instrucciones de bifurcación condicional para números sin signo. Fuente: elaboración propia.

Para el caso de **números con signo,** la tabla 47 lista las instrucciones correspondientes. Igual que en las comparaciones de números sin signo, aquí se supone que los bits de *status* N, V y Z se han actualizado después de la operación A – B. Las palabras **más grande**, **menos grande** e **igual** se usan para indicar la diferencia entre dos números **con signo**. Si el bit de *status* N es igual a 0, el signo de la resta es positivo y A debe ser ≥ B, haciendo $V = 0$ e indicando que no ha ocurrido *overflow*. La ocurrencia de un *overflow* genera un cambio de signo. Esto indica que si $N = V = 1$, se produjo un cambio de signo y el resultado fue positivo, lo cual establece que A es más grande o igual que B. Además, la condición $A \geq B$ será cierta si $N = V = 0$, o si $N = V = 1$.

Para que se cumpla la condición de que A sea mayor que B pero no igual ($A > B$), el resultado de la resta debe ser positivo y distinto de cero. Un resultado igual a 0 tiene signo positivo, por lo que se debe chequear que Z sea 0 para excluir la posibilidad de que $A = B$. El resto de condiciones de la tabla se pueden derivar de manera similar. Las condiciones BE (bifurcación si es igual) y BNE (bifurcación si no es igual) especificadas para los números sin signo se aplican también a los números con signo, y se pueden determinar con $Z = 1$ y $Z = 0$, respectivamente.

Condición de bifurcación	Mnemónico	Condición	Bits de status
Bifurcación si es mayor.	BG	$A > B$	$(N \oplus V) + Z = 0$
Bifurcación si es mayor o igual.	BGE	$A \geq B$	$(N \oplus V) = 0$
Bifurcación si es menor.	BL	$A < B$	$(N \oplus V) = 1$
Bifurcación si es menor o igual.	BLE	$A \leq B$	$(N \oplus V) + Z = 1$

Tabla 47. Instrucciones de bifurcación condicional para números con signo. Fuente: elaboración propia.

Interrupciones: las interrupciones suponen una ruptura abrupta en la secuencia normal del programa saltando hacia código que da ese servicio y, cuando se ha terminado, se vuelve a la ejecución del programa donde se quedó antes de producirse la interrupción.

Las interrupciones pueden ser:

- **Interrupciones hardware:** son generadas por los circuitos asociados al microprocesador en respuesta a algún evento como pulsar una tecla del teclado.
- **Interrupciones software:** son generadas por un programa para llamar a ciertas subrutinas almacenadas en memoria ROM o RAM. Es posible cambiarlas y crear otras nuevas.

Los pasos para llamar a una interrupción son: 1) identificar la interrupción ocurrida, pasar los parámetros a la subrutina, llamar a la interrupción. Las interrupciones salvaguardan los flags y los registros que emplean.

La llamada a una interrupción y el posterior retorno desde esta se realiza por medio de los mnemónicos INT e IRET, respectivamente.

7.3. Modos de direccionamiento

El campo de operación de una instrucción especifica la operación a ejecutar. Esta operación debe ejecutarse sobre el dato almacenado en los registros del procesador o en la memoria. Cómo se seleccionan los operandos durante la ejecución del programa depende del **modo de direccionamiento** de la instrucción. El modo de direccionamiento especifica una regla para interpretar o modificar el campo de direcciones de la instrucción antes de que se haga realmente referencia al operando. A la dirección del operando generada mediante la aplicación de esa regla se le llama **dirección efectiva**. Los procesadores utilizan técnicas de modo de direccionamiento para ajustarse a las siguientes características:

1) Proporcionar flexibilidad al usuario en la programación mediante punteros a la memoria, contadores para el control de bucles, indexar datos y reubicar programas.
2) Reducir el número de bits de los campos de direcciones de la instrucción.

Modo inmediato: el operando se encuentra en la propia instrucción, es decir, que la instrucción de modo inmediato tiene un campo de operando en lugar de un campo de direcciones. El campo de operando contiene el operando a utilizar junto con la operación que se especifica en la instrucción. Las instrucciones con modo inmediato son muy utilizadas, por ejemplo, para inicializar registros con un valor constante.

Ejemplo:

```
MOV CX, 0010H
```

Modo directo: en este modo, el campo de direcciones de la instrucción proporciona el lugar (dirección de memoria) donde está el operando, para luego ejecutar una instrucción de transferencia de datos o de manipulación de datos. En función del lugar donde se encuentre el operando, el direccionamiento puede ser **directo a registro** o **directo a memoria**.

Ejemplos:

```
MOV AX, BX

MOV CX, Etiqueta
```

Modo indirecto: la posición o campo de direcciones de la instrucción no se refiere al operando sino la dirección de memoria en la que éste se encuentra (dirección efectiva), por lo que se necesita un acceso adicional a memoria. La dirección de memoria se puede dar mediante direccionamiento directo a memoria o direccionamiento relativo. Su uso más común es el acceso a diversas informaciones mediante tablas de punteros.

Ejemplos:

```
MOV AL, [[100]]

MOV CL, [[B + 1234h]]
```

Modo relativo: la instrucción indica el desplazamiento del operando con respecto a un puntero. La dirección efectiva es calculada por la unidad de control, sumando o restando, el desplazamiento al puntero de referencia que suele estar en un registro.

Dirección efectiva = Reg. Referencia + desplazamiento

Ejemplos:

```
MOV AL, [BX]

MOV CH, Número[SI]

MOV BL, [SP+4]
```

Dependiendo del puntero se tienen diferentes modos de direccionamiento, los cuales se resumen en la tabla 48.

Modo de direccionamiento	Registro de referencia	Dirección efectiva
Relativo a contador de programa.	Contador de programa (CP).	DE = CP + desplazamiento.
Relativo a registro base.	Un registro base (RB).	DE = RB + desplazamiento.
Relativo a registro índice.	Un registro índice (RI).	DE = RI + desplazamiento.
Relativo a pila.	Registro de pila (SP).	DE = SP + desplazamiento.

Tabla 48. Diferentes modos de direccionamiento relativo. Fuente: elaboración propia.

Modo implícito: en la instrucción no se indica explícitamente el lugar donde se encuentra el operando. Requiere que el programador conozca con que operandos se está trabajando. Por ejemplo, la instrucción **complementar el acumulador,** tiene un modo implícito ya que el operando del registro acumulador queda implícito en la definición de la instrucción. De hecho, cualquier instrucción que

utiliza el registro acumulador sin un segundo operando es una instrucción con modo implícito. Igualmente, las instrucciones que manipulan datos en la pila del procesador, como **ADD**, son instrucciones con modo implícito, ya que los operandos están implícitamente en la posición superior de la pila.

Ejemplo:

MUL BX → DX, AX = AX x BX

Donde AX y DX son operandos implícitos

RET

Realiza las siguientes operaciones:

IP ← [SP]

SP ← SP + 2

7.4. Estructura de un programa en ensamblador

En el siguiente recuadro se muestra la estructura global de un programa en lenguaje ensamblador con los principales bloques necesarios para inicializar los registros, definir los datos, escribir el código a ejecutar y finalización del programa.

```
DOSSEG          ; prepara los segmentos para trabajar con DOS
.MODEL SMALL    ; define el modo de creación del ejecutable
.STACK 100h     ; define el tamaño de la pila
.DATA           ; zona de definición de los datos
                 Definición de datos
.code
MOV AX, @data   ; inicialización de los datos en…
MOV DS, AX      ; el segmento de datos
                 Instrucciones del programa

MOV AH, 4CH     ; terminación del programa y
INT 21h         ; devolución del control a DOS
END             ; fin de programa
```

Los programas en leguaje ensamblador deben escribirse en un editor de texto ASCII. El nombre del fichero debe tener como extensión *.ASM. Para ensamblar un fichero se debe teclear desde la consola MS-DOS en la línea de comandos:

MASM NombreFichero.ASM

Si no se produce ningún error, se debe enlazar tecleando en la línea de comandos:

LINK NombreFichero.OBJ

Después de haber visto la estructura general, veremos ahora la posición de los elementos del código por 4 columnas:

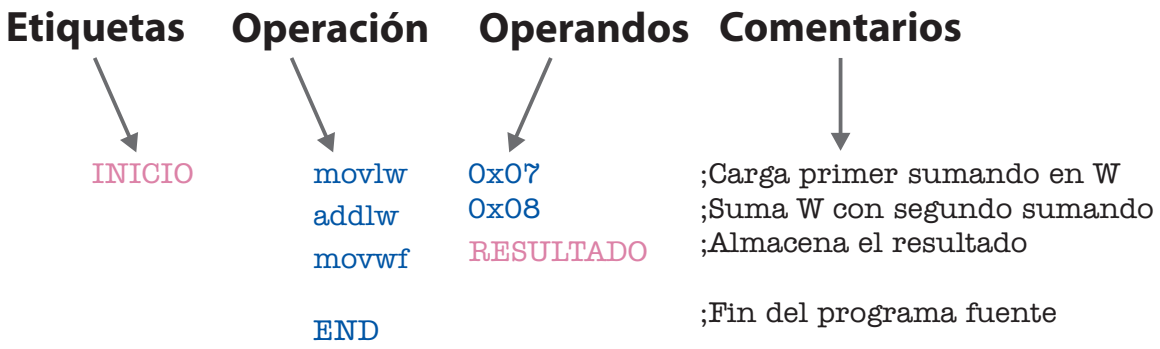


Figura 58. Posición de los diferentes elementos que conforman las líneas de código de un programa. Fuente: pulse aquí para ver la fuente.

Columna 1. *Etiquetas.* Las etiquetas se rigen por normas específicas:

- Deben situarse en la primera columna.
- Deben contener únicamente caracteres alfanuméricos.
- El máximo de caracteres es de 31.

Columna 2. *Operación.* Aquí situarán las instrucciones. Este es el único campo que nunca puede estar vacío. Este siempre contiene una instrucción o una directiva del ensamblador.

Columna 3. *Operandos.* El campo de operandos o de dirección puede contener una dirección o un dato, o bien estar en blanco. Normalmente contendrá registros o literales con los que se operará.

Columna 4. *Comentario.* Este campo es opcional. En él se situará cualquier comentario personalizado que deseemos especificar. Son útiles para saber qué hace un programa sin tener que descifrar el código entero. El compilador (ensamblador) ignorará todo texto más allá del carácter punto y coma ";". Los comentarios generalmente se sitúan en la cuarta columna para describir la acción de una línea de código, pero pueden situarse en cualquier parte de programa para describir cualquier otro evento, siempre que estén después del carácter ";".

Normalmente, las diferentes columnas se separan por una tabulación. El espacio mínimo entre dos columnas es de un carácter, que puede ser un espacio en vez de una tabulación.

Delimitadores (separación entre campos).

- Los campos van separados sólo con espacios y/o tabulaciones. No agregue nunca otros caracteres (comas, puntos, etc.).
- No utilice espacios extra, particularmente después de comas que separan operandos (Ej.: MOV 5, W).
- No use caracteres delimitadores (espacios y tabulaciones) en nombres o etiquetas.

A continuación, basándonos en la plantilla anterior, realizaremos la operación que suma dos números enteros (etiquetados Num 1 y Num 2), mediante la inserción de la parte del programa correspondiente, la cual se resalta de forma sombreada.

```
DOSSEG          ; prepara los segmentos para trabajar con DOS
.MODE SMALL     ; define el modo del ejecutable
.STACK 100h     ; define el tamaño de la pila
.DATA           ; zona de definición de los datos
Num1 DB 20h
Num2 DB 33h
Res  DB ?
.code
MOV AX, @data   ; inicialización de los datos en
MOV DS, AX      ; el segmento de datos
MOV AL, Num1
MOV AL, Num2
MOV Res, AL
MOV AH, 4Ch     ; terminación del programa y
INT 21h         ; devolución del control a DOS
END             ; fin de programa
```

Como se puede apreciar, las líneas sombreadas se ocupan de sumar los números almacenados en Num1 y Num2 y guardarlo en Res. Básicamente el número Num1 se guarda en byte menos significativo del registro acumulador (AL), posteriormente se la adiciona el número guardado en Num2 y el resultado final se guarda en Reg.

El siguiente ejemplo permite escribir en la consola MS-DOS el texto 'Hola Mundo'.

```
DOSSEG          ; prepara los segmentos para trabajar con DOS
.MODEL SMALL    ; define el modo del ejecutable
.STACK 100h     ; define la pila
.DATA           ; zona de definición de los datos
Texto DB 'Hola mundo$'
.code
MOV AX, @data   ; inicialización de los datos en
MOV DS, AX      ; el segmento de datos
MOV AH, 9       ; activa la impresión en pantalla
LEA DX, Texto   ; tranfiere el texto que se desea imprimir.
INT 21h
MOV Ah, 4Ch     ; terminación del programa y
INT 21h         ; devolución del control a DOS
END             ; fin de programa
```

Glosario

Álgebra de Boole

El álgebra booleana es un sistema matemático deductivo centrado en los valores cero y uno (falso y verdadero). Es una estructura algebraica que esquematiza las operaciones lógicas Y, O, NO y SI (AND, OR, NOT, IF), así como el conjunto de operaciones unión, intersección y complemento.

ALU (unidad aritmética lógica)

Ejecuta las operaciones aritméticas y de otro tipo que se especifican en el programa.

Bit

Dígito binario, que puede ser 0 o 1.

Byte

Conjunto de 8 bits que comprende los números del 0 al 255.

Código BCD

Es el código decimal usado más usado. Es el código de dígitos decimales codificados en binario (*BCD, Binary Coded Decimal*), que asigna una representación binaria sin signo de 4 bits a cada dígito entre 0 y 9, no usándose las palabras del código entre 1010 y 1111 (del 10 al 15 en decimal).

Código Gray

El código Gray tiene una característica fundamental. La variación entre una combinación y otra es de solo un bit. Esta es una característica muy importante en muchas aplicaciones como detección y corrección de errores en sistemas de comunicaciones digitales, de allí la importancia del mismo.

Compuertas lógicas

Son elementos de electrónica digital que permiten realizar operaciones lógicas entre cantidades binarias.

CPU (Central Processing Unit)

Es el hardware dentro de un procesador u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.

Lenguaje ensamblador

Es el lenguaje que sustituye a los códigos de operación binarios y las direcciones con **nombres simbólicos** o **mnemónicos** y que proporciona otras características útiles al programador.

Lenguaje máquina

Es el lenguaje binario en el que se definen y almacenan las instrucciones en la memoria, es decir, operaciones expresadas mediante la codificación binaria de cadenas de 1's y 0's.

Mapa de Karnaugh

Es un diagrama hecho de cuadros, donde cada cuadro está asociado una combinación binaria de entrada y su contenido representa el valor que toma la función para dicha combinación. Su estructura del mapa depende del número de variables que definen la función. A continuación veremos cómo se representan los K-mapas de 2, 3 y 4.

Señales binarias

Son señales eléctricas que indican voltajes y corrientes en los sistemas digitales, usan solamente dos valores discretos, 0 y 1.

Sistema digital

Dispositivo diseñado para la generación, transmisión, procesamiento o almacenamiento de señales digitales. También se puede decir que es una combinación de dispositivos diseñados para manipular cantidades físicas o información que estén representadas en forma digital; es decir, que sólo puedan tomar valores discretos.

Sistemas combinacionales

Son aquellos sistemas digitales en los que la salida sólo depende de la entrada actual, sin que se tenga en cuenta estados anteriores de la entrada o la salida. Por lo tanto, no necesita contener módulos de memoria.

Sistemas secuenciales

Son aquellos sistemas digitales que utilizan elementos de memoria para almacenar la información pasada del sistema. El sistema secuencial más simple es el biestable (dos estados posibles).

Tabla de verdad

Una tabla de verdad, o tabla de valores de verdad, es una tabla que muestra el valor de verdad de una proposición compuesta, para cada combinación de verdad que se pueda asignar a sus componentes.

Unidad de control

Se encarga de supervisar el flujo de información entre las diferentes unidades del procesador.

Enlaces de interés

Documental Lo Digital y Analógico En El Medio Digital

Documental y serie de entrevistas y estudios de entorno donde se abordan ¿Qué es Lo Digital? ¿Dónde se Originó? ¿Cuál es la historia? ¿Cuál es su Relevancia? ¿Cuál será el futuro de lo Digital? Entrevistas con especialistas y casos reales También se aborda el significado y precedente de lo analógico.

https://www.youtube.com/watch?v=cfvKy35P1QU

Lenguaje ensamblador

Tutorial completo de lenguaje ensamblador con explicación detallada de los diferentes tipos de instrucciones y algunos ejemplos para su uso.

http://comunidad.dragonjar.org/f177/tutorial-completo-de-lenguaje-ensamblador-8845/

Web de Logisim

Página principal de Logisim, software de libre distribución que será empleado a lo largo de la asignatura para desarrollar los ejercicios prácticos virtuales. Desde la web se podrá acceder a la documentación relacionada con la instalación y utilización del software.

http://www.cburch.com/logisim/index_es.html

Bibliografía

Referencias bibliográficas

Mano, M. & Kime, C. (2005). *Fundamentos de diseño lógico y de computadoras* (1st ed.). Madrid: Pearson Educación.

Molina Cantero, A., Díaz Ruiz, S., & Escudero Fombuena, J. (2015). *Estructura y Tecnología de Computadores* (1st ed.). Sevilla: Editorial Panella. Recuperado de https://www.dte.us.es/docencia/etsii/gii-ti/cedti/documentacion/estructura-y-tecnologia-de-computadores.pdf/

Flórez Fernández, H. (2010). *Diseño lógico* (1st ed.). Bogotá: Ediciones de la U.

Bibliografía recomendada

Angulo Usategui, J. & García Zubía, J. (2002). *Sistemas digitales y tecnología de computadores* (1st ed.). Madrid: Thomson Paraninfo.

Baena Oliva, C. (1997). *Problemas de circuitos y sistemas digitales* (1st ed.). Madrid: McGraw-Hill.

Gajski, D., Valero Cortés, M., & González Colás, A. (1997). *Principios de diseño digital* (1st ed.). Madrid [etc.]: Prentice Hall.

García Zubía, J. (2003). *Problemas resueltos de electrónica digital* (1st ed.). Madrid: Thomson.

Mandado, E. (1996). *Sistemas electrónicos digitales* (1st ed.). México: Alfaomega.

Palmer, J. & Perlman, D. (1995). *Introducción a los sistemas digitales* (1st ed.). México: McGraw-Hill Interamericana.

Roth, C. (2005). *Fundamentos de diseño lógico* (1st ed.). México: Thomson.

Wakerly, J., Alatorre Miguel, E., & Gámez Cuatzin, H. (2001). *Diseño digital* (1st ed.). México: Pearson educación.

Agradecimientos

Autor
Dr. D. Daniel Romero Pérez

Departamento de Desarrollo de Contenidos
Diseñadoras
D.ª Carmina Gabarda López
D.ª Ana Gallego Martínez
D.ª Cristina Ruiz Jiménez
D.ª Sara Segovia Martínez

Reservados todos los derechos VIU - 2017 ©.