UOC Universitat Oberta de Catalunya

SOC Servei d'Ocupació de Catalunya

Introducción a la electrónica digital

Esteve Gené Pujols

25 horas

Generalitat de Catalunya
Departament d'Empresa i Ocupació

Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Tabla de contenidos

Módulos	Contenidos
1. Introducción al diseño de sistemas digitales	1.1 Introducción a los circuitos lógicos 2.1 Circuitos lógicos combinacionales 3.1 Circuitos lógicos secuenciales
2. Simulación de sistemas digitales	2.1 Introducción al diseño digital: lenguajes descriptores de hardware 2.2 Descripción de sistemas digitales con VHDL 2.3 Del diseño VHDL a la síntesis de sistemas digitales
3. Implementación de sistemas digitales sobre dispositivos programables	3.1 Introducción a los dispositivos programables: FPGA 3.2 Del lenguaje de descripción a la síntesis del dispositivo
4. Sistemas de propósito específico	4.1 Introducción a los sistemas de propósito específico 4.2 Características de los procesadores digitales de señal (DSP)

1. Introducción al diseño de sistemas digitales

Un computador es una máquina construida a partir de dispositivos electrónicos básicos, interconectados adecuadamente. Podemos decir que estos dispositivos son las piezas con las que se construye un sistema digital.

El objetivo del curso **Introducción a la electrónica digital** es entender cómo está formado un computador desde el punto de vista electrónico. Para poder entenderlo, es necesario conocer a fondo los dispositivos electrónicos digitales básicos. Este es el objetivo de este módulo y de los siguientes.

1.1 Introducción a los circuitos lógicos

Los dispositivos electrónicos digitales más elementales son las puertas lógicas y los bloques lógicos, que forman los circuitos lógicos. Un circuito lógico se puede ver como un conjunto de dispositivos que manipulan de una manera determinada las señales electrónicas que les llegan (las señales de entrada) y generan como resultado otro conjunto de señales (las señales de salida).

Hay dos grandes tipos de circuitos lógicos:

- Los **circuitos combinacionales**, que se caracterizan por que el valor de las señales de salida en un momento determinado depende del valor de las señales de entrada en ese mismo momento.
- Los **circuitos secuenciales,** en los que el valor de las señales de salida en un momento determinado depende de los valores que han llegado por las señales de entrada desde la puesta en funcionamiento del circuito y hasta ese mismo momento (tienen, por lo tanto, capacidad de memoria).

El objetivo fundamental de este módulo es conocer a fondo los circuitos lógicos combinacionales, es decir, saber cómo están formados y ser capaces de utilizarlos con agilidad, hasta el punto de estar totalmente familiarizados con ellos.

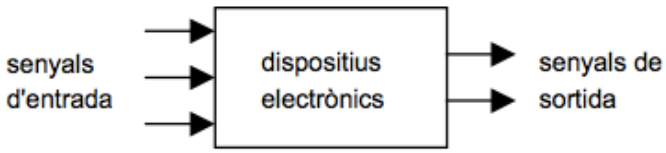
Para llegar a este punto será necesario haber satisfecho los objetivos siguientes:

- Entender **el álgebra de Boole** y las diferentes maneras de expresar funciones lógicas.
- Conocer las diferentes **puertas lógicas**, ver cómo se pueden utilizar para sintetizar funciones lógicas y ser capaces de hacerlo; entender por qué es deseable minimizar el número de puertas y de niveles de puertas de los circuitos, y saber hacerlo.
- Conocer la funcionalidad de los diferentes bloques combinacionales y ser capaces de utilizarlos en el diseño de circuitos.

En definitiva, tras el estudio de este módulo, debemos ser capaces de construir fácilmente un circuito cualquiera usando los diferentes dispositivos que se habrán conocido, así como de entender la funcionalidad de cualquier circuito dado.

1.1.1 Circuitos, señales, funciones lógicas

Entendemos por circuito un sistema formado por un cierto número de señales de entrada (cada señal corresponde a un cable), un conjunto de dispositivos electrónicos que ejecutan operaciones sobre las señales de entrada (las manipulan electrónicamente) y que generan un cierto número de señales de salida. Así pues, las señales de salida se pueden ver como funciones de las señales de entrada y se puede decir que los dispositivos electrónicos computan estas funciones.



Traducción del texto de la imagen: señales de entrada / dispositivos electrónicos / señales de salida

En los circuitos que forman los sistemas computadores, los cables se pueden encontrar en dos valores de tensión (voltaje): tensión alta o tensión baja. Estos dos valores de tensión se identifican normalmente por los símbolos 1 y 0

respectivamente, de forma que se dice que una señal vale 0 (cuando en el cable correspondiente hay tensión baja) o vale 1 (cuando en el cable hay tensión alta, también llamada tensión de alimentación). Las señales que pueden tomar los valores 0 o 1 se denominan señales lógicas o binarias. Un circuito lógico es aquel donde las señales de entrada y de salida son lógicas. Las funciones que computa un circuito lógico son funciones lógicas.

Código binario

El **código binario** es un sistema de numeración donde todas las cantidades se representan utilizando como base dos cifras: cero y uno (0 y 1). En otras palabras, es un sistema de numeración de base 2, mientras que el sistema que utilizamos más habitualmente es de base 10, o decimal.

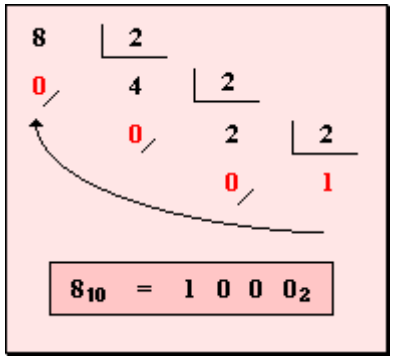
De binarios a decimales

Dado un número *n*, binario, para expresarlo en decimal, se tiene que escribir cada número que lo compone (bit), multiplicado por la base del sistema (base = 2), elevado a la posición que ocupa.

Ejemplo: $1101_2 = 1 * 2^3 + 1 * 2^2 + 0 * 1^1 + 1 * 2^0 = 13_{10}$

De decimales a binario

Para pasar un número de base 10 a base 2 se divide el número inicial en base 10 sucesivamente por 2 hasta obtener un cociente menor que 2. Escribiendo el último cociente y los restos en forma ascendente se obtiene el número en base 2.



El complemento a dos

El **complemento a dos** de un número ***N*** que, expresado en el sistema binario, está compuesto por ***n*** dígitos, se define como:

$$C_2^N = 2^n - N$$

Veamos un ejemplo. Tomemos el número *N* = 45 que, cuando se expresa en binario es $N = 101101_2$, con seis dígitos, y calculamos su complemento a dos:

N = 45, *n* = 6; $2^6 = 64$ y, por lo tanto $C_2^N = 64 - 45 = 010011_2$

Puede parecer complicado, pero es muy fácil obtener el complemento a dos de un número a partir de su complemento a uno porque el complemento a dos de un número binario es una unidad más grande que su complemento a uno, es decir:

$$C_2^N = C_1^N + 1$$

1.1.2 Álgebra de Boole

Un **álgebra de Boole** es una entidad matemática formada por un conjunto que contiene dos elementos, unas operaciones básicas sobre estos elementos y una lista de axiomas que definen las propiedades que cumplen las operaciones.

Los dos elementos de un álgebra de Boole se pueden denominar falso y cierto o, más usualmente, 0 y 1. Así, una variable booleana o variable lógica puede tomar los valores 0 y 1.

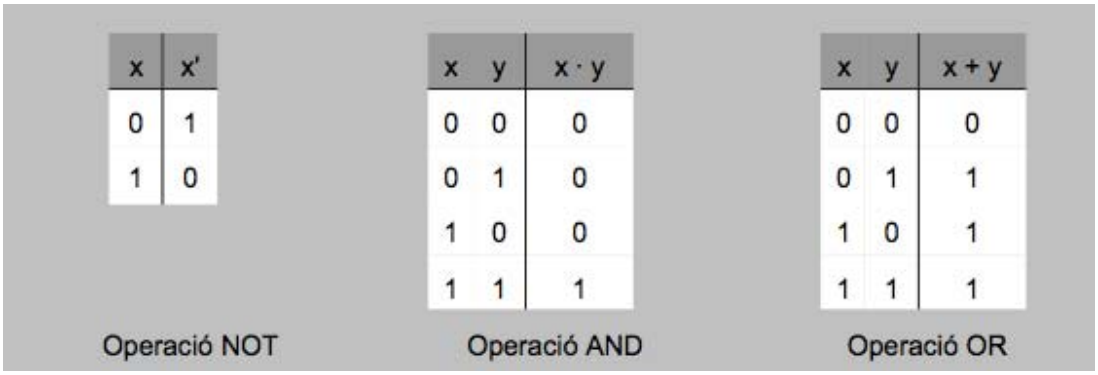
Las operaciones booleanas básicas son las siguientes:

- la **negación** o **complementación** o **NOT**, que corresponde a la partícula *no* y se representa con una comilla simple (□). Así, la expresión *x*□ denota la negación de la variable *x* y se lee "no x".
- el **producto lógico** o **AND**, que corresponde a la conjunción *y* de la lógica y se representa por el símbolo ·. Así, si *x* e *y* son variables lógicas, la expresión $x \cdot y$ denota su producto lógico y se lee "x e y".
- la **suma lógica** u **OR**, que corresponde a la conjunción *o* y se representa por el símbolo +. Así, la expresión $x + y$ denota la suma lógica de las variables *x* e *y* y se lee "x o y"

La operación de negación se puede representar también de otras maneras. La más usual es $\overline{X}$.

Es importante...

... no confundir los operadores · y + con las operaciones producto entero y suma entera a las que estamos acostumbrados. El significado de los símbolos vendrá dado por el contexto en el que nos encontramos. Así, si estamos trabajando con enteros, 1 + 1 = 2 (uno más uno igual a dos), mientras que si estamos en un contexto booleano, 1 + 1 = 1 (cierto o cierto igual a cierto, tal como se puede ver en la tabla de la operación OR).



x	x'
0	1
1	0

Operació NOT

x	y	x · y
0	0	0
0	1	0
1	0	0
1	1	1

Operació AND

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

Operació OR

Estas operaciones booleanas básicas se pueden definir escribiendo el resultado que dan por cada posible combinación de valores de las variables de entrada, tal como se muestra en la figura anterior. En las tablas de esta figura, a la izquierda de la raya vertical están todas las combinaciones posibles de las variables y, a la derecha, el resultado de la operación para cada combinación.

1.1.3 Tabla de la verdad

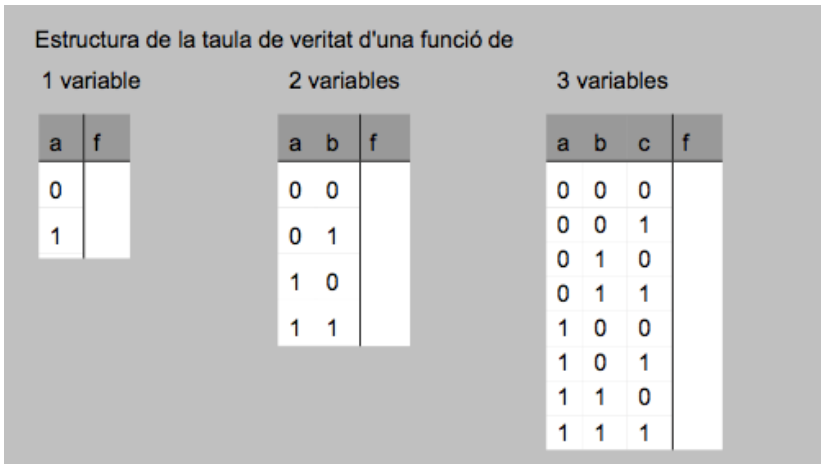
Una tabla de la verdad expresa una función lógica especificando el valor que tiene la función por cada posible combinación de valores de las variables de entrada.

En concreto, a la izquierda de la tabla, hay una lista con todas las combinaciones de valores posibles de las variables de entrada y, a la derecha, el valor de la función para cada una de las combinaciones. Por ejemplo, las tablas que habíamos visto en la figura de antes eran, de hecho, las tablas de la verdad de las funciones NOT, AND y OR.

Si una función tiene *n* variables de entrada, y dado que una variable puede tomar solo dos valores, 0 o 1, las entradas pueden tomar 2*n* combinaciones de valores diferentes. Por lo tanto, su tabla de la verdad tendrá a la izquierda *n* columnas (una por cada variable) y 2*n* filas (una por cada combinación posible). A la derecha, habrá una columna con los valores de la función.

Las filas las escribiremos siempre en orden lexicográfico: es decir, si interpretamos las diferentes combinaciones como números naturales, escribiremos primero la combinación correspondiente al número 0 (formada solo por ceros), después la

correspondiente al número 1 y así sucesivamente en orden creciente hasta la correspondiente al número 2*n*-1 (formada solo por unos).



Traducción del texto de la imagen: Estructura de la tabla de la verdad de una función de 1 variable

1.2 Circuitos lógicos combinacionales

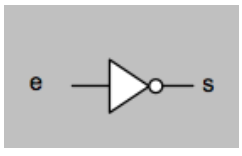
1.2.1 Puertas lógicas. Síntesis y análisis

Los dispositivos electrónicos que computan funciones lógicas se denominan puertas lógicas. Son dispositivos que están conectados a un cierto número de señales de entrada y una señal de salida.

A continuación, presentamos el símbolo gráfico con el que se representa cada puerta lógica. En todas las figuras que aparecen a continuación, las señales de entrada quedan a la izquierda de las puertas y la señal de salida queda a la derecha.

Puerta NOT o inversor

Esta puerta se representa gráficamente con este símbolo:

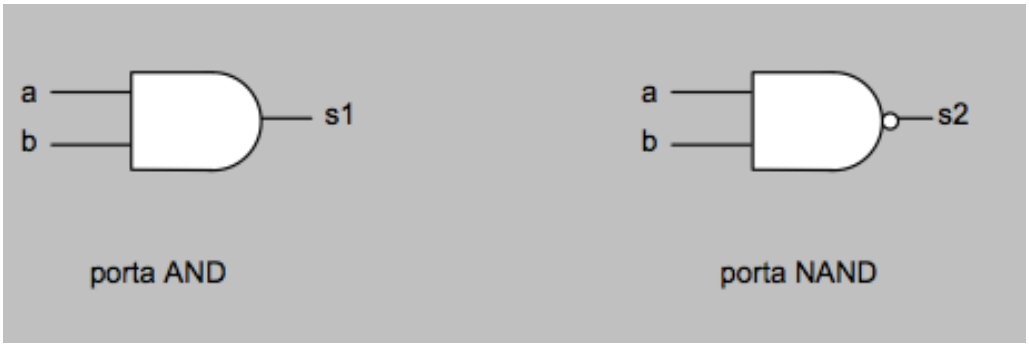


e	s
0	1
1	0

El funcionamiento es el siguiente: si en la entrada *e* hay un 0 (tensión baja), entonces en la salida *s* habrá un 1 (tensión alta). Y a la inversa, si hay un 1 en su punto *e*, entonces habrá un 0 en el punto *s*.

Puertas AND y NAND

Se representan gráficamente con estos símbolos:



a	b	S1
0	0	0
0	1	0
1	0	0
1	1	1

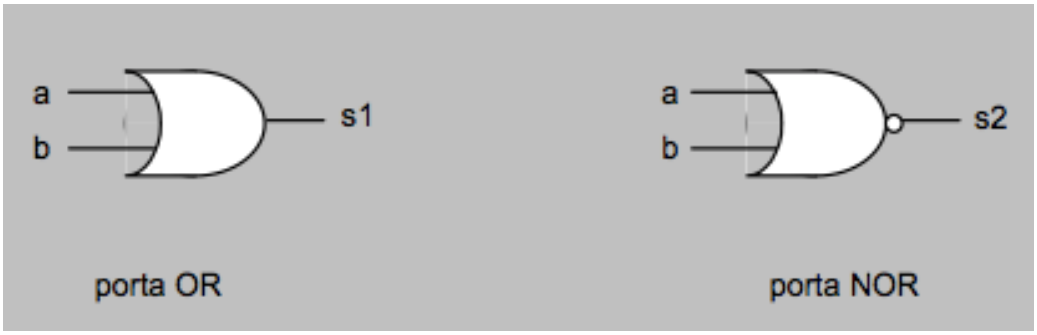
a	b	S2
0	0	1
0	1	1
1	0	1
1	1	0

La puerta AND implementa la función lógica AND, es decir: la salida *s*1 vale 1 solo si las dos entradas *a* y *b* valen 1. Por lo tanto, $s1 = a \cdot b$.

La puerta NAND implementa la función lógica NAND. En su punto *s*2 hay un 1 siempre que en alguna de las dos entradas *a* o *b* haya un 0. Es decir, $s2 = (a \cdot b)'$.

Puertas OR y NOR

Se representan gráficamente con estos símbolos:



a	b	S1
0	0	0
0	1	1
1	0	1
1	1	1

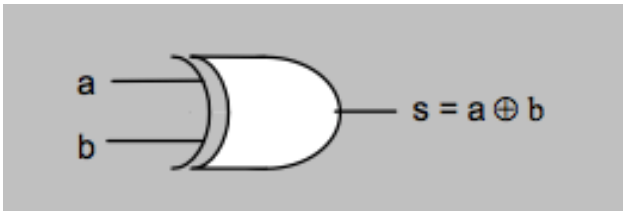
a	b	S2
0	0	1
0	1	0
1	0	0
1	1	0

La puerta OR implementa la función lógica OR, es decir: a la salida *s*1 hay un 1 si cualquiera de las dos entradas está en 1. Por lo tanto, *s*1 = *a* + *b*.

La puerta NOR implementa la función lógica NOR. En su punto *s*2 encontraremos un 1 solo cuando en las dos entradas haya un 0, es decir: *s*2 = (*a* + *b*)'.

Puerta XOR

Implementa la función lógica XOR, es decir, la salida *s* vale 1 si alguna de las dos entradas vale 1, pero no si valen 1 las dos a la vez. Se representa gráficamente con este símbolo:



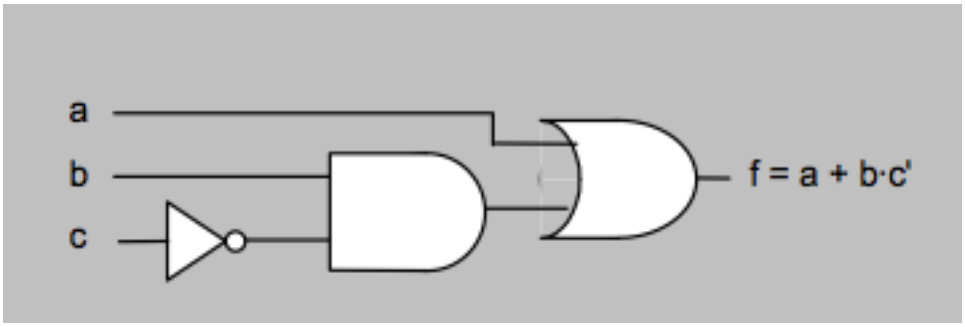
a	b	s
0	0	0
0	1	1
1	0	1
1	1	0

Síntesis y análisis

Cualquier función lógica se puede implementar usando estas puertas, es decir, se puede construir un circuito que se comporte como la función. El proceso de obtener una expresión de una función a partir del circuito que la implementa se denomina **análisis**.

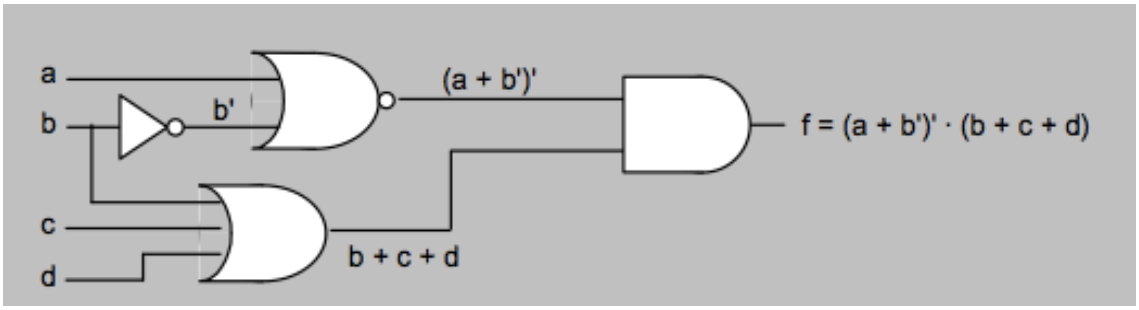
Para **sintetizar** (o implementar) una función a partir de su expresión algebraica, basta con sustituir cada operador de la función por la puerta lógica adecuada.

Por ejemplo, un término producto de tres variables se implementará con una puerta AND de tres entradas. Las puertas tienen que estar interconectadas entre sí y con las entradas según lo indique la expresión. Por ejemplo, el circuito que implementa la función f(a, b, c) = a + b · c' es el siguiente:



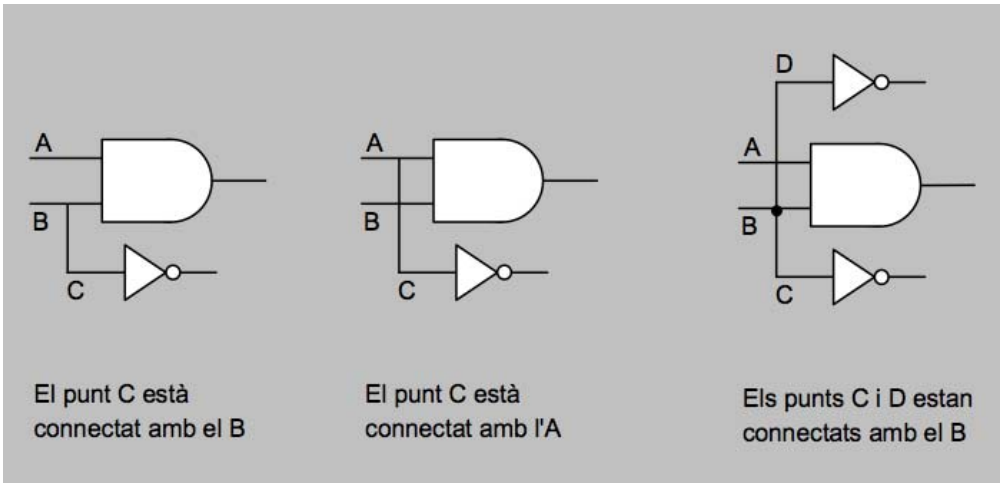
Otro ejemplo, el circuito siguiente implementa la función:

f (a, b, c, d) = (a + b')'(b + c + d).



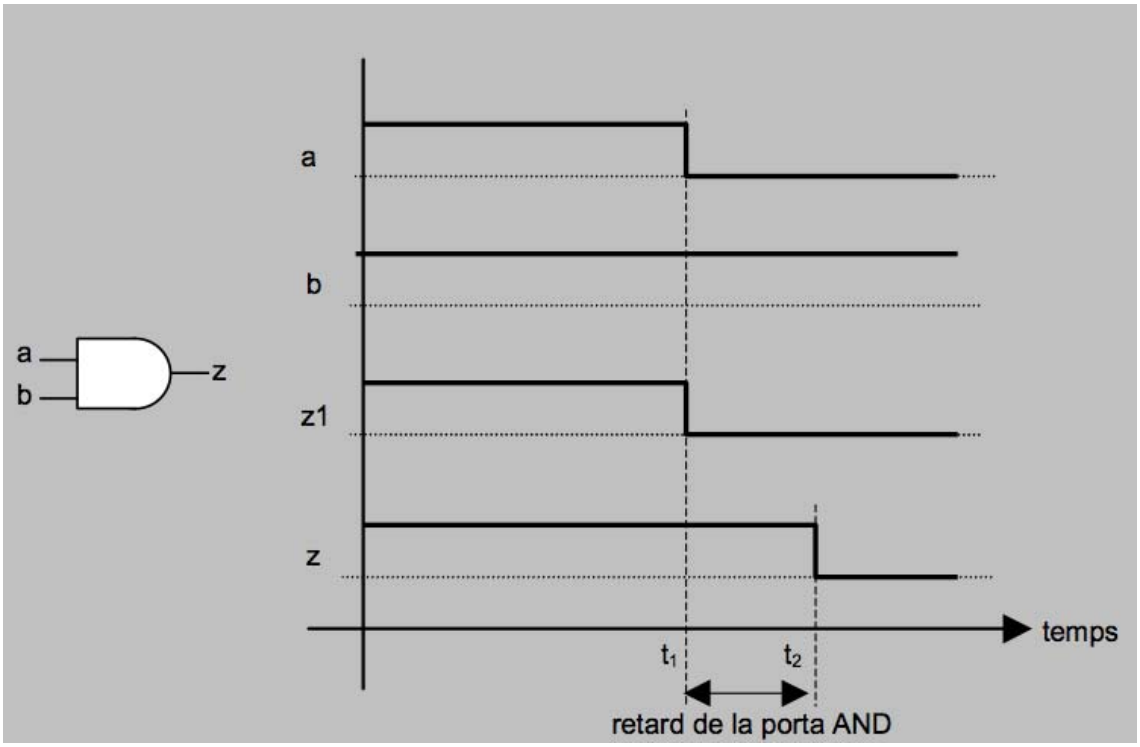
En un circuito, si una línea empieza sobre otra línea perpendicular, se entiende que las líneas están conectadas (y por lo tanto, que tienen el mismo valor lógico). Si dos líneas perpendiculares se cruzan, se entiende que no están conectadas. Si sobre una línea se pone un punto, se entiende que todas las líneas que lo tocan están conectadas.

A continuación, se muestran algunos ejemplos:



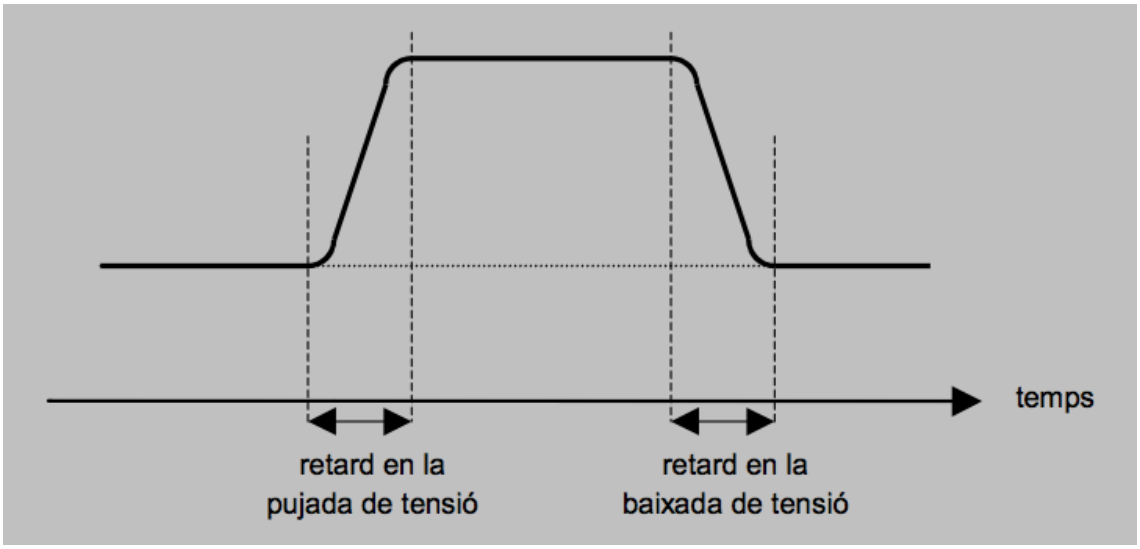
1.2.2 Retrasos, cronogramas, niveles de puertas

Las puertas lógicas no responden instantáneamente a las variaciones en las señales de entrada, sino que tienen un cierto retraso. La mejor manera de entender este concepto es mirando la siguiente figura, donde hay un circuito sencillo (solo una puerta AND) y un cronograma de su funcionamiento.



Un cronograma es una representación gráfica de la evolución de las señales de un circuito a lo largo del tiempo. Supongamos que en las entradas *a* y *b* tenemos conectados dos interruptores que nos permiten en cada momento poner estas señales a 0 o a 1. En el ejemplo de la figura anterior, hemos supuesto que inicialmente las dos señales están a 1 y que en el instante *t*1 ponemos la señal a 0. En este momento, la señal *z* se tendría que poner a 0 porque $0 \cdot 1 = 0$. Esta hipotética situación es la que se muestra en el cronograma con la señal *z*1. No obstante, en la realidad, *z* no se pone a 0 hasta el instante *t*2, debido a que los dispositivos electrónicos internos de la puerta AND tardan un cierto tiempo en reaccionar. Diremos que la puerta AND tiene un retardo de *t*2 - *t*1.

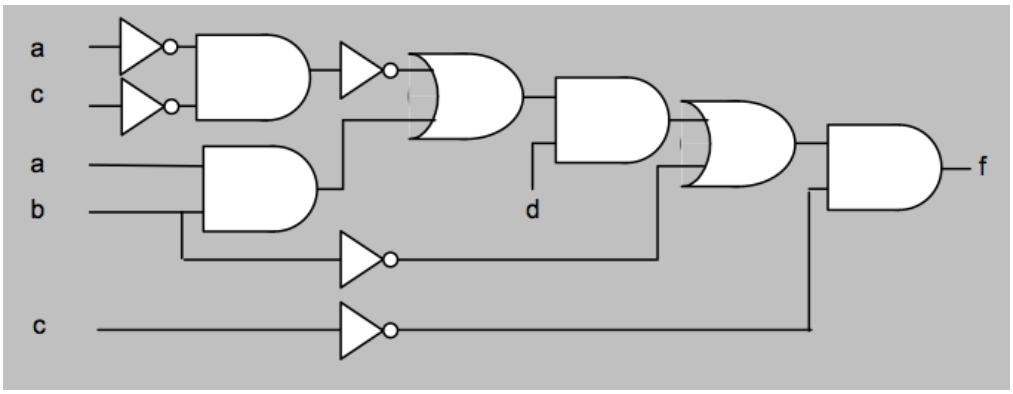
Cada puerta tiene un retardo diferente, que depende de la tecnología que se haya usado para construirla. Los retardos son muy pequeños, del orden de nanosegundos, pero hay que tenerlos en cuenta a la hora de construir físicamente un circuito. De hecho, la transición entre diferentes niveles de tensión de una señal tampoco es instantánea, sino que se produce tal como se muestra en la figura siguiente:

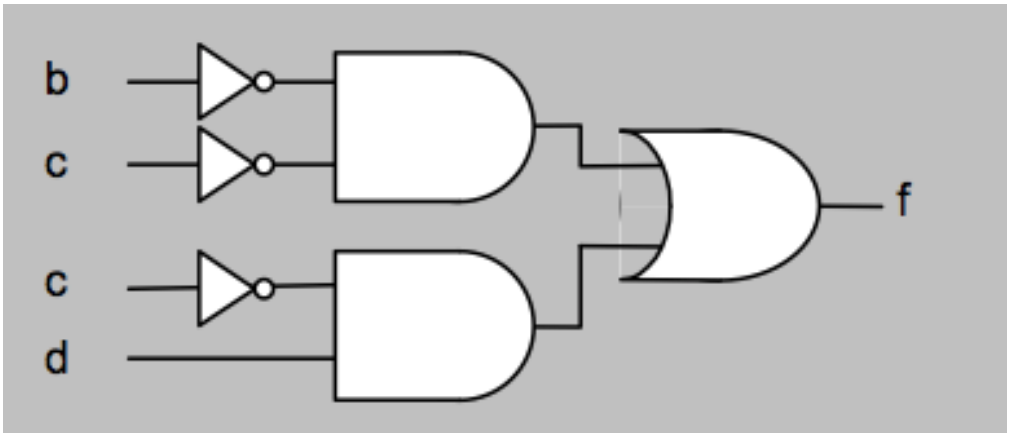


Uno de los objetivos de los ingenieros electrónicos que construyen circuitos es que el tiempo de respuesta del circuito sea lo más pequeño posible. Dado que cada puerta tiene un cierto retardo, un circuito será en general más rápido cuantos menos niveles de puertas haya entre las entradas y las salidas.

El **número de niveles de puertas** de un circuito es el máximo número de puertas que una señal tiene que atravesar consecutivamente para generar la señal de salida. Al contabilizar el número de niveles de puertas de un circuito, no se tienen en cuenta las puertas NOT (por razones que no vamos a ver en este curso).

Por ejemplo, la figura siguiente muestra el número de niveles de puertas de dos circuitos diferentes (podéis comprobar que la función que implementan es la misma):

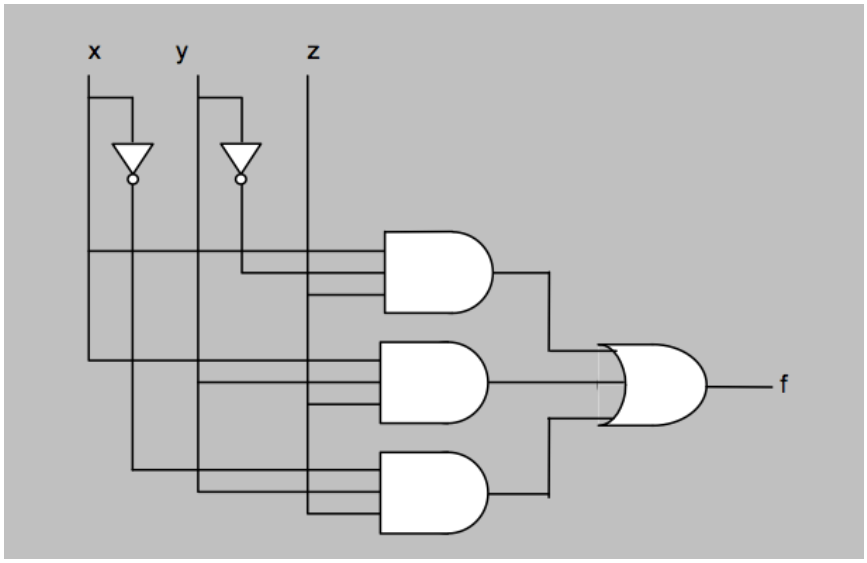




Un buen ingeniero escogería el circuito de la segunda figura para implementar esta función, puesto que es más rápido.

No existe una fórmula universal para encontrar la expresión de una función que dará lugar al circuito más rápido posible. No obstante, conocemos una manera de garantizar que un circuito no tendrá más de dos niveles de puertas: partir de la expresión de la función en suma de términos mínimos. En efecto, el circuito correspondiente a una suma de términos mínimos tiene, además de las puertas NOT, un primer nivel de puertas AND (que computan los diferentes términos mínimos) y un segundo nivel donde habrá una única puerta OR, con tantas entradas como términos mínimos tenga la expresión. Por ejemplo, la figura siguiente muestra el circuito que se obtiene al sintetizar esta función:

f = xy'z + xyz + x'yz



Los circuitos que se obtienen a partir de las sumas de términos mínimos se denominan circuitos a dos niveles. Denominaremos síntesis a dos niveles al proceso de obtenerlos.

Tener en cuenta los retardos de las diferentes puertas y de las transiciones entre niveles de tensión es fundamental a la hora de construir circuitos. No obstante, en este curso, vamos a considerar que los circuitos son ideales, de forma que no se tendrán nunca en cuenta los retardos, es decir, se asumirá siempre que son 0.

1.2.3 Bloques combinacionales

Un bloque combinacional es un circuito lógico combinacional con una funcionalidad determinada. Está construido a partir de puertas, como los circuitos que hemos visto hasta ahora.

Hasta este momento, las piezas que hemos usado para sintetizar circuitos han sido puertas lógicas. Después de estudiar este capítulo, podremos usar también los bloques combinacionales como piezas para diseñar circuitos más complejos, como por ejemplo un computador, que no se pueden pensar a escala de puertas pero sí a escala de bloques.

Multiplexor

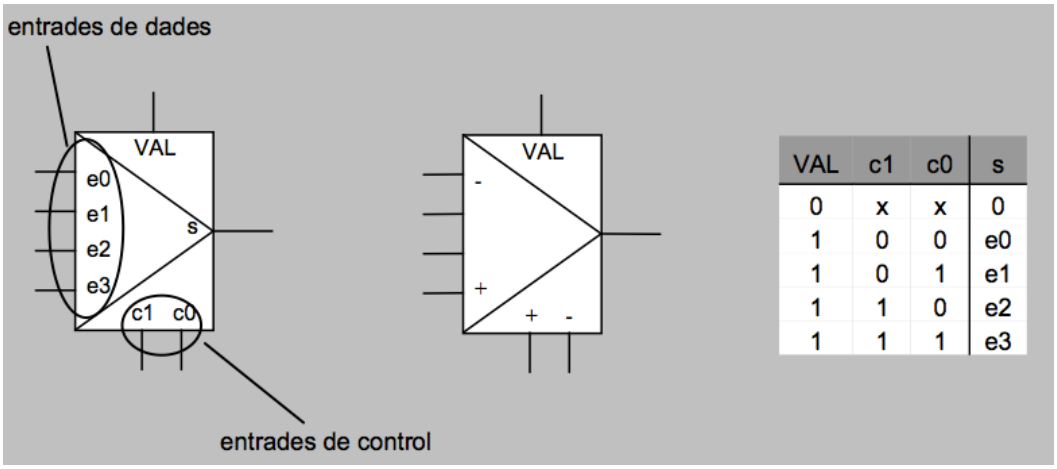
Imaginemos que en una ciudad hay tres calles que confluyen en otra calle de un único carril. Hará falta un urbano o algún tipo de señalización para controlar que en cada momento circulen hacia la calle de salida los coches provenientes de un única calle confluente.

Un multiplexor es un bloque que ejerce la función de urbano en los circuitos electrónicos. Tiene un cierto número de señales de entrada que compiten para conectarse a una única señal de salida y unas señales de control que sirven para determinar cuál de las señales de entrada se conecta en cada momento con la salida.

Más concretamente, las entradas y salidas de un multiplexor son las siguientes:

- 2*m* entradas de datos, identificadas por la letra e y numeradas desde 0 hasta 2*m*-1. Diremos que la entrada de datos numerada con el 0 es la de menos peso y la numerada con el 2*m*-1, la de más peso.
- una salida de datos, *s*.
- m entradas de control o de selección, identificadas por la letra *c* y numeradas desde 0 hasta *m*-1. Diremos que la entrada de control numerada con el 0 es la de menos peso y la numerada con el *m*-1, la de más peso.
- una entrada de validación, que denominamos VAL.

La siguiente figura muestra dos posibles representaciones gráficas de un multiplexor de cuatro entradas de datos (*m* = 2).



VAL	c1	c0	s
0	x	x	0
1	0	0	e0
1	0	1	e1
1	1	0	e2
1	1	1	e3

La diferencia entre ellas es que en la primera ponemos los nombres de las entradas, mientras que en la segunda solo indicamos con los signos + y - cuáles son las de mayor y menor peso (el resto se asume que están ordenadas en orden de peso). Las dos representaciones son igualmente válidas.

Para especificar el tamaño de un multiplexor, diremos cuántas entradas de datos tiene o bien cuántas entradas de control tiene. Por ejemplo, un multiplexor de ocho entradas de datos es lo mismo que un multiplexor de tres entradas de control. Si en un multiplexor no dibujamos la entrada de validación, asumiremos que esta está a 1.

El funcionamiento del multiplexor es el siguiente:

- Cuando la entrada de validación vale 0, la salida del multiplexor se pone a 0 (y por lo tanto el valor de las entradas de datos es indiferente, tal como reflejan las x de la tabla de la verdad).
- Cuando la entrada de validación vale 1, entonces las entradas de control determinan cuál de las entradas de datos se conecta con la salida, de la manera siguiente: se interpretan las variables conectadas a las entradas de control (*c*1 y *c*0 en el ejemplo) como un número codificado en binario con *m* bits (la entrada de más peso corresponde al bit de menos peso); si el número codificado es *i*, la entrada de datos que se conecta con la salida es la numerada con el número *i*.

Codificadores y descodificadores

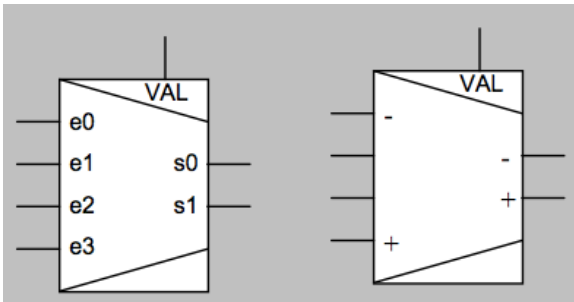
La función de un **codificador** es generar la codificación binaria de un número dado. Los codificadores disponen de las señales siguientes:

- Una entrada de validación, VAL, que funciona igual que en el caso de los multiplexores: si vale 0, todas las salidas valen 0 (cuando no dibujamos la entrada de validación, asumiremos que vale 1).

- 2*m* entradas de datos (de un bit), identificadas por la letra *e* y numeradas de 0 a 2*m*-1 (la de número más alto es la de más peso).
- *m* salidas de datos (de un bit), identificadas por la letra *s* y numeradas de 0 a *m*- 1, que se interpretan como si formaran un número codificado en binario con *m* bits (la salida de más peso corresponde al bit de más peso).

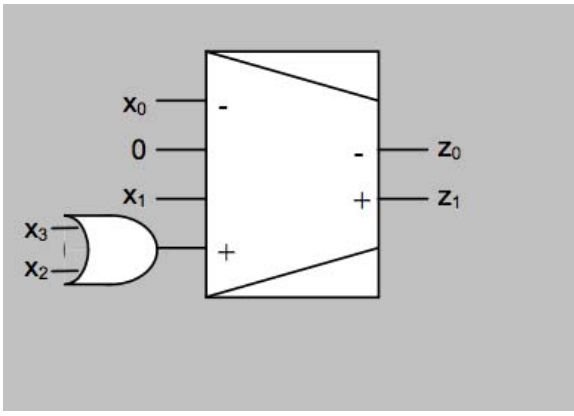
El funcionamiento de un codificador es el siguiente:
Cuando la entrada de validación vale 1, si la entrada de más peso de entre las que están a 1 es la numerada con el número i, entonces las salidas codifican en binario el número i.

La siguiente figura muestra la representación gráfica de un codificador 4-2 y la tabla de la verdad que explica su funcionamiento (observemos que, como en el caso de los multiplexores, podemos usar los símbolos + y - en lugar de los nombres de las entradas y salidas).



VAL	e3	e2	e1	e0	s1	s0
0	x	x	x	x	0	0
1	0	0	0	0	0	0
1	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

La figura siguiente muestra un ejemplo de uso de un codificador y la tabla de la verdad que describe el funcionamiento del circuito.



X_3	X_2	X_1	X_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	x	1	0
0	1	x	x	1	1
1	0	x	x	1	1
1	1	x	x	1	1

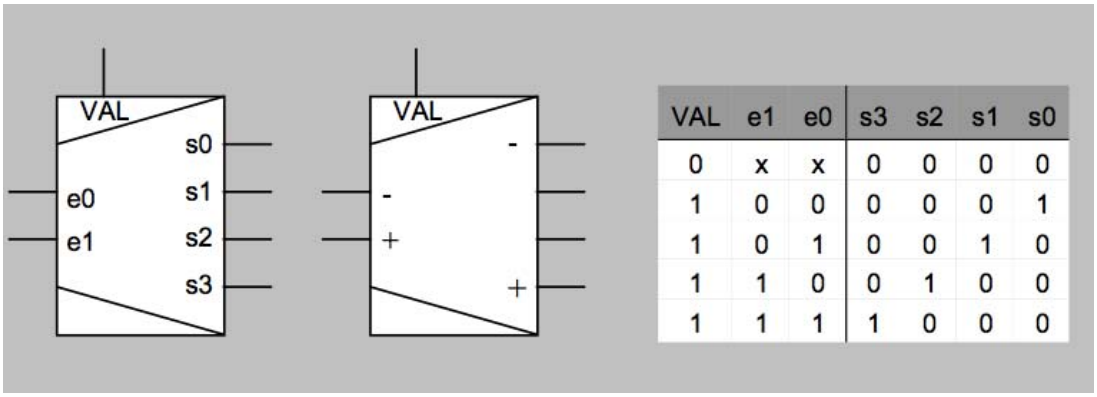
Recordemos que, si en un codificador no dibujamos la entrada de validación, asumiremos que esta está a 1.

Los **descodificadores** hacen la función inversa de los codificadores: dada una combinación binaria presente en la entrada, indican a qué número decimal corresponde.

Los descodificadores disponen de las señales siguientes:

- Una entrada de validación.
- *m* entradas de datos, identificadas por la letra *e* y numeradas de 0 a *m*-1, que se interpretan como si formaran un número codificado en binario (la entrada de más peso corresponde al bit de más peso).
- 2*m* salidas, identificadas por la letra *s* y numeradas de 0 a 2*m*-1, de las cuales solo una vale 1 en cada momento (si la entrada de validación está a 0, entonces todas las salidas valen 0).

El funcionamiento de un descodificador es el siguiente:
Cuando la entrada de validación vale 1, si las entradas codifican en binario el número *i*, entonces se pone a 1 la salida numerada como *i*.



VAL	e1	e0	s3	s2	s1	s0
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

Los descodificadores también se pueden usar para implementar funciones lógicas. Si la función tiene *n* variables, usaremos un descodificador *n*-2*n* y conectaremos las variables en las entradas en orden de peso. De este modo, la salida *i* del descodificador se pondrá a 1 cuando las variables, interpretadas como bits de un

número en binario, codifiquen el número *i*. Por ejemplo, si conectamos las variables [*x*1 *x*0] en las entradas [*e*1 *e*0] del descodificador de la figura anterior, la salida *s*2 valdrá 1 cuando [*x*1 *x*0] = [1 0].

Por lo tanto, para implementar la función será suficiente con conectar a una puerta OR las salidas correspondientes a las combinaciones que hacen que la función valga 1. Cuando alguna de estas combinaciones esté presente en la entrada del descodificador, la salida correspondiente se pondrá a 1 y por lo tanto de la puerta OR saldrá un 1. Cuando las variables construyan una combinación que hace que la función valga 0, todas las entradas de la puerta OR valdrán 0 y por lo tanto también su salida.

Decaladores lógicos y aritméticos

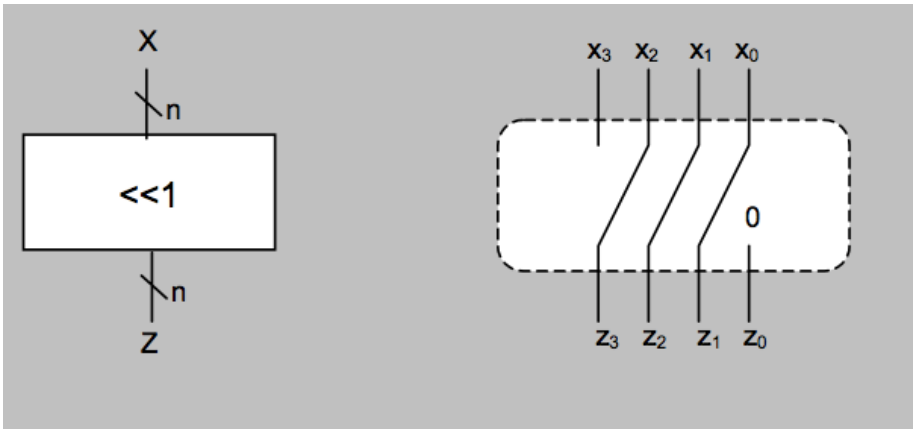
Los decaladores tienen una señal de entrada y una señal de salida, ambas de *n* bits. La señal de salida se obtiene desplazando los bits de entrada *m* veces hacia la derecha o hacia la izquierda.

• Si el desplazamiento es hacia la izquierda, los *m* bits de menos peso de la salida se ponen a 0.

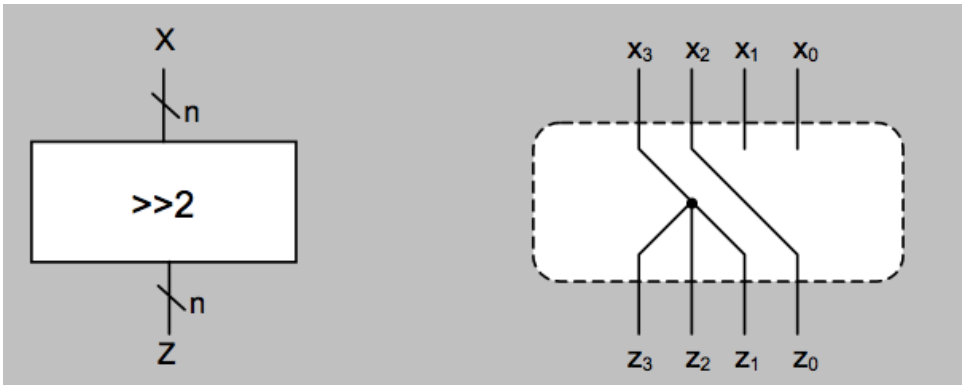
• Si el desplazamiento es hacia la derecha, hay dos posibilidades para los *m* bits de más peso de la salida: en los decaladores lógicos se ponen a 0 y en los decaladores aritméticos toman el valor del bit de más peso de la entrada.

Fijémonos en que, si interpretamos las entradas y salidas como número codificados en complemento a 2 con *n* bits, lo que hacen los decaladores aritméticos es mantener en la salida el signo de la entrada.

Decalador (lógico o aritmético) de 1 bit a la izquierda:



Decalador aritmético de 2 bits a la derecha:



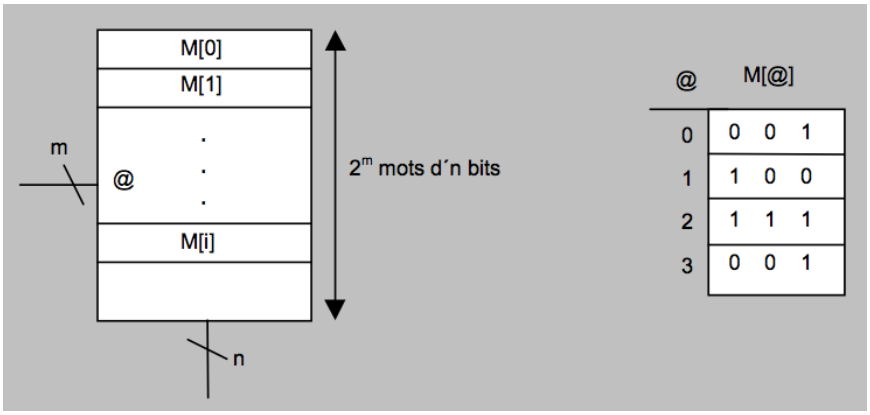
Memoria ROM

Podemos ver una memoria ROM como un archivador con cajones que guardan bits. Cada cajón tiene una cierta capacidad (todos tienen la misma) y el archivador tiene un número determinado de cajones, que es siempre una potencia de 2.

La memoria ROM tiene los elementos siguientes:

- 2*m* palabras o datos de *n* bits, cada una en una posición (cajón) diferente de la memoria ROM. Las posiciones que contienen los datos están numeradas desde el 0 hasta el 2*m*-1 y a estos números se les llama direcciones.
- Una entrada de direcciones de *m* bits, que se identifica por el símbolo @. Los *m* bits de la entrada de direcciones se interpretan como números codificados en binario (y por lo tanto hay que determinar cuál es el peso de cada bit).
- Una salida de datos de *n* bits.

El funcionamiento de la ROM es el siguiente:



Así pues, solo se puede acceder al valor de una palabra (leerlo) en cada instante (es como si en cada momento solo se pudiera abrir un cajón).

La figura muestra un posible contenido de una memoria ROM de 4 palabras de 3 bits. En este ejemplo,

M[0] = 001 M[1] = 100 M[2] = 111 M[3] = 001

La memoria ROM es un bloque combinacional que permite guardar el valor de 2*m* palabras de *n* bits.

La denominación **ROM** deriva del inglés **Read Only Memory** porque se refiere a memorias en las que no se pueden realizar escrituras sino solo lecturas. Cuando los *m* bits de la entrada de direcciones (interpretados en binario) codifican el número *i*, entonces la salida toma el valor del dato que ha almacenado en la dirección *i*. Para referirnos a este dato, usaremos la notación M[*i*] y diremos que estamos leyendo el dato de la dirección *i*.

Comparador

El comparador es un bloque combinacional que compara dos números codificados en binario e indica qué relación existe entre ellos. Dispone de las señales siguientes:

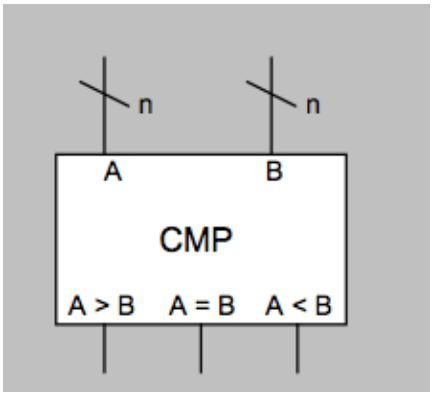
• Dos entradas de datos de *n* bits, que reciben los nombres A y B. Se interpretan como número codificados en binario.

• Tres salidas de un bit, de las que solo una vale 1 en cada momento:

La salida A > B vale 1 si el número que llega por la entrada A es mayor que el que llega por la entrada B.

La salida A = B vale 1 si los dos números de entrada son iguales.

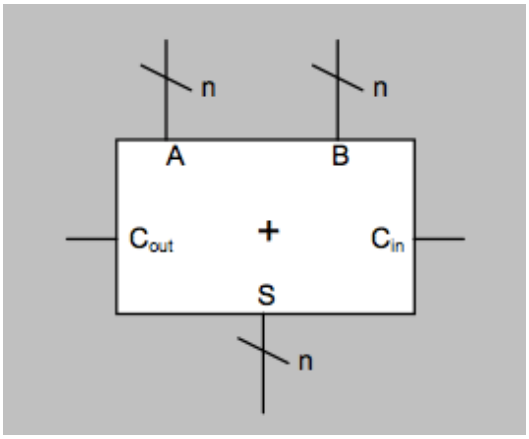
La salida A < B vale 1 si el número que llega por la entrada A es más pequeño que el que llega por la entrada B.



Sumador

El sumador es un bloque combinacional que realiza la suma de dos números codificados en binario o bien en complemento a 2.

La figura siguiente muestra la representación gráfica de un sumador de *n* bits.

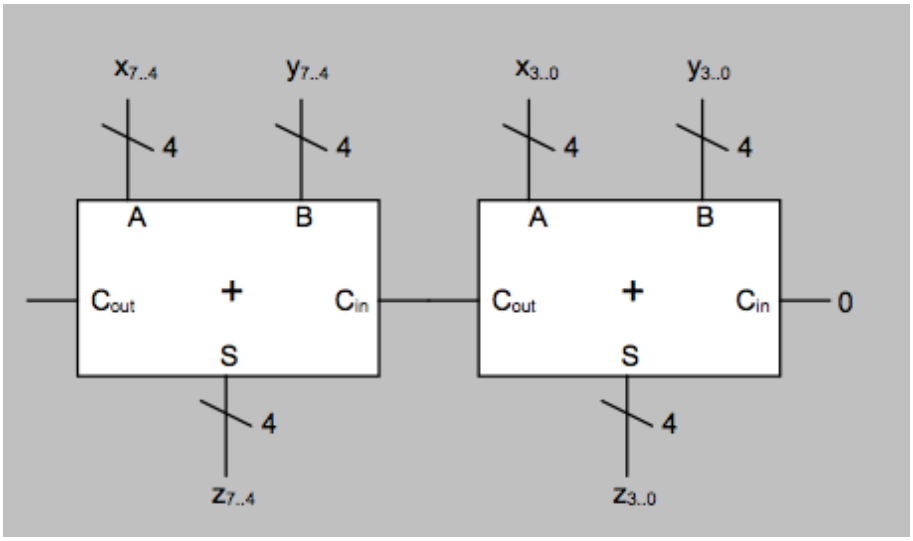


Las señales de las que dispone son las siguientes:

- Dos entradas de datos de *n* bits, que denominaremos A y B, por donde llegarán los números que se van a sumar.
- Una salida de *n* bits, S, que tomará el valor de la suma de los números A y B.
- Una salida de un bit, C_{out}, que valdrá 1 si al hacer la suma se produce transporte en el bit de más peso.
- Una entrada de un bit, C_{in}, por donde llega un transporte de entrada.

La entrada C_{in} es útil cuando se quieren sumar números de $2 \cdot n$ bits y solo se dispone de sumadores de *n* bits. En este caso, se encadenan dos sumadores: el primero suma los *n* bits más bajos de los números y el segundo, los *n* bits más altos. La salida C_{out} del primer sumador se conecta con la entrada C_{in} del segundo para que el resultado sea correcto.

La siguiente figura muestra un ejemplo para el caso $n = 4$, en el que hacemos la suma Z = X + Y y donde X, Y y Z son números de 8 bits.



Fijémonos en que el sumador que suma los bits más bajos lo dibujamos a la derecha, porque así los bits quedan ordenados de la manera como estamos acostumbrados a verlos.

Si no necesitamos tener en cuenta un transporte de entrada, conectaremos un 0 a la entrada C_{in}.

Como ya sabemos, el hecho de limitar la longitud de los números a un determinado número de bits (*n*) tiene como consecuencia que el resultado de una suma no sea siempre correcto (es incorrecto cuando se produce rebosamiento, es decir, cuando el resultado requiere más de *n* bits para ser codificado). En las sumas binarias, sabemos que si se produce transporte en el bit de más peso, entonces el resultado es incorrecto. En cambio, en las sumas en complemento a 2 no hay ninguna relación entre el transporte y el rebosamiento.

Unidad lógica aritmética (ALU)

Una unidad aritmética y lógica es un aparato capaz de realizar un determinado conjunto de operaciones aritméticas y lógicas sobre dos números de entrada codificados en binario o en complemento a 2.

Las unidades aritméticas y lógicas se denominan UAL o, también, ALU (del inglés *Arithmetic and Logic Unit*). Para diseñar una ALU, hay que especificar el conjunto de operaciones que queremos que realice. Por ejemplo, una ALU puede hacer la suma, la resta, el AND y el OR de los operandos de entrada. En cada momento, se especificará a la ALU cuál es la operación que debe ejecutar.

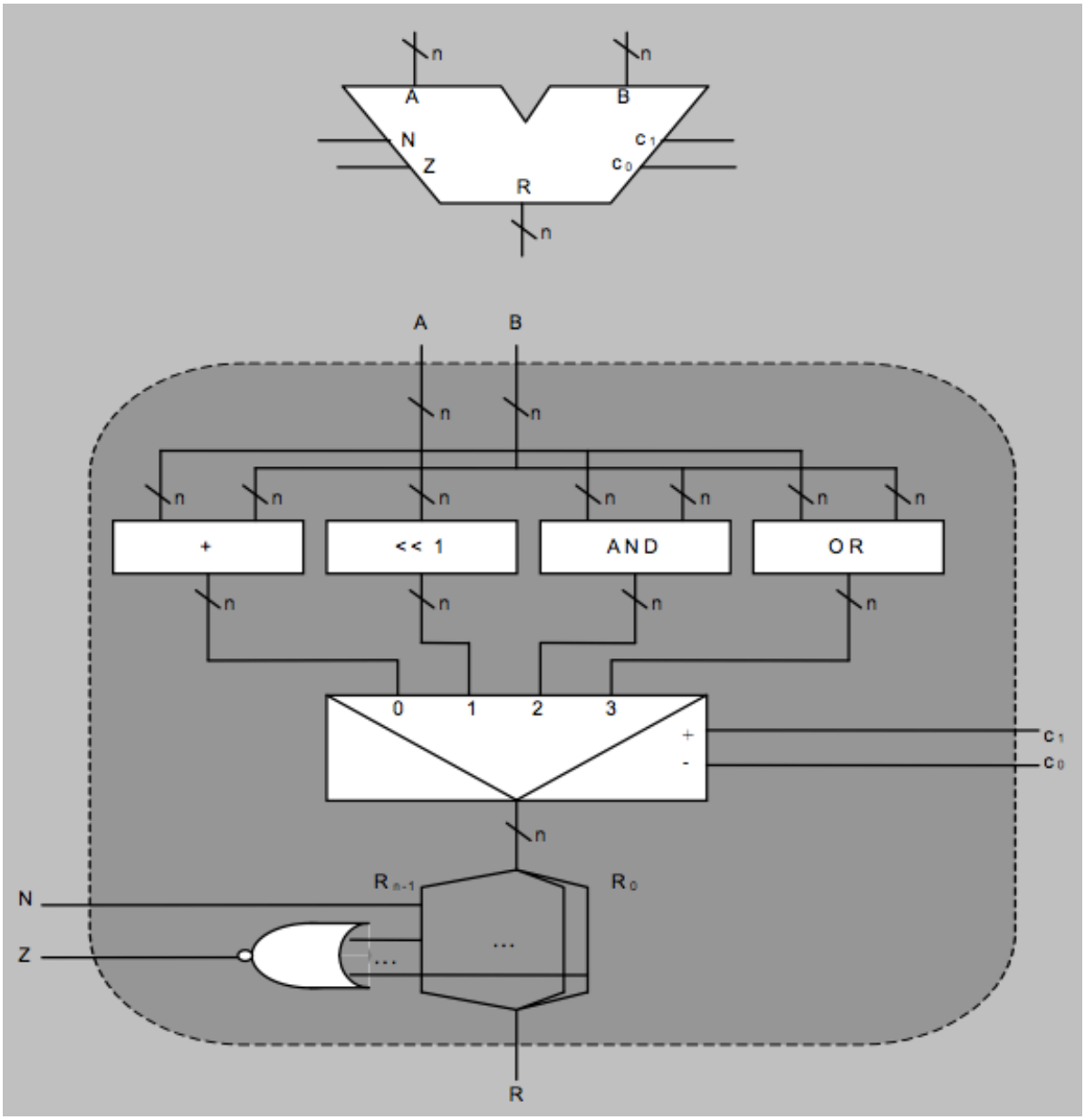
Las señales de las que dispone una ALU son las siguientes:

- dos entradas de datos de *n* bits, A y B, por donde llegarán los números sobre los que se tienen que ejecutar las operaciones.
- una salida de datos de *n* bits, R, donde se obtendrá el resultado de la operación
- un cierto número de entradas de control, ci, de 1 bit cada una. Si la ALU es capaz de realizar 2*m* operaciones diferentes, tiene que tener *m* entradas de control. Cada combinación de las entradas de control indicará a la ALU que ejecute una operación concreta. Un cierto número de salidas de 1 bit, que se denominan bits de estado, y tienen la función de indicar algunas circunstancias que se pueden haber producido durante el cálculo.

Los bits de estado más habituales son los que indican si se ha producido transporte en el último bit (se denomina C), si se ha producido rebosamiento (V), si el resultado de la operación ha sido negativo (N) o si el resultado de la operación ha sido cero (Z).

El bit de rebosamiento se suele identificar con las letras O o V, que derivan de la palabra inglesa para rebosamiento, *overflow*. Por ejemplo, la ALU de la siguiente figura puede ejecutar cuatro operaciones:

c_1	c_0	R
0	0	A + B
0	1	2*A
1	0	A AND B
1	1	A OR B



En esta figura hemos disgregado los bits del bus correspondientes a la señal de salida R para poder dibujar la implementación de los bits N y Z. Se tiene que interpretar que todos los bits de R se conectan a la puerta NOR (de *n* entradas) que computa Z.

1.3 Circuitos lógicos secuenciales

En el módulo "Los circuitos lógicos combinacionales" hemos visto que los circuitos computan funciones lógicas de las señales de entrada: el valor de las señales de salida en un instante determinado depende del valor de las señales de entrada en ese mismo momento. Cuando las señales de entrada varían, entonces las de salida

también variarán en consecuencia (después del retardo introducido por las puertas y los bloques, que en este curso no tenemos en cuenta).

Ahora bien, en algunas aplicaciones se necesita que el valor de las señales de salida no dependa solo de las entradas en el mismo momento, sino que tenga también en cuenta los valores que han tomado las entradas con anterioridad. En los circuitos que hemos conocido hasta ahora, esto no es posible: se necesitan los elementos que conforman los circuitos lógicos secuenciales.

En este módulo vamos a conocer el concepto de sincronización y se van a estudiar los biestables o también llamados *flip-flop*, que son los dispositivos secuenciales más básicos, y los bloques secuenciales, que se construyen a partir de biestables y tienen una funcionalidad determinada.

El objetivo fundamental de este módulo es conocer los circuitos lógicos secuenciales, es decir, saber cómo están formados y poder utilizarlos con agilidad. Para llegar a ese punto, habrá que haber satisfecho los objetivos siguientes:

- A partir de la funcionalidad que se quiere que tenga un circuito lógico, saber discernir si el circuito tiene que ser de tipo secuencial o combinacional.
- Entender el concepto de memoria, la necesidad de una sincronización en los circuitos lógicos secuenciales y el funcionamiento de la señal de reloj.
- Conocer el funcionamiento del biestable D y de todas las entradas de control de las que puede disponer.
- Conocer la funcionalidad de los diferentes bloques secuenciales y saber utilizarlos en el diseño de circuitos.

Tras el estudio de este módulo, debemos ser capaces de construir un circuito cualquiera que combine tanto dispositivos secuenciales como combinacionales, así como entender la funcionalidad de un circuito dado que contenga todos estos elementos.

1.3.1 Necesidad de memoria en los circuitos lógicos

Sea un circuito con una señal de entrada *X* y uno de salida *Z*, los dos de *n* bits, que interpretamos como números representados en complemento a 2. Supongamos que queremos que $Z = X + 2$. Con los elementos estudiados en el módulo "Los circuitos lógicos combinacionales" sabemos cómo hacerlo, incluso de muchas maneras diferentes. Cuando el valor presente en la entrada *X* varíe, entonces *Z* también cambiará de valor en consecuencia.

Supongamos ahora que queremos que el valor de *Z* corresponda a la suma de todos los valores que han estado presentes en la entrada *X* durante un intervalo de tiempo determinado (durante el cual el valor de *X* ha variado). Con los dispositivos lógicos que conocemos hasta ahora no lo podemos conseguir porque, al cambiar el valor de *X*, el valor anterior ha desaparecido y ya no lo podemos usar para calcular la suma.

Es necesario que este circuito sea capaz de recordar o retener los valores anteriores de algunas señales, es decir, debe tener memoria. Esta es la funcionalidad que distingue los **circuitos lógicos secuenciales** de los **combinacionales**.

1.3.2 Reloj, sincronización

En los circuitos combinacionales, la única noción temporal que interviene es el presente. En cambio, en los circuitos secuenciales se tiene en cuenta la evolución temporal de las señales (y aparece, como se verá más adelante, la noción de futuro).

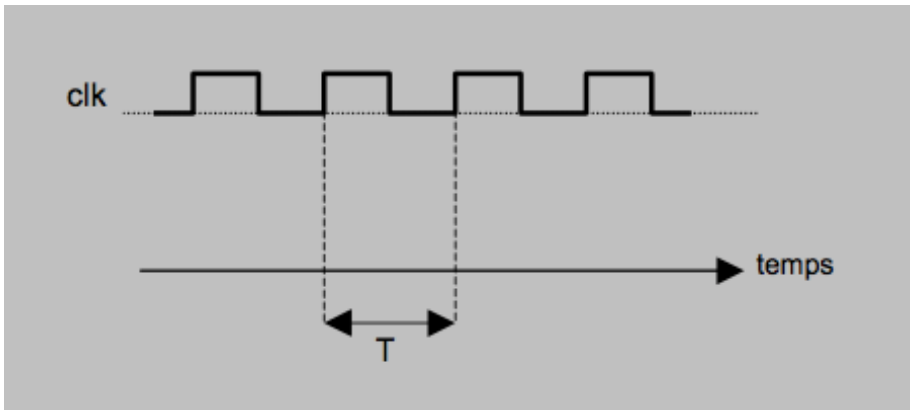
Ahora bien, en la descripción del circuito del ejemplo anterior, ¿qué quiere decir exactamente "todos los valores que han estado presentes en la entrada *X* durante un tiempo determinado"? La señal *X* puede ir cambiando de valor de forma aleatoria en el tiempo: puede valer 13 durante 4 ns, después -25 durante 10 ns, después 0 durante 1 ns, etc. ¿Cómo puede determinar el circuito en qué momento *X* cambia de valor, es decir, en qué momento tiene que considerar que "*X* ha

dejado de tener el valor antiguo" y empieza "a tener el valor nuevo"? Para poder determinarlo, el circuito debe disponer de un mecanismo de sincronización. En los circuitos secuenciales que vamos a estudiar en este módulo, se utiliza una señal de reloj como forma de sincronización.

El **reloj** es una señal que sirve para determinar los instantes en los que un circuito secuencial ve o es sensible al valor de las señales y responde en consecuencia. A esta tarea que lleva a cabo la señal de reloj se la denomina sincronización de los circuitos.

En concreto, la señal de reloj toma los valores 0 y 1 de manera cíclica y continua desde la puesta en marcha de un circuito y hasta que este se para. Usualmente se usa la notación **clk** para hacer referencia a la señal de reloj (deriva del inglés *clock*).

La siguiente figura muestra el cronograma de la señal de reloj:



El ciclo que forma la secuencia de valores 0 y 1 tiene una duración determinada y constante, *T*, que se denomina periodo. Se mide en segundos o, más habitualmente, en nanosegundos (mil millonésimas de segundo).

Los instantes en los que la señal de reloj pasa de 0 a 1 se denominan flancos ascendentes. El intervalo de tiempo que hay entre un flanco y el siguiente se denomina ciclo o ciclo de reloj. Por lo tanto, la duración de un ciclo es un periodo, *T* segundos.

La frecuencia del reloj es la inversa del periodo, es decir, es el número de ciclos de reloj que tienen lugar durante un segundo. Se mide en hercios (ciclos por segundo); lo más habitual es usar el múltiplo megahercios (millones de ciclos por segundo), que se abrevia MHz. Por ejemplo, si tenemos un reloj con un periodo de 2 ns, su frecuencia es de:

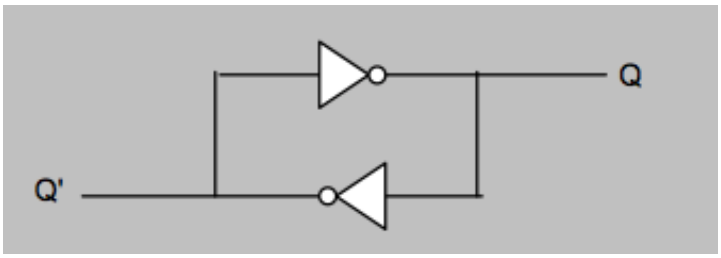
$$\frac{1\,cicle}{2{\cdot}10^{-9}\,segons} = 0'5{\cdot}10^{9}\ \text{cicles/segon} = 500{\cdot}10^{6}\ \text{cicles/segon} = 500\ \text{MHz}$$

La señal de reloj puede sincronizar los circuitos de varias formas. En este curso, solo vamos a ver la que se usa de forma más habitual, la llamada sincronización por flanco ascendente. Esta forma de sincronización establece que los dispositivos secuenciales de un circuito serán sensibles a los valores de las señales en los instantes de los flancos ascendentes, tal como vamos a ver en el apartado siguiente.

El biestable D (o *flip-flop* D)

En el apartado anterior hemos visto la necesidad de que los circuitos lógicos tengan capacidad de memoria. En este apartado vamos a ver cómo se construyen los dispositivos que pueden recordar los valores de las señales.

Examinando el circuito que se muestra en la figura siguiente,



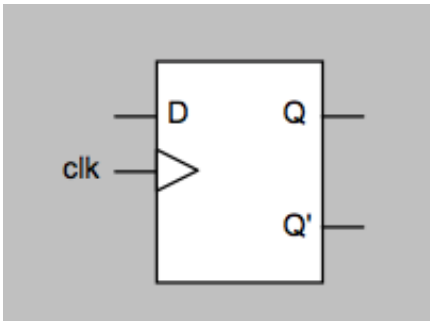
vemos que el valor que haya en los puntos *Q* y *Q*' (0 o 1) se mantendrá indefinidamente, puesto que la salida de cada inversor está conectada con la

entrada del otro. Por lo tanto, podemos decir que este circuito es capaz de recordar, o mantener en el tiempo, un valor lógico.

Ahora bien, este circuito no es muy útil porque no admite la posibilidad de modificar el valor recordado. Interesa diseñar un circuito que tenga esta misma capacidad de memoria, pero que además permita que el valor en su punto *Q* pueda cambiar en función de los requerimientos del usuario. Un circuito con estas características se denomina biestable.

Los **biestables** son los dispositivos de memoria más elementales: permiten guardar un bit de información. Un **biestable** tiene dos salidas, *Q* y *Q*'. Se dice que *Q* es el valor que guarda el biestable en cada momento, o el valor almacenado al biestable, y que *Q*' es su negación.

Existen diferentes tipos de biestables. En este curso, solo vamos a ver uno, **el biestable D.** La siguiente figura muestra su representación gráfica:



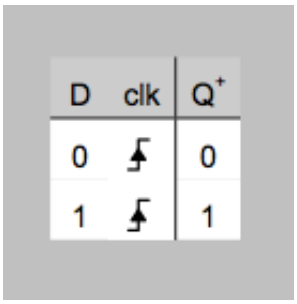
Podemos observar que dispone de una entrada de reloj, puesto que se trata de un dispositivo secuencial.

El **biestable D** funciona de la manera siguiente:

La salida *Q* toma el valor que haya en la entrada D en cada flanco ascendente de reloj. Durante el resto del ciclo, el valor de *Q* no cambia.

Es decir, el biestable solo es sensible al valor presente en la entrada D en los instantes de los flancos ascendentes.

La siguiente figura muestra la tabla de la verdad que describe el comportamiento del biestable D (no ponemos la columna correspondiente a *Q*' porque es la negación de *Q*).

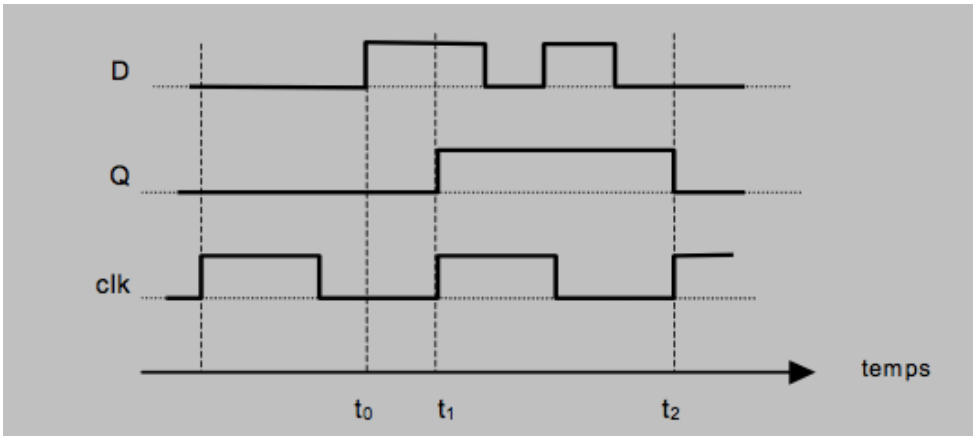


Por lo tanto, *Q*+ no identifica ninguna señal del circuito, sino el valor de la propia señal *Q* en un instante futuro, a partir del momento en el que se produzca el próximo flanco. Así pues, esta notación nos permite describir con precisión la evolución temporal de las señales en un circuito lógico secuencial.

En esta figura, se introducen algunas notaciones que se van a usar de ahora en adelante:

el símbolo  representa un flanco ascendente de reloj
el símbolo "+" a la derecha del nombre de una señal se refiere al *valor que tomará esta señal cuando se produzca el siguiente flanco ascendente de reloj*

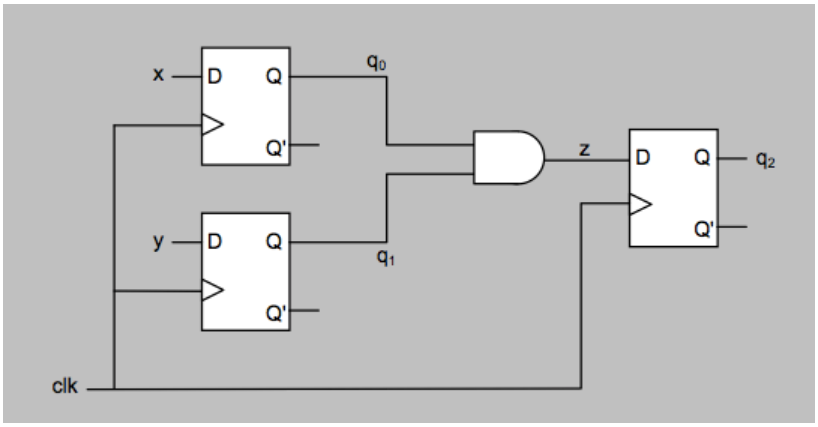
La siguiente figura muestra el cronograma del comportamiento de un biestable D:



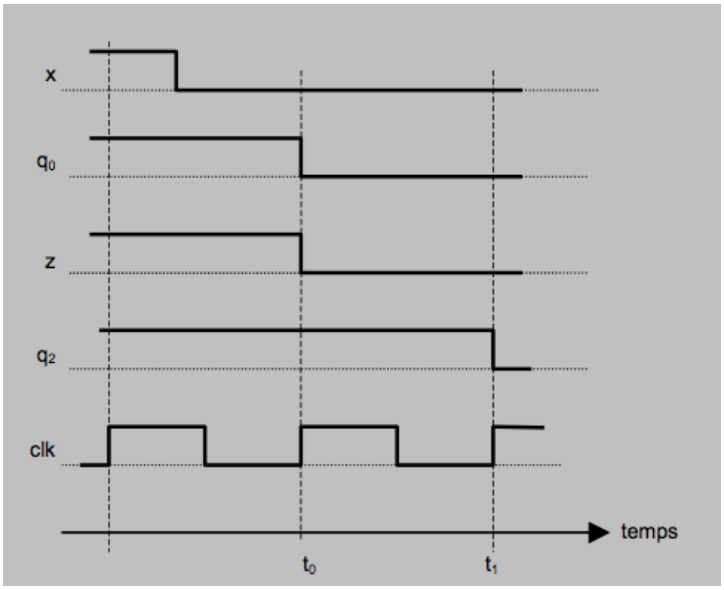
Se puede observar que, a pesar de que *D* se pone a 1 en el instante *t*0, *Q* no cambia de valor hasta el instante *t*1 porque hasta ese momento no se produce un flanco ascendente de reloj.

A pesar de las subsiguientes variaciones de *D*, *Q* se mantiene inalterado hasta el instante *t*2, en el que se produce el próximo flanco de reloj.

En los circuitos reales, es habitual realizar alguna función combinacional sobre la salida de uno o más biestables y conectar el resultado a la entrada de otro biestable. La siguiente figura muestra un ejemplo:



El cronograma de la evolución de este circuito durante un cierto intervalo de tiempo (para simplificar el dibujo, no hemos puesto ni *i* ni *q*1; asumimos que las dos señales se mantienen a 1 todo el rato):



En el instante *t*0, *q*0 se pone a 0 porque a la entrada *D* del biestable correspondiente hay un 0 (*x* = 0); en consecuencia, *z* también se pone a 0. Al dibujar la línea del cronograma correspondiente a *q*2, podríamos dudar de si se tiene que poner a 0 en ese mismo instante, puesto que sobre la línea vertical correspondiente a *t*0, *z* está tanto a 1 como 0.

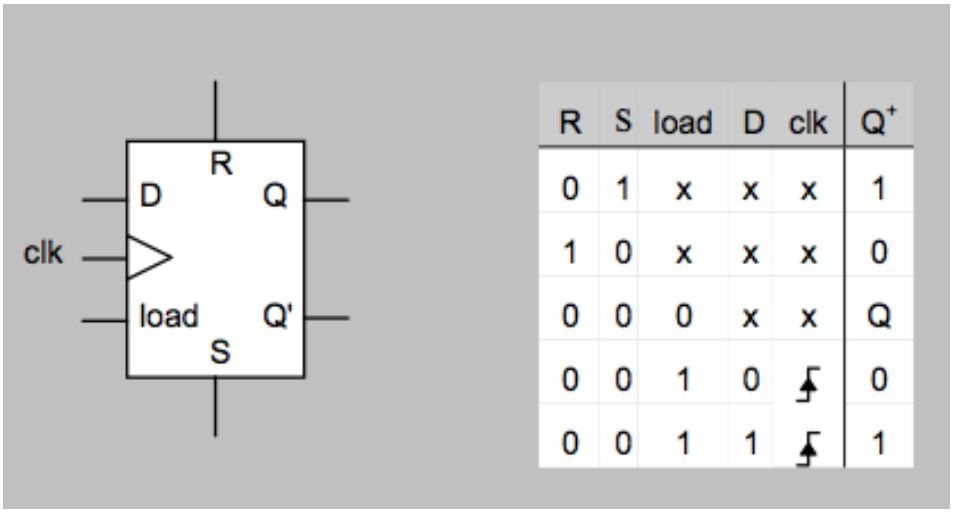
El valor de un biestable D puede variar en los instantes de flancos ascendentes en función del valor que haya a las entradas D y *load*. Ahora bien, hay que tener la capacidad de darle un valor inicial: el valor que tomará al ponerse en marcha un circuito.

Las **entradas asíncronas** de un biestable permiten modificar su valor de forma instantánea, independientemente del valor de la señal de reloj y de las entradas D y *load*. Se dice que las entradas asíncronas tienen más prioridad que el resto de entradas.

Los biestables suelen disponer de dos entradas asíncronas:

- R (del inglés *reset*): cuando se pone a 1, el biestable se pone a 0.
- S (del inglés *set*): en el momento en el que se pone a 1, el biestable se pone a 1.

La figura siguiente muestra la representación gráfica de un biestable D con entradas asíncronas y señal de carga, así como la tabla de la verdad que describe su comportamiento:



R	S	load	D	clk	Q^+
0	1	x	x	x	1
1	0	x	x	x	0
0	0	0	x	x	Q
0	0	1	0	↓	0
0	0	1	1	↓	1

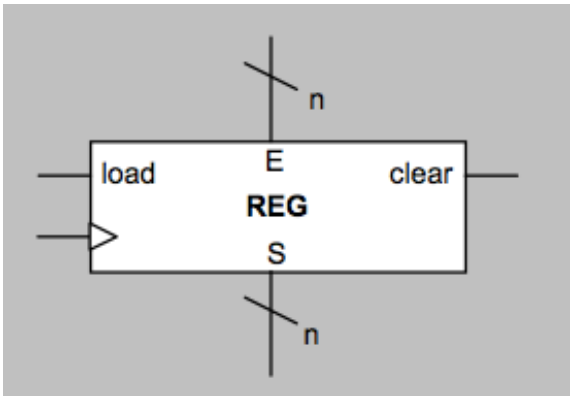
Cuando en un circuito no dibujamos las señales *load*, R y S de un biestable, asumiremos por defecto que valen 1, 0 y 0 respectivamente.

Los circuitos secuenciales suelen disponer de una señal que actúa de manera asíncrona y tiene como misión inicializar el circuito. Esta señal, que denominaremos inicio, está conectada a las entradas asíncronas de los biestables (a R o S según si el valor inicial tiene que ser 0 o 1). Al ponerse el circuito en funcionamiento, la señal inicio vale 1 durante un ciclo de reloj (se dice que hace un pulso a 1) y después baja a 0 y se mantiene hasta el final.

Bloques secuenciales

Registro

Hemos visto que un biestable permite guardar el valor de un bit. Para guardar el valor de una palabra de *n* bits, se necesitarán *n* biestables D. La figura siguiente muestra la representación gráfica de un registro.



Se puede ver que dispone de las señales siguientes:

- Una entrada de datos de *n* bits, E; cada uno de los bits de este bus está conectado con la entrada D de uno de los *n* biestables que forman el registro.
- Una salida de datos de *n* bits, S, que es un bus formado por las salidas Q de los *n* biestables que forman el registro.
- Dos entradas de control de un bit, *load* y *clear*; estas dos señales están conectadas respectivamente a la señal *load* y a la entrada asíncrona R de cada uno de los biestables del registro.
- Una entrada de reloj, conectada a las entradas de reloj de todos los biestables.

El funcionamiento de un registro es el siguiente:

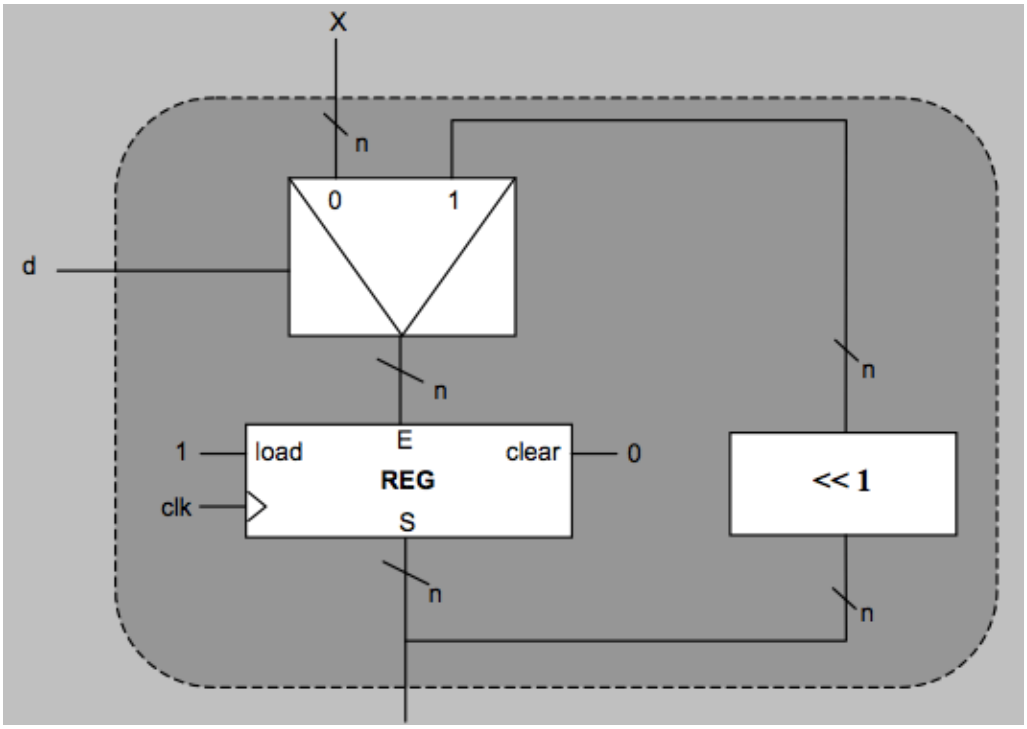
La señal *clear* sirve para poner el contenido del registro a 0. Dado que se conecta con las entradas R de los biestables, es una señal asíncrona, es decir, actúa independientemente del reloj, y es lo más prioritario: cuando está a 1, los *n* bits del registro se ponen a 0, independientemente del valor de las demás señales. Cuando *clear* está a 0, entonces los *n* biestables que forman el registro se comportan como *n* biestables D con señal de carga.

Este funcionamiento se puede expresar mediante esta tabla de la verdad:

clear	load	clk	S^+
1	x	x	0
0	0	x	S
0	1	↓	E

Cuando modificamos el valor de un registro haciendo que se cargue con el valor que hay en la entrada E, decimos que hacemos una escritura. Cuando analizamos el contenido de un registro a partir de la salida S, decimos que hacemos una lectura.

A partir de un registro y bloques combinacionales, se pueden diseñar circuitos con una funcionalidad determinada. Por ejemplo, el circuito de la siguiente figura permite que el registro se pueda cargar con el valor de la entrada X o que pueda desplazar su contenido 1 bit a la izquierda, en función de la señal d.



En los circuitos secuenciales, asumiremos siempre que hay una única señal de reloj (clk). Sin embargo, en las figuras a veces no se conectan todas las entradas de reloj con una misma línea para clarificar el dibujo.

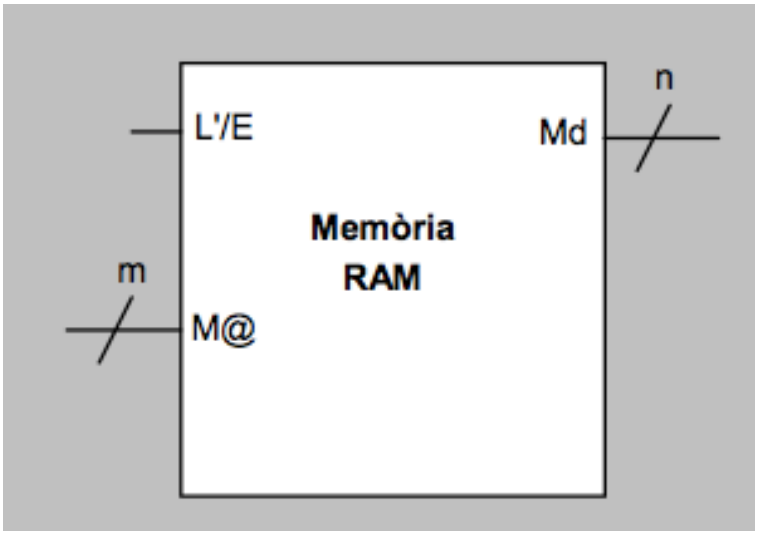
En general, si en un circuito hay más de un punto identificado por un mismo nombre de señal, se entiende que los puntos están conectados, aunque no estén unidos por una línea.

La memoria RAM

La memoria RAM es un bloque secuencial que permite guardar el valor de un cierto número de palabras (2*m*) de un cierto número de bits (*n*).

La denominación RAM proviene del inglés *Random Access Memory* (memoria de acceso aleatorio). Se le dio este nombre porque el tiempo que se tarda en hacer una lectura o una escritura no depende de a qué palabra se esté accediendo (a diferencia de lo que pasaba en otros dispositivos de memoria que se usaban en los primeros computadores).

La siguiente figura muestra la representación de una memoria RAM con un solo puerto de lectura/escritura.



Se puede ver que dispone de las señales siguientes:

- Una entrada de direcciones, M@. Si la memoria tiene capacidad para 2*m* palabras, la entrada de direcciones tendrá *m* bits.
- Una entrada/salida de datos, Md, de *n* bits (si las palabras que guarda la memoria son de *n* bits).
- Una entrada de control, L'/E, que indica en todo momento si se tiene que realizar una lectura o una escritura.

El funcionamiento de una memoria RAM es el siguiente:

- Si L'/E = 0, entonces se realiza una lectura: por el bus Md sale el valor de la palabra que está guardada en la dirección indicada por M@.
- Si L'/E = 1, entonces se realiza una escritura: la palabra indicada por M@ toma el valor que hay en Md (un cierto intervalo de tiempo después de que L'/E se haya puesto a 1).

La tabla de la verdad siguiente resume el funcionamiento de la memoria RAM.

L'/E	
0	Md:= M[M@]
1	M[M@]:= Md

La capacidad de una memoria RAM se suele medir en bytes (palabras de 8 bits). Como hemos dicho, suele contener varios millones de palabras y por eso, para indicar su capacidad, se suelen usar las letras *k*, *M* y *G*, que tienen los significados siguientes:

lletra	significat	exemple
k	$2^{10} = 1024 \cong 10^3$	16 kb (16 kbytes) = 2^{16} bytes
M	$2^{20} = k \cdot k \cong 10^6$	32 Mb (32 Megabytes) = 2^{25} bytes
G	$2^{30} = k \cdot M \cong 10^9$	2 Gb (2 Gigabytes) = 2^{31} bytes

2. Simulación de sistemas digitales

2. 1. Introducción al diseño digital: lenguajes descriptores de hardware

Los diseñadores de hardware usan herramientas de software para el diseño de circuitos digitales. Los diseños normalmente empiezan con especificaciones descritas con lenguajes de descripción de hardware. En este curso, vamos a estudiar uno de los lenguajes más populares, el **VHDL**. A partir de descripciones en VHDL, se pueden construir circuitos de gran complejidad usando herramientas que transforman la descripción del circuito en un conjunto de transistores que finalmente formarán un circuito integrado, también llamado chip.

VHDL es un resultado del programa VHSIC, soportado por el Departamento de Defensa de los Estados Unidos durante las décadas de 1970 y 1980. Inicialmente, el lenguaje fue desarrollado con el objetivo de servir como herramienta de intercambio de diseño entre diferentes diseñadores. Las diferentes descripciones tenían que poder ser entendidas sin equívocos por las dos partes. Más tarde, fue utilizado como herramienta de modelización de diseños, totalmente adecuado para ser una descripción de entrada a simuladores. En 1987, VHDL pasa a ser el estándar IEEE 1076. En 1988, el MilStd454 exige que todos los ASIC relacionados con el Departamento de Defensa estén descritos en VHDL. En 1993, el lenguaje fue revisado y se formó el estándar IEEE 1164. Finalmente, en 1996, el lenguaje VHDL fue definido como un estándar de síntesis de circuitos (IEEE 1076.3).

Principales ventajas de usar VHDL:

- Diseño independiente de dispositivo. Permite la inclusión de diseños (IP, *intellectual properties*) externos. Facilidad de crear bibliotecas. No se necesita conocimiento del dispositivo o tecnología.
- Portabilidad. Permite la transferencia de diseños en diferentes entornos CAD, simulación y síntesis.
- Potencia y flexibilidad. Dada su capacidad de descripción funcional o comportamental, permite una elevada eficacia al ser utilizado en entornos de síntesis automática.

- Facilidad de migración. Diseños concebidos con tecnologías PLD o FPGA pueden ser migrados a ASICS. La descripción de alto nivel del VHDL servirá como elemento de verificación de la implementación final. Interesante también como vehículo de generación de vectores de test en alto nivel.
- Capacidades de *benchmarking.* Permite fácilmente comparar diferentes implementaciones hechas con diferentes descripciones o sintetizadores.
- Rápido *time-to-market.* Combinado con herramientas de síntesis automática, permite llevar un diseño al mercado en tiempo récord.

2.2 Descripción de sistemas digitales con VHDL

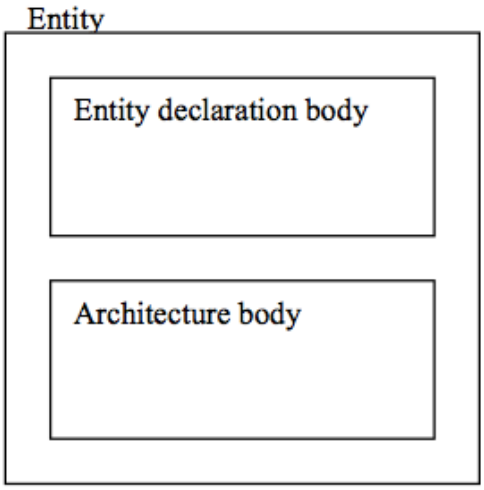
Empecemos por un ejemplo elemental

A continuación, mostramos la descripción VHDL de un comparador de dos palabras de 8 bits:

```
-- comp8 es un comparador de dos palabras de 8 bits

entity comp8 is
        port (a,b: in bit_vector(7 down 0);
                igual: outbit); --igual es activo H
end comp8;

architecture dataflow of comp8 is
begin
        igual <=''1'' when (a=b) else ''0'';
end dataflow;
```

2.2.1 Estructura y organización del lenguaje

Ya sea un sistema completo o una parte de él (diseño jerárquico), un elemento (*entity*) está siempre descrito a partir de dos bloques: una **declaración de interfaz** (*entity declaration body*), donde se declaran las variables de entrada/salida y su tipo, así como una **descripción de su funcionamiento** (*architecture body*), que puede contener cualquier combinación de descripciones comportamental (*behavioral*) y estructural.



Concepto de entity **declaration body** y architecture **body**

En el ejemplo anterior, las palabras de especial significado para el lenguaje (y por lo tanto reservadas) están resaltadas y subrayadas y los identificadores definidos por el diseñador mostrados en cursiva (y así se hace a lo largo de estos apuntes). La descripción VHDL, en el ejemplo, empieza con un comentario indicado por el símbolo --. Después, en las cuatro siguientes líneas de descripción, encontramos la entity **declaration,** donde se declaran las variables de interfaz y su tipo.

Entity declaration

La *entity* ***declaration*** empieza siempre por:

entity *nombre_elemento* **is**

y acaba por

end *nombre_elemento;*

Cada señal de entrada/salida de la entidad está referida como un **puerto.** En la *entity* ***declaration*** hay que declarar estas variables y su modo (*mode*) y tipo (*types*). Los posibles modos son:

- **in,** estas señales entran en la entidad. Corresponden a este tipo las señales reloj, las señales de entrada de control y las señales de datos unidireccionales.

- **out,** corresponde a señales que salen de la entidad, su *driver* forma parte de la entidad y no se realimentan dentro de ella (son salidas limpias de la entidad, es decir, estas señales no se usan en el interior de la entidad).
- **buffer,** corresponde a una señal de salida pero que es utilizada también por la propia entidad. Cuando una señal de salida se usa internamente, la declaración tiene que ser **buffer** y no **out.**
- **inout,** para señales, normalmente de datos, bidireccionales.

Las variables son identificadas mediante una combinación de caracteres alfabéticos, numéricos y guiones inferiores (*underscores*) con las restricciones siguientes:

- el primer carácter tiene que ser una letra
- el último carácter no puede ser un guión inferior
- no puede haber dos guiones inferiores seguidos

Las variables de entrada/salida tienen que ser también declaradas en cuanto al tipo (*type*) de variables. Estos tipos son los siguientes:

- **bit,** variables binarias que pueden tomar los valores 0 y 1,
- **bit_vector,** corresponde a un vector binario, en el ejemplo, a, b **in** bit_vector(7 **down** 0), indica que las variables de entrada (in) a y b son vectores de ocho bits donde la coordenada 7 corresponde al MSB. Los bits individuales (componentes del vector) son señalados como (0), a(1), ……. a(7) y b(7) serán los MSB,

• **std_logic** corresponde a una extensión de bit, definida en la versión 1164 de 1993 del VHDL; estas variables binarias pueden tomar los valores siguientes:

‘U’	no inicializado
‘X’	desconocido
‘0’	0
‘1’	1
‘Z’	alta impedancia
‘W’	desconocido débil (*weak*)
‘La	0 débil
‘H’	1 débil
‘-‘	no importa (*don’t care*)

para utilizar este tipo hay que introducir las dos líneas siguientes en la descripción:

library ieee;
use ieee.std_logic_1164.all;

- **std_logic_vector,** corresponde a un vector de variables std_logic.
- **boolean,** variables booleanas que toman los valores TRUE y FALSE. Se utilizan, normalmente, como variables de regreso de funciones.
- **integer,** corresponde a la declaración de variables organizadas como números enteros decimales multidígito; la anchura de estas variables tiene que ser declarada. Ejemplo: …… integer **range** 0 **to** 1023.

  También se pueden utilizar numeraciones en diferentes bases y en coma flotante, así como organizaciones matriciales. Finalmente, el dominio de una variable discreta también puede ser definido por el usuario, por ejemplo:
  type *formato_mío* **is** (alto, bajo, medio*);* define un nuevo tipo (*type*) discreto determinado por formato_mío.

  A veces, en la entity **declaration** también se definen algunas variables que no son puertos (es decir entradas/salidas del bloque) sino que se utilizan para parametrizar la descripción de los puertos. Esto se hace mediante la indicación **generic.**

  Un ejemplo puede ser:

```
library ieee;
use ieee.std_logic_1164.all;
entity rdff is
        generic     (mida:  integer  := 2);
        port        (clk, reset:     in std_logic,
                     d:              in std_logic_vector(mida-1 downto 0);
                     q:              buffer std_logic_vector(mida-1 downto 0));
end rdff;
```

Architecture body

En el bloque ***architecture body*** se define la función del elemento declarado en *entity* ***declaration*****.** Las variables de los puertos (definidos ya en entity **declaration)** no tienen que ser declaradas nuevamente en esta sección, por lo tanto las

variables de los puertos se pueden utilizar directamente en el architecture **body.** Normalmente, para definir la función de un elemento hay que introducir nuevas variables, que tendrán que ser declaradas en las primeras líneas de la descripción del architecture **body.** Estos nuevos modos son **constant, signal, variable,** a los que deberemos declararles el tipo.

Ejemplo:

```
constant   ample:     integer;
signal     comptador: bit_vector(7 down to 0);
variable   signe:     bit;
```

Els modes **signal** i **variable** poden ser inicialitzats:

```
constant   ample:     integer := 8;
signal     comptador: bit_vector(7 down to 0)  := "1100";
variable   signe:     bit  := '0';
```

(observad que, para asignar un valor a una constante o a un parámetro, se usa el símbolo :=).

Podemos entender el bloque ***entity declaration body*** como una descripción del elemento en forma de caja negra, entradas, salidas y un nombre descriptor. El *bloque* ***architecture body*** nos hace ver qué hay dentro de esta caja y, tanto puede ser a nivel de cómo se comporta (*behavioral*) o como está hecho (estructural) o una combinación de ambos.

En nuestro ejemplo:

architecture *dataflow* **of** *comp8* **is**

La palabra *dataflow* es una elección nuestra para definir la forma como estamos definiendo el elemento *comp8.* Cualquier identificador sería válido. En concreto, la descripción dada como ejemplo al principio del componente *comp8* es de tipo comportamental. Indicamos qué hace, o cómo se comporta, pero no cómo está hecho. Hemos elegido el identificador *dataflow* para esta descripción porque especificamos cómo se comporta mediante el flujo de los datos. En esta descripción el operador <= indica asignación.

A continuación, mostramos otra posible descripción comportamental, ahora en forma de un procedimiento secuencial (esta sería la manera como haríamos una versión programada del componente, posiblemente a partir de un µcontrolador; concretamente, en VHDL si no se introducen retardos específicos, la ejecución de la secuencia es de forma instantánea).

```
-- comp8 es un comparador de dues paraules de 8 bits
entity comp8 is
        port    (a,b:   in bit_vector(7 down 0);
                 igual: out bit);          -- igual es actiu H
end comp8;

-- segona manera de descriure comp8
architecture comportamental_nova of comp8 is
begin
compara: process (a,b)
        begin
                if    a = b       then    igual <= '1';
                                  else    igual <= '0';
                end if;
        end process compara;
end comportamental_nova;
```

Observamos que toda descripción vía procedimiento empieza por

nombre_proceso: **process** (lista)

y acaba por

end process *nombre_proceso;*

Las variables mostradas en la lista se denominan variables sensibles. El procedimiento se ejecuta la primera vez y después solo cuando hay algún cambio en alguna de estas variables). La puesta en marcha del procedimiento es insensible a variaciones de variables no contenidas en la lista.

Veamos a continuación una tercera manera comportamental, ahora en forma de proceso booleano:

```
-- comp8 es un comparador de dues paraules de 8 bits
entity comp8 is
        port    (a,b:   in bit_vector(7 down 0);
                 igual: out bit);        -- igual es actiu H
end comp8;

-- tercera manera de descriure comp8
architecture booleana of comp8 is
begin
        igual   <=              not(a(0) xor b(0))
                        and     not(a(1) xor b(1))
                        and     not(a(2) xor b(2))
                        and     not(a(3) xor b(3));
end booleana;
```

Operadores booleanos

Los operadores booleanos en VHDL son **and, or, nand, nor** y **xor**.

Finalmente, presentamos una descripción estructural en un entorno de una cierta librería work.gatespkg que tuviera las puertas (componentes) xnor2 y and4:

```
-- comp8 es un comparador de dues paraules de 8 bits
entity comp8 is
        port    (a,b:   in bit_vector(7 down 0);
                 igual: out bit);        -- igual es actiu H
```

```
end comp8;

-- quarta manera de descriure comp8
use work.gatespkg.all;
architecture estructural of comp8 is
        signal x:       std_logic_vector(0 to 3);
begin
u0:     xnor2   port map (a(0),b(0),x(0));
u1:     xnor2   port map (a(1),b(1),x(1));
u2:     xnor2   port map (a(2),b(2),x(2));
u3:     xnor2   port map (a(3),b(3),x(3));
u4:     and4    port map (x(0),x(1),x(2),x(3),x(4),igual);
end estructural;
```

Asignaciones y temporización

La asignación **<=** que hemos indicado antes, muy utilizada en descripciones comportamentales, puede tener asignado un retardo.

igual <= ‘‘1’’ **after** 8 ns;

indica la asignación del valor 1 a la variable *igual* con un retardo de 8 ns, que de alguna manera nos podría modelar el retardo o tiempo de propagación de un componente finósico.

Comparación de diferentes descripciones

Un entorno como MAX Plus II sintetiza de manera automática una implementación sobre PLD o FPGA a partir de una descripción VHDL. Si bien las descripciones pueden orientar diferentes implementaciones, el sintetizador del entorno es muy eficiente y la variación de recursos utilizados ante diferentes implementaciones es muy exigua.

Indicaciones concurrentes

Hemos visto que las sentencias de un procedimiento se hacen secuencialmente. En diseño electrónico, muchas veces queremos emplear acciones concurrentes en el tiempo. Las descripciones estructurales, toda lista de ecuaciones lógicas o aritméticas, una lista de procedimientos, estructuras de decisión como **when-else** o **case-then**, todas ellas son ejecutadas concurrentemente sin que el orden en el que están definidos tenga ningún efecto.

Operadores aritméticos

VHDL usa los operadores aritméticos siguientes: ***, **, +, -, /, abs, mod, remo**

Operadores relacionales

VHDL usa los operadores relacionales siguientes: **=, <, >, /=, <=, >=**

Como ejemplo:

```
signal a,b:     bit_vector(4 downto 0);
signal c:       integer range 0 to 4;

        if a >= b then
                ex1 <= '1';
        else
                ex1 <= '0';
```

Observad el significado diferente de los símbolos <= o >= en función de si se trata de una condición o de una asignación.

Elementos de control de secuencia

Para implementar bucles en procedimientos, VHDL permite los siguientes elementos de control de secuencia:

- bucles **for**

Com a exemple:

```
for i in 7 downto 0 loop
        fes alguna cosa;
end loop;
```

- bucles **while**

Com a exemple:

```
while i < 7 loop
        fes alguna cosa;
end loop;
```

Elementos de decisión en procedimientos

Estos son **if-the-else** y **case-then**:.

if-then-else

amb una estructura del tipus:

```
if (condició) then
        fes alguna cosa;
else
        fes una altra cosa diferent;
end if;
```

- l'anterior estructura pot ser estesa a un **elseif**:

```
if (condició1) then
        fes alguna cosa;
elseif (condició2) then
        fes una altra cosa diferent;
else
        fes una altra cosa completament diferent;
end if;
```

- **case-when**

Com a exemple:

```
case comptador is
        when "00" =>
                a <= b;
        when "10" =>
                a <= c;
        when others =>
                a <= d;
end case;
```

Lista de palabras reservadas

abs	**case**	**generic**	**nand**	**process**
access	**component**	**group**	**new**	**pure**
after	**configurat**	**guarded**	**next**	**range**
alias	**ion**	**if**	**nor**	**record**
all	**constant**	**impure**	**not**	**register**
and	**disconnect**	**in**	**null**	**reject**
architectu	**downto**	**inertial**	**of**	**rem**
re	**else**	**inout**	**on**	**report**
array	**elsif**	**is**	**open**	**return**
assert	**end**	**label**	**or**	**rol**
attribute	**entity**	**library**	**others**	**ror**
begin	**exit**	**linkage**	**out**	**select**
block	**file**	**literal**	**package**	**severity**
body	**for**	**loop**	**port**	**signal**
buffer	**function**	**map**	**postponed**	**shared**
bus	**generate**	**mod**	**procedure**	**sla**
sll	**then**	**unaffected**	**variable**	**with**
sra	**to**	**units**	**wait**	**xnor**
srl	**transport**	**until**	**when**	**xor**
subtype	**type**	**use**	**while**	

2.3 Del diseño VHDL a la síntesis de sistemas digitales

Ejemplos básicos con VHDL

Inversor

```
entity inv is
  port(x: in bit; y: out bit);
end inv;
architecture rtl of inv is
begin
  y <= not x;
end rtl;
```

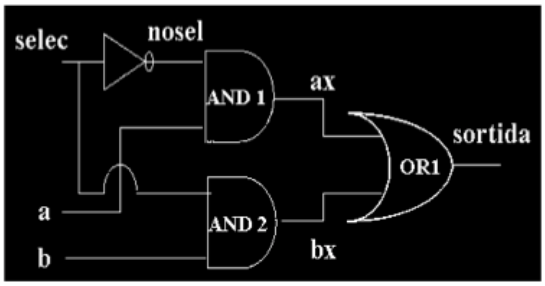
OR

```
entity porta_or is
   port(x,y : in bit; z : out bit);
end port_or;
architecture rtl of porta_or is
begin
   z <= x or y;
end rtl;
```

AND

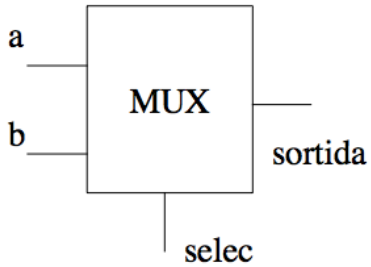
```
entity porta_and is
  port(x,y : in bit; z : out bit);
end porta_and;
architecture rtl of porta_and is
begin
  z <= x and y;
end rtl;
```

▪ ARQUITECTURA: exemple: multiplexor: estil estructural



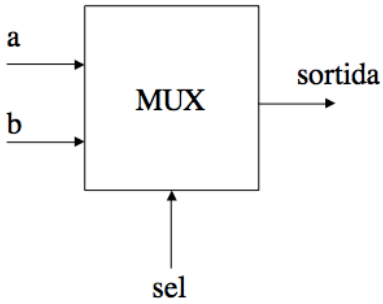

```
architecture estructural of mux is
component inv port(x: in bit; y: out bit);
end component;
component porta_or port(x,y : in bit; z : out bit);
end component;
component porta_and port(x,y : in bit; z : out bit);
end component;
signal ax, bx, nosel: bit;

begin
inv1: inv port map (x => selec, y => nosel);
and1: porta_and port map (x => nosel, y => a, z => ax
and2: porta_and port map (x => selec, y => b, z => bx
or1: porta_or port map (x => ax, y => bx, z => sortida
end architecture estructural;
```

▪ ARQUITECTURA: exemple: multiplexor: estil algorísmic




```
architecture comportamental of mux is
begin
   mux_p: process (a,b,selec)
   begin
      if (selec='0') then
            sortida<=a;
      else
            sortida<=b;
      end if;
   end process mux_p;
end architecture comportamental;
```

▪ ARQUITECTURA: exemple: multiplexor: estil RTL




```
architecture RTL of mux is
begin
   sortida <= a when selec = '0'
                else b;
end architecture RTL;
```

Descripción de lógica síncrona

En los circuitos digitales síncronos, las actuaciones de muchos bloques están condicionadas por la señal reloj (*clock*), que puede ser activa por nivel o por flanco. Ved un ejemplo de descripción de un *flip-flop* de tipo D actuado por flanco de subida (rising_edge-triggered):

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
        port ( d, clk:  in std_logic;
               q:       out std_logic);
end dff;

architecture exemple1_sincron of dff is
begin
        process (clk) begin
                if (clk'event and clk = '1') then
                q <= d;
                end if;
        end process;
end exemple1_sincron;
```

Después de declarar la librería ieee y el estándar 1164, la descripción declara los puertos dentro del entity **declaration body:** se declaran dos entradas, *d* y clk y una salida, *q.*

La descripción comportamental definida en el architecture **body** se hace mediante un procedimiento (*process*) en el que se declara como variable sensible a clk.

Mediante un **if** (*condition*), se condiciona el asignación instantánea de la variable *d* a *q* (estado). La condición es doble, por una parte *clk'event* que es una función que actúa sobre *clk* y devuelve TRUE si hay algún cambio de valor en *clk*, por otra parte la condición *clk* = 1. El and de las dos condiciones equivale a la condición de flanco de subida.

En el caso de un *flip-flop* activo por nivel alto, la condición de la if sería simplemente (*clk* = 1). Observad la importancia de que *clk* esté declarado en la lista de variables sensibles del procedimiento.

Cuando se utiliza std_logic_1164, se pueden usar las funciones rising_edge(variable) y falling_edge(variable).

La función rising_edge(clk) es exactamente equivalente a *clk''event* **and** *clk* = '1' y la función falling_edge(clk) a clk'event **and** *clk* = '0'.

Reset asíncrono

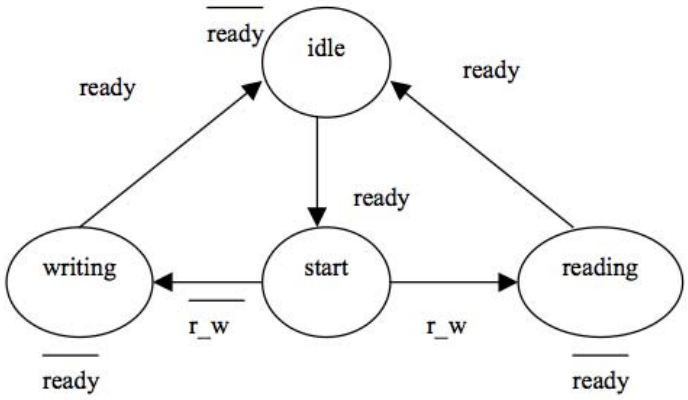
Muchos dispositivos *flip-flop* disponen de una entrada de *reset* asíncrono. A continuación, vemos una descripción de un *flip-flop* de tipo D activo por flanco de subida con capacidad de *reset* asíncrono. Observad cómo se consigue la doble sensibilidad de *clk* y *reset* y cómo la actuación de *reset* es independiente de *clk* y prioritaria usando la orden de ejecución en un elemento **if-elseif** de un procedimiento:

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_amb_reset is
        port ( d, clk, reset:   in std_logic;
                q:      out std_logic);
end dff_amb_reset;

architecture exemple2_sincron_amb_reset_asincron of dff_amb_reset is
begin
        process (clk, reset) begin
                if reset = '1' then
                        q <= '0';
                elseif rising_edge(clk) then
                        q <=d;
                end if;
        end process;
end exemple2_sincron_amb_reset_asincron;
```

Descripción de máquinas de estados

Muchos bloques lógicos constituyentes de un sistema se definen a partir de una máquina de estados finitos (FSM). VHDL permite varias técnicas para describirlos. Muchos de ellos son muy directos en su descripción.

Consideremos el caso de una FSM que actuaría como elemento parcial de un control de un bloque de memoria. La FSM recibe dos variables de entrada: *ready*, que indica cuando la memoria está preparada, y read_write (r_w), que indica si se pretende hacer una lectura o una escritura. La FSM genera dos variables: oe y we, que se aplicarán a las señales *output enable* y *write enable* del bloque de memoria. El diagrama de transición de estados y la tabla de variables de salida en función del estado están indicadas a continuación:



Taula de sortides

estat	oe	we
idle	0	0
start	0	0
writing	0	1
reading	1	0

Una descripción de esta máquina es la siguiente:

```
library ieee;
use ieee.std_logic_1164.all;
entity exemple_FSM is
        port(  r_w, ready:    in      std_logic;
               reset, clk:    in      std_logic;
               oe, we:        out     std_logic);
end exemple_FSM;

architecture implementacio_FSM of exemple_FSM is
        type estats is (idle, start, writing, reading);
        signal estat_actual:   estats;
        begin
        procediment: process (r_w, ready, reset, clk) begin
          if reset= '1' then
                estat_actual <= idle;
          elsif (clk'event and clk = '1') then
                case estat_actual is
                        when idle =>  oe <= '0'; we <= '0';
                                if ready = '1' then
                                        estat_actual <= start;
                                else
                                        estat_actual <= idle;
                                end if;
                        when start => oe<= '0'; we <= '0';
                                if r_w = '1' then
                                        estat_actual <= reading;
                                else

                                        estat_actual <= writing;
                                end if;
                         when reading => oe <= '1'; we <= '0'
                                if ready = 1 then
                                        estat_actual <= idle;
                                else
                                        estat_actual <= reading;
                                end if;
                         when writing => oe <='0'; we <= '1';
                                if ready = 1 then
                                        estat_actual <= idle;
                                else
                                        estat_actual <= writing;
                                end if;
                    end case;
                end if;
            end process procediment;
    end implementació_FSM;
```

Diseño jerárquico en VHDL

El diseño de circuitos y sistemas complejos se suele fragmentar en partes (componentes, subcircuitos) manejables por el diseñador, lo que da lugar a concepto de diseño jerárquico. El entorno **Max Plus II** dispone de una capacidad flexible de jerarquía de partes, dentro de un mismo diseño, que pueden ser definidos con diferentes técnicas.

Algunas de las ventajas más importantes del diseño jerárquico se pueden enumerar así:

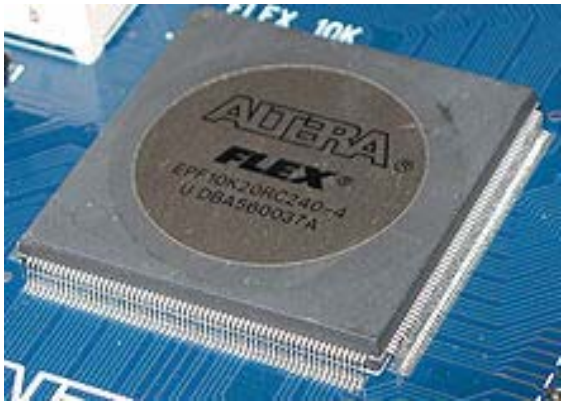
- Una jerarquización de un diseño comporta una buena estructuración y usualmente una mejor definición.
- Cada parte o componente puede ser verificada de manera separada, lo que significa la puesta a punto, verificación, simulación.
- En un entorno, las partes, si están adecuadamente definidas, pueden formar parte de una librería, pueden ser reutilizadas por diferentes diseñadores.
- Permite un diseño paralelo, es decir en colaboración en equipo con otros ingenieros.

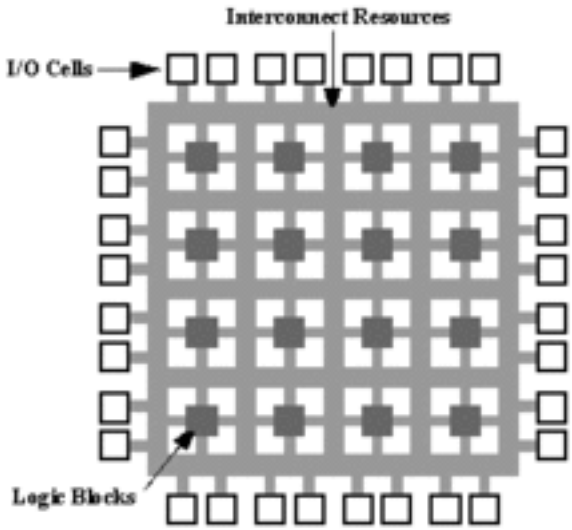
3. Implementación de sistemas digitales sobre dispositivos programables

3.1 Introducción a los dispositivos programables: FPGA

Una **FPGA** (***field programmable gate array***) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada in situ mediante un lenguaje de programación especializado. La lógica programable puede reproducir desde funciones tan sencillas como las que realiza una puerta lógica hasta sistemas complejos en un chip.

Ejemplos físicos de FPGA:





Arquitectura de un FPGA

Los FPGA se utilizan en aplicaciones similares a los ASIC aunque son más lentos, tienen un mayor consumo de potencia y no pueden comprender sistemas tan complejos como ellos. Aun así, los FPGA tienen las ventajas de ser reprogramables (lo que añade una gran flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor.

Ciertos fabricantes cuentan con FPGA que solo se pueden programar una vez, por lo que sus ventajas e inconvenientes se encuentran a medio camino entre los ASIC y los FPGA reprogramables.

Tradicionalmente, los ingenieros han utilizado los FPGA con herramientas de programación hechas por expertos. No obstante, como los FPGA se han vuelto más rápidos y más rentables, los ingenieros e investigadores con poca o ninguna experiencia en diseño de hardware digital están buscando aprovechar los FPGA para crear soluciones personalizadas. Para alcanzar este creciente interés, los proveedores están creando herramientas de más alto nivel que hacen más fácil programar FPGA y brindar los beneficios de la tecnología FPGA a nuevas aplicaciones.

Los FPGA son el resultado de la convergencia de dos tecnologías diferentes, los dispositivos lógicos programables (**PLD** *Programmable Logic Devices*) y los

circuitos integrados de aplicación específica (**ASIC** *application-specific integrated circuit*). La historia de los PLD empezó con los primeros dispositivos PROM (*Programmable Read-Only Memory*) y se les añadió versatilidad con los PAL (*Programmable Array Logic*), que permitieron un mayor número de entradas y la inclusión de registros. Estos dispositivos han seguido creciendo en tamaño y potencia. Mientras tanto, los ASIC siempre han sido potentes dispositivos, pero su uso ha requerido tradicionalmente d una considerable inversión tanto de tiempo como de dinero. Intentos de reducir esta carga han provenido de la modularización de los elementos de los circuitos, como los ASIC basados en cielo, y de la estandarización de las máscaras, tal como Ferranti fue pionero con la ULA (*Uncommitted Logic Array*). El paso final era combinar las dos estrategias con un mecanismo de interconexión que pudiera programar utilizando fusibles, antifusibles o cielo RAM, como los innovadores dispositivos Xilinx de mediados de la década de 1980. Los circuitos resultantes son parecidos en capacidad y aplicaciones a los PLD más grandes, aunque hay diferencias puntuales que delatan antepasados diferentes. Además de computación reconfigurable, los FPGA se utilizan en controladores, codificadores/decodificadores y en el prototipado de circuitos VLSI y microprocesadores a medida.

El primer fabricante de estos dispositivos fue Xilinx y los dispositivos de Xilinx se mantienen como uno de los más populares en compañías y grupos de búsqueda. Otros vendedores en este mercado son Atmel, Altera, AMD y Motorola.

Características

Una jerarquía de interconexiones programables permite a los bloques lógicos de un FPGA ser interconectados según la necesidad del diseñador del sistema, muy parecido a una *breadboard* (placa de uso genérico reutilizable o semipermanente) programable. Estos bloques lógicos e interconexiones pueden ser programados después del proceso de manufactura por el usuario/diseñador, así que el FPGA puede ejercer cualquier función lógica necesaria.

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de los FPGA con microprocesadores y periféricos relacionados para formar un sistema programable en un chip. Ejemplo de estas tecnologías híbridas pueden ser

encontrados en los dispositivos Virtex-II PRO y Virtex-4 de Xilinx, los que incluyen uno o más procesadores PowerPC integrados junto con la lógica del FPGA. El FPSLIC de Atmel es otro dispositivo similar, que utiliza un procesador AVR en combinación con la arquitectura lógica programable de Atmel. Otra alternativa es hacer uso de núcleos de procesadores implementados mediante el uso de la lógica del FPGA. Estos núcleos incluyen los procesadores MicroBlaze y PicoBlaze de Xlinx, Nens y Nens II de Altera, así como los procesadores de código abierto LatticeMicro32 y LatticeMicro8.

Muchos FPGA modernos soportan la reconfiguración parcial del sistema, lo que permite que una parte del diseño sea reprogramada, mientras las otras partes siguen funcionando. Este es el principio de la idea de la computación reconfigurable o los sistemas reconfigurables.

Arquitectura interna de un FPGA

Los FPGA fueron inventados en 1984 por Ross Freeman, cofundador de Xilinx, y surgieron como una evolución de los PLA y de los CPLD. Un **CPLD** (del acrónimo inglés ***complex programmable logic device***) es un dispositivo electrónico; los CPLD extienden el concepto de un PLD (del acrónimo inglés *programmable logic device*) a un mayor nivel de integración puesto que permite implementar sistemas más eficaces y porque utilizan menor espacio, mejoran la fiabilidad del diseño y reducen costes. Un CPLD se forma con múltiples bloques lógicos, cada uno parecido a un PLD. Los bloques lógicos se comunican entre sí utilizando una matriz programable de interconexiones, lo que hace más eficiente el uso del silicio y conduce a una mejor eficiencia a menor coste.

Tanto los CPLD como los FPGA contienen un gran número de elementos lógicos programables. Si medimos la densidad de los elementos lógicos programables en puertas lógicas equivalentes, se podría decir que en un CPLD encontraríamos del orden de decenas de miles de puertas lógicas equivalentes y en un FPGA, del orden de cientos de miles hasta millones de ellas.

Aparte de las diferencias en densidad entre los dos tipos de dispositivos, la diferencia fundamental entre los FPGA y los CPLD es su arquitectura. La arquitectura de los CPLD es más rígida y consiste en una o más sumas de productos programables donde sus resultados van a parar a un número reducido de biestables síncronos (también denominados *flip-flops*). La arquitectura de los FPGA, por otro lado, se basa en un gran número de pequeños bloques utilizados para reproducir sencillas operaciones lógicas, que cuentan a su vez con biestables síncronos. La enorme libertad disponible en la interconexión de dichos bloques concede a los FPGA una gran flexibilidad.

Otra diferencia importante entre FPGA y CPLD es que en la mayoría de los FPGA se pueden encontrar funciones de alto nivel (como sumadores y multiplicadores) intrínsecas a la propia matriz de interconexiones, así como bloques de memoria.

Los FPGA se utilizan en aplicaciones similares a los ASIC, pero tienen los siguientes inconvenientes y ventajas respecto a ellos.

Inconvenientes:

- son más lentos,
- consumen más potencia,
- no pueden realizar sistemas tan complejos que contienen puertas lógicas con un comportamiento reprogramable, lo que permite la implementación de dispositivos lógicos.

Ventajas:

- son reprogramables,
- tienen costes de desarrollo y adquisición mucho más bajos,
- tiempo de desarrollo menor.

Históricamente, los FPGA surgen como una evolución de los conceptos desarrollados en los CPLD.

Los FPGA también se pueden diferenciar por utilizar diferentes tecnologías de memoria:

- **Volátiles**: basadas en RAM. Su programación se pierde al eliminar la alimentación. Requieren una memoria externa no volátil para configurarlas al arrancar (antes o durante el *reset*).
- **No volátiles**: basadas en ROM. Las hay de dos tipos, las reprogramables y las no reprogramables:

    1. **Reprogramables**: basadas en EPROM o flash. Estas se pueden borrar y volver a reprogramar, aunque con un límite de unos 10.000 ciclos.
    2. **No reprogramables**: basadas en fusibles. Solo se pueden programar una vez, lo que las hace poco recomendables para trabajos en laboratorio.

En cuanto a la **seguridad**, los FPGA tienen ventajas y desventajas en comparación con los ASIC o microprocesadores seguros. La flexibilidad del FPGA hace que las modificaciones maliciosas que pueda haber durante la fabricación sean de menor riesgo. Para muchos FPGA, el diseño cargado está expuesto mientras se carga (en general, en cada encendido del dispositivo). Para abordar esta cuestión, algunos FPGA soportan el cifrado *bitstream encryption*.

3 .2 Del lenguaje de descripción a la síntesis del dispositivo

Programación FPGA

El trabajo del programador es definir la función lógica que realizará cada uno de los CLB, seleccionar su modo de trabajo de cada IOB e interconectarlo. El diseño de circuitos y sistemas complejos acostumbra a ser fragmentado en partes (componentes, subcircuitos) manejables por el diseñador, lo que da lugar a concepto de diseño **jerárquico**. El entorno **Max Plus II** dispone de una capacidad flexible de jerarquía de partes, dentro de de un mismo diseño, que pueden ser definidos con diferentes técnicas.

En un flujo de diseño típico, un desarrollador de aplicaciones FPGA simulará el diseño en varias etapas durante el proceso de diseño. Inicialmente, la descripción

RTL en VHDL o Verilog se simula mediante la creación de bancos de pruebas para simular el sistema y observar los resultados. Después de que el motor de síntesis ha trazado el diseño a un netlist, el netlist se traduce a una descripción del nivel de la puerta donde la simulación se repite para confirmar la síntesis a cabo sin errores. Finalmente, el diseño se presenta en el FPGA, momento en el que las demoras de propagación se pueden añadir y ejecutar la simulación de nuevo con estos valores de back-anotado en el netlist. El diseñador cuenta con la ayuda de entornos de desarrollo especializados en el diseño de sistemas a implementarse en un FPGA. Un diseño puede ser capturado ya sea esquemático o haciendo uso de un lenguaje de programación especial. Estos lenguajes de programación especiales son conocidos como HDL o *hardware description language* (lenguajes de descripción de hardware). Los HDL más utilizados son los siguientes:

- **VHDL**: tal como hemos visto en el módulo anterior, es un lenguaje definido por el IEEE usado por ingenieros para describir circuitos digitales.
- **Verilog**: es un lenguaje de descripción de hardware usado para modelar sistemas electrónicos.
- **ABEL**: es un lenguaje de descripción de hardware y un conjunto de herramientas de diseño para programar dispositivos lógicos programables.

En un intento de reducir la complejidad y el tiempo de desarrollo en fases de prototipaje rápido, y para validar un diseño en HDL, existen varias propuestas y niveles de abstracción del diseño. Entre otros, **National Instruments LabVIEW FPGA** propone un lenguaje de programación gráfica de alto nivel.

Aplicaciones del FPGA

Cualquier circuito de aplicación específica puede ser implementado en un FPGA, siempre y cuando disponga de los recursos necesarios. Las aplicaciones donde más comúnmente se utilizan los FPGA incluidos los **DSP** (procesamiento digital de señales), radio definida por software, sistemas aeroespaciales y de defensa, prototipo de ASC, sistemas de imágenes para medicina, sistemas de visión para computadores, reconocimiento de voz, bioinformática, emulación de hardware de computadora. Tenemos que saber que su uso en otras áreas es cada vez mayor, sobre todo en aquellas aplicaciones que requieren un alto grado de paralelismo.

Los FPGA encuentran aplicaciones especialmente en cualquier área o algoritmo que pueda hacer uso del alto grado de paralelismo ofrecido por su arquitectura. Uno de los ellos es la rotura de códigos, en particular ataques de fuerza bruta a algoritmos criptográficos.

Además, se usan cada vez más en aplicaciones convencionales de alto rendimiento donde los núcleos computacionales como FFT o Convolución son implementados en un FPGA en vez de en un procesador de uso general.

Existe código fuente disponible (bajo licencia GNU GPL) de sistemas como microprocesadores, microcontroladores, filtros, módulos de comunicaciones y memorias, entre otras. Estos códigos se denominan **cores**.

Fabricantes de FPGA

A principios del 2007, el mercado de los FPGA se ha colocado en un estado donde hay dos grandes productores de FPGA de propósito general y un conjunto de otros competidores que se diferencian para ofrecer dispositivos de capacidades únicas.

- **Xilinx** es uno de los dos grandes líderes en la fabricación de FPGA.
- **Altera** es el otro gran líder.
- Lattice Semiconductor lanzó al mercado dispositivos FPGA con tecnología de 90 nm. Lattice es un proveedor líder en tecnología no volátil, FPGA basadas en tecnología Flash, con productos de 90 nm y 130 nm.
- Actel tiene FPGA basados en tecnología Flash reprogramable. También ofrece FPGA que incluyen mezcladores de señales basadas en Flash.
- QuickLogic dispone de productos basados en antifusibles (son programables solo una vez).
- Atmel es uno de los fabricantes cuyos productos son reconfigurables. (El Xilinx XC62xx fue uno de estos, pero no se fabrican en la actualidad.) Se enfocaron en suministrar microcontroladores AVR con FPGA, todo en el mismo encapsulado.
- Achronix Semiconductor tiene en desarrollo FPGA muy rápidos. En la actualidad, tienen FPGA que funcionan a 1,5GHz.
- MathStar, Inc. ofrece FPGA que ellos denominan FPOA (*field programmable object arrays*).

- Tabula anunció en marzo del 2010 una nueva tecnología FPGA que utiliza la lógica de tiempo multiplexado y la interconexión de mayor potencial de ahorro para aplicaciones de alta densidad.

4. Sistemas de propósito específico

4.1 Introducción a los sistemas de propósito específico

En este módulo nos vamos a centrar en el procesador de señales digitales o **DSP.**

El **procesador de señales digitales** conocido en inglés como **DSP (*digital signal processor*)** es un procesador o microprocesador que incorpora el hardware capaz de ejecutar los algoritmos para el procesamiento digital de una señal de entrada en tiempo real, como puede ser la entrada de la señal de un fichero de audio, para

obtener las operaciones correspondientes y extraer la salida. Como trabaja con señales digitales, necesita un convertidor de las señales analógicas a digitales (ADC) a la entrada y un convertidor digital analógico (DAC) a la salida, normalmente. Como todos los sistemas basados en procesadores programables, necesita una memoria donde guardar los datos con los que va a trabajar y el programa que ejecuta.

Los DSP poseen también varias soluciones vía hardware o software para las instrucciones de trabajo, variable de gran trascendencia en el momento de tratar una señal mostreada. Estas herramientas hacen que los DSP se impongan muchas veces en la construcción de un dispositivo especializado solo para este proceso en particular. Se observa entonces que la ventaja principal de este sistema es el ser diseñado para trabajar con la mayor cantidad de contratiempos posibles y en una determinada cantidad de tiempo.

Si se tiene en cuenta que un DSP puede trabajar con varios datos en paralelo y un diseño e instrucciones específicas para el procesado digital, se puede ver su gran potencialidad para este tipo de aplicaciones. Estas características constituyen la principal diferencia de un DSP y otros tipos de procesadores.

Se utilizan en circuitos relacionados con la imagen, el sonido, las telecomunicaciones y la regulación y el control, como teléfonos móviles, reproductores Mp3, cámaras digitales, sintonizadores de TDT, regulación de la velocidad, posicionamiento de precisión, entre otros, para las funciones de procesado de la señal en tiempo real, como reducción de ruido, filtrado en general, compresión, descompresión, detección y corrección de errores.

Su principal ventaja es la potencia que le proporciona su estructura, que le permite trabajar en paralelo con una memoria de datos de acceso rápido y gran capacidad, gran poder de ejecución gracias a las unidades MAC y ALU, y todo en tiempo real.

Se han desarrollado de forma sostenida durante los últimos cuarenta años, desde que la disponibilidad de computadores hizo posible la aplicación práctica de algoritmos que antes solo podían ser evaluados de forma manual. Las continuas

mejoras tecnológicas han permitido sustituir los circuitos analógicos por circuitos digitales, que ocupan un menor volumen y que están libres de problemas de tolerancia de los componentes, calibración y deriva térmica que afectan a los analógicos.

4.2 Características de los procesadores digitales de señal (DSP)

Arquitectura

Los DSP no utilizan la arquitectura de Von Neumann, donde los datos y los programas están en la misma memoria, sino que hacen uso de la arquitectura Harvard, donde datos y programas están en memorias diferentes. Cada memoria es dirigida con diferentes buses e, incluso, es posible que la memoria de datos tenga diferente anchura que la de programas.

Con la tecnología Harvard, como tenemos memorias diferentes, logramos acelerar la ejecución de las instrucciones, puesto que mientras estamos ejecutando una instrucción (que usa la memoria de datos) podemos empezar a descodificar la siguiente instrucción (que usa la memoria de programa).

Normalmente, los DSP utilizan una arquitectura Harvard modificada mediante tres buses, una para el programa y dos de datos. Así, la CPU puede leer una instrucción y dos operandos a la vez, de este modo se gana en tiempo. Estos buses pueden ser como en los procesadores convencionales de 16, 32 o 64 bits; o anchos de bus tan diferentes como 24, 48 o 56 bits. Como en los procesadores convencionales, también incluye un *program counter* y un *stack pointer*.

Los elementos básicos de un DSP, aparte de la memoria de datos y de programa como ya hemos hablado, son convertidores A/D en las entradas y D/A en las salidas, del mismo modo que dentro del procesador DSP hay multiplicadores y acumuladores, una ALU y registros:

- **ALU:** la **unidad lógica algorítmica (ALU)**, que se encarga de la ejecución de los cálculos algorítmicos.

- **DMA:** memoria de acceso directo que trabaja a frecuencia tan rápida como el procesador.
- **MAC:** multiplicador acumulador encargado de realizar el producto y acumular el resultado.
- **Buses de datos:** en paralelo para dar más potencia.
- **Registros de desplazamiento**
- **Estructura Harvard:** basada en el almacenamiento por separado usando más de un bus.
- ***Pipeline*:** ejecución en cadena.
- **Coma fija/coma flotante**

El 1979, Bello introdujo el primer chip DSP, era el Mac 4 Microprocessor, capaz de procesar señales digitales. Otro avance en PDS fue el Altamira DX-1, que trabajaba utilizando *pipelines* (cadenas de ejecuciones con retraso entre ellas) y que permitía una gran potencia de ejecución. Pero el primero PDS fabricado por Texas Instruments (TI), presentado en 1983, fue uno de los mayores éxitos. Actualmente, Texas Instruments es uno de los fabricantes de PDS más importantes.

Programación

Para programar un DSP, se utiliza un programa que se guarda como un código máquina en el interior del DSP. Si un programador escribiera un programa de DSP utilizando código máquina le sería muy difícil. Por eso, se desarrolló un lenguaje ensamblador para programar los DSP. Sus instrucciones, mnemónicas, son simbólicas y en correspondencia una a una con las instrucciones de máquina. Se utilizan un ensamblador, un enlazador y un compilador que traduce los códigos fuente. Todo esto sirve para traducir el programa escrito en lenguaje ensamblador a los códigos de máquina del DSP. Son los casos de LabVIEW y Matlab.

Aplicaciones de los DSP

Las aplicaciones más habituales en las que se utilizan los DSP son el procesado de audio y vídeo, así como cualquier otra aplicación que requiere el procesado en tiempo real. Con estas aplicaciones, se puede eliminar el eco en las líneas de comunicaciones, lograr hacer más claras las imágenes de órganos internos en los

equipos de diagnóstico médico, cifrar conversaciones en teléfonos móviles para mantener la privacidad, analizar datos sísmicos para encontrar nuevas reservas de petróleo, hacer posible las comunicaciones inalámbricas LAN, el reconocimiento de voz, los reproductores digitales de audio, los módems inalámbricos, las cámaras digitales y una larga lista de elementos que pueden ser relacionados con el procesado de señales.

Aquí tenemos más aplicaciones de los DSP:

• **Verificación de la calidad del suministro eléctrico**: medición del valor efectivo, potencia, factor de potencia, contenido armónico y *ficker*.

• **Radares**: medición de la distancia y de la velocidad de los contactos. Comprensión del polos, lo que permite incrementar la longitud de los polos para aumentar el alcance, manteniendo la resolución en distancia.

• **Sónares**: formación de haces para orientar electrónicamente la reparación de transductores; en modo activo, medición de la distancia, la demarcación y la velocidad de los contactos; en modo pasivo, clasificación de los contactos en función del ruido que producen.

• **Medicina**: reducción del ruido y diagnóstico automático de electrocardiogramas y electroencefalogramas; formación de imágenes en tomografía axial computerizada (escáner), resonancia magnética nuclear y ecografía (ultrasonido).

• **Análisis de vibraciones en máquinas**: para detectar prematuramente el desgaste de rodamientos o engranajes, comparando el análisis espectral de las vibraciones con un espectro de referencia obtenido cuando la máquina no tiene defectos.

• **Oceanografía**: alerta prematura de maremotos o tsunamis cuando se propagan en el océano abierto, en función de las características de estas oleadas que las diferencian de las olas y de las mareas; análisis armónico y predicción de mareas; medida de la energía de las oleadas con el objetivo de dimensionar muelles y otras estructuras sumergidas.

• **Astronomía**: detección de planetas en estrellas lejanas, en función del movimiento oscilatorio que inducen en las estrellas alrededor de las que orbitan.

• **Radioastronomía**: búsqueda de patrones en las señales recibidas por los radiotelescopios para detectar inteligencia extraterrestre (SETI).

• **Imágenes**: mejora de la luz, contraste, color y nitidez, restauración de imágenes borrosas debido al movimiento de la cámara o del elemento fotografiado. Compresión de la información.

Transformada

Uno de los principales beneficios del DSP es que las transformaciones de señales son más fáciles de realizar. La **transformada de Fournier discreta** (TFD) es una de las más importantes. Esta transformada nos permite convertir una señal de dominio de tiempo en una de dominio de frecuencia. La TFD permite un análisis más sencillo y eficaz de la frecuencia, sobre todo en aplicaciones de eliminación de ruido y en otros tipos de filtrado (filtros pasa bajos, pasa altos, pasa banda).

Otra de las transformadas importantes es la transformada del coseno discreta, similar a la anterior en cuanto a los cálculos necesarios para poder obtenerla, pero esta convierte las señales en componentes del coseno trigonométrico. Esta transformada es una de las bases del algoritmo del compresor de imágenes JPEG.

Fuentes y autorías

Material propio del autor, *Esteve Gené Colinas*

Hermida R.; del Corral A.; Pastor E.; Sánchez F. (1998). *Fundamentos de computadores.* Madrid: Editorial Síntesis.

Gajsky, D. D. (1997). *Principios de diseño digital.* Prentice Hall.

Altera Corporation (1995). *Data Book.* San Jose: Altera Corporation.

Armstrong, J. R. (1998). *Chip-level modeling with VHDL.* Englewoods Cliffs: Prentice Hall.

Xilinx Inc. (1994). *Programmable Logic Data Book.* San Jose: Xilinx Inc.

http://logic.ly/

wikipedia: es.wikipedia.org

weble.upc.es/dcise/Practiques

Tutorial en línea de VHDL en inglés: **http://www.vhdl-online.de/tutorial/**

VHDL Cookbook, accesible en ftp://ftp.cs.adelaide.edu.au/pub/VHDL/

http://www.cannic.uab.es/docencia/DSD/IntroduccioVHDL.htm