UOC Universitat Oberta de Catalunya

SOC Servei d'Ocupació de Catalunya

Introducció a l'electrònica digital

Esteve Gené Pujols

25 hores

Generalitat de Catalunya
Departament d'Empresa i Ocupació

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

TAULA DE CONTINGUTS

1. INTRODUCCIÓ AL DISSENY DE SISTEMES DIGITALS

Un computador és una màquina construïda a partir de dispositius electrònics bàsics, interconnectats adequadament. Podem dir que aquests dispositius són les "peces" amb els quals es construeix un sistema digital.
L'objectiu del curs **Introducció a l'electrònica digital** és entendre com està format un computador electrònicament; i per a poder entendre-ho, cal conèixer a fons els dispositius electrònics digitals bàsics. Aquest és l'objectiu d'aquest mòdul i dels següents.

1.1 INTRODUCCIÓ ALS CIRCUITS LÒGICS

Els dispositius electrònics digitals més elementals són les portes lògiques i els blocs lògics, que formen els circuits lògics. Un circuit lògic es pot veure com un conjunt de dispositius que manipulen d'una manera determinada els senyals electrònics que els arriben (els senyals d'entrada), i generen com a resultat un altre conjunt de senyals (els senyals de sortida).

Hi ha dos grans tipus de circuits lògics:

- Els **circuits combinacionals**, que es caracteritzen perquè el valor dels senyals de sortida en un moment determinat depèn del valor dels senyals d'entrada en el mateix moment.
- Els **circuits seqüencials**, en els quals el valor dels senyals de sortida en un moment determinat depèn dels valors que han arribat pels senyals d'entrada des de la posada en funcionament del circuit i fins al mateix moment (tenen, per tant, capacitat de memòria).

L'objectiu fonamental d'aquest mòdul és conèixer a fons els circuits lògics combinacionals; és a dir, saber com estan formats i familiaritzar-s'hi del tot per tal de ser capaços d'utilitzar-los amb agilitat.

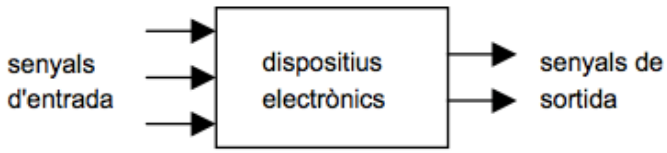
Per a arribar a aquest punt cal haver satisfet els objectius següents:

- Entendre l'**àlgebra de Boole** i les maneres diferents d'expressar funcions lògiques.
- Conèixer les diferents **portes lògiques**, veure com es poden utilitzar per a sintetitzar funcions lògiques i ser capaços de fer-ho. Entendre per què és desitjable minimitzar el nombre de portes i de nivells de portes dels circuits, i saber fer-ho.
- Conèixer la funcionalitat dels diferents blocs combinacionals, i ser capaços d'utilitzar-los en el disseny de circuits.

  En definitiva, després d'aquest mòdul hem de ser capaços de construir fàcilment un circuit qualsevol usant els diferents dispositius que s'hi presenten, i entendre la funcionalitat de qualsevol circuit donat.

1.1.1 CIRCUITS, SENYALS, FUNCIONS LÒGIQUES

Entenem per circuit un sistema format per un cert nombre de senyals d'entrada (cada senyal correspon a un cable), un conjunt de dispositius electrònics que fan operacions sobre els senyals d'entrada (els manipulen electrònicament), i que generen un cert nombre de senyals de sortida. Els senyals de sortida, doncs, es poden veure com a funcions dels senyals d'entrada, i es pot dir que els dispositius electrònics computen aquestes funcions.



En els circuits que formen els sistemes computadors, els cables es poden trobar en dos valors de tensió (voltatge): tensió alta o tensió baixa. Aquest dos valors de tensió s'identifiquen normalment pels símbols 1 i 0 respectivament; de manera que es diu que un senyal val 0 quan en el cable corresponent hi ha tensió baixa, o que

val 1 quan en el cable hi ha tensió alta, també anomenada *tensió d'alimentació*. Els senyals que poden prendre els valors 0 o 1 s'anomenen *senyals lògics* o *binaris*. En un circuit lògic els senyals d'entrada i de sortida són lògics, i les funcions que computa aquest circuit són funcions lògiques.

CODI BINARI

El **codi binari** és un sistema de numeració en el qual totes les quantitats es representen utilitzant com a base dues xifres: zero i un (0 i 1). En altres paraules, és un sistema de numeració de base 2, mentre que el sistema que utilitzem més habitualment és de base 10, o decimal.

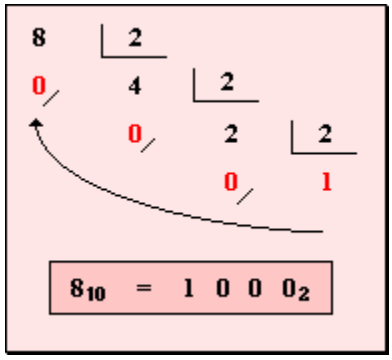
Binaris a decimals

Per a expressar en decimal un nombre *N*, binari, donat, s'ha d'escriure cada nombre que el compon (bit), multiplicat per la base del sistema (base = 2), elevat a la posició que ocupa.

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

Decimals a binari

Per tal de passar un nombre de base 10 a base 2 cal dividir el nombre inicial en base 10 successivament per 2 fins a obtenir un quocient més baix que 2.
Escrivint l'últim quocient i les restes en forma ascendent s'obté el nombre en base 2.



EL COMPLEMENT A DOS

El **complement a dos** d'un nombre ***N*** que, expressat en el sistema binari, està compost per ***n*** dígits, es defineix de la manera següent:

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

Vegem-ne un exemple. Agafem el nombre *N* = 45, que quan s'expressa en binari és $N = 101101_2$, amb 6 dígits, i en calculem el complement a dos:

N = 45, *n* = 6; $2^6 = 64$ i, per tant: $C_2^N = 64 - 45 = 010011_2$

Pot semblar complicat, però és molt fàcil obtenir el complement a dos d'un nombre a partir del seu complement a u, perquè el complement a dos d'un nombre binari és una unitat més gran que el seu complement a u, és a dir:

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

1.1.2 ÀLGEBRA DE BOOLE

Una **àlgebra de Boole** és una entitat matemàtica formada per un conjunt que conté dos elements, unes operacions bàsiques sobre aquests elements, i una llista d'axiomes que defineixen les propietats que compleixen les operacions.

Els dos elements d'una àlgebra de Boole es poden anomenar *fals* i *cert* o, més usualment, *0* i *1*. Així, una variable booleana o variable lògica pot prendre els valors 0 i 1.

Les operacions booleanes bàsiques són:

- la **negació** o **complementació** o **NOT,** que correspon a la partícula "no" i es representa amb una cometa simple ('). Així, l'expressió x' denota la negació de la variable x, i es llegeix "no x".
- el **producte lògic** o **AND**, que correspon a la conjunció "i" de la lògica i es representa pel símbol "·". Així, si x i y són variables lògiques, l'expressió $x \cdot y$ denota el seu producte lògic, i es llegeix "x i y".
- la **suma lògica** o **OR**, que correspon a la conjunció "o" i es representa pel símbol "+". Així, l'expressió $x+y$ denota la suma lògica de les variables x i y, i es llegeix "x o y".

L'operació de negació es pot representar també d'altres maneres. La més usual: $\overline{X}$

És important...

... no confondre els operadors "·" i "+" amb les operacions producte enter i suma entera a les quals estem acostumats. El significat dels símbols el dóna el context en el qual ens trobem. Així, si estem treballant amb enters, 1 + 1 = 2 ("u més u igual a dos"), mentre que si estem en un context booleà, 1 + 1 = 1 ("cert o cert igual a cert", tal com mostra la taula de l'operació 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

Aquestes operacions booleanes bàsiques es poden definir escrivint el resultat que donen per a cada combinació possible de valors de les variables d'entrada, tal com mostra la figura anterior. A l'esquerra de la ratlla vertical de les taules d'aquesta

figura hi ha totes les combinacions possibles de les variables, i a la dreta trobem el resultat de l'operació per a cada combinació.

1.1.3 TAULA DE LA VERITAT

Una taula de veritat expressa una funció lògica especificant el valor que té la funció per a cada combinació possible de valors de les variables d'entrada.

Concretament, a l'esquerra de la taula hi ha una llista amb totes les combinacions de valors possibles de les variables d'entrada; i a la dreta, el valor de la funció per a cadascuna de les combinacions. Per exemple, les taules que havíem vist a la figura d'abans eren, de fet, les taules de veritat de les funcions NOT, AND i OR.

Si una funció té *n* variables d'entrada, i com que una variable pot prendre només dos valors, 0 o 1, les entrades poden prendre 2*n* combinacions de valors diferents. Per tant, la seva taula de veritat tindrà a l'esquerra *n* columnes (una per cada variable) i 2*n* files (una per cada combinació possible); a la dreta hi haurà una columna amb els valors de la funció.

Les files les escriurem sempre en ordre lexicogràfic: és a dir, si interpretem les diferents combinacions com a nombres naturals, escriurem primer la combinació corresponent al nombre 0 (formada per zeros i prou), després la corresponent al nombre 1 i successivament en ordre creixent fins a la corresponent al nombre 2*n* – 1 (formada per uns i prou).

Estructura de la taula de veritat d'una funció de

1 variable

a	f
0	
1	

2 variables

a	b	f
0	0	
0	1	
1	0	
1	1	

3 variables

a	b	c	f
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Estructura de la taula de veritat d'una funció de = Estructura de la taula de veritat d'una funció
1 variable = D'1 variable
2 variables = De 2 variables
3 variables = De 3 variables

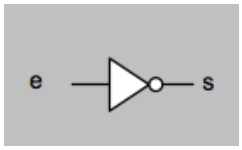
1.2 CIRCUITS LÒGICS COMBINACIONALS

1.2.1 PORTES LÒGIQUES. SÍNTESI I ANÀLISI

Els dispositius electrònics que computen funcions lògiques s'anomenen *portes lògiques*. Són dispositius que estan connectats a un cert nombre de senyals d'entrada i un senyal de sortida.

A continuació presentem el símbol gràfic amb el qual es representa cada porta lògica. En totes les figures que apareixen a continuació, els senyals d'entrada queden a l'esquerra de les portes, i el senyal de sortida, a la dreta.

Porta NOT o inversor
Aquesta porta es representa gràficament amb aquest símbol:

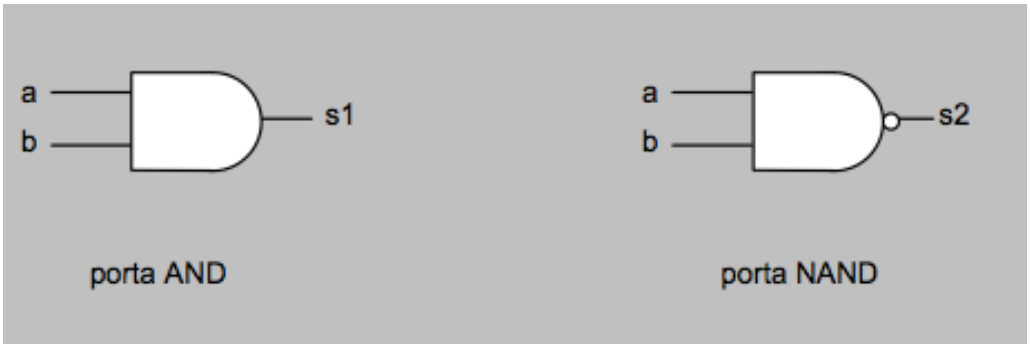


e	***s***
0	1
1	0

El funcionament és el següent: si a l'entrada *e* hi ha un 0 (tensió baixa), llavors a la sortida *s* hi haurà un 1 (tensió alta). I a la inversa, si hi ha un 1 al punt *e*, llavors hi haurà un 0 al punt *s*.

Portes AND i NAND

Es representen gràficament amb aquests símbols:



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

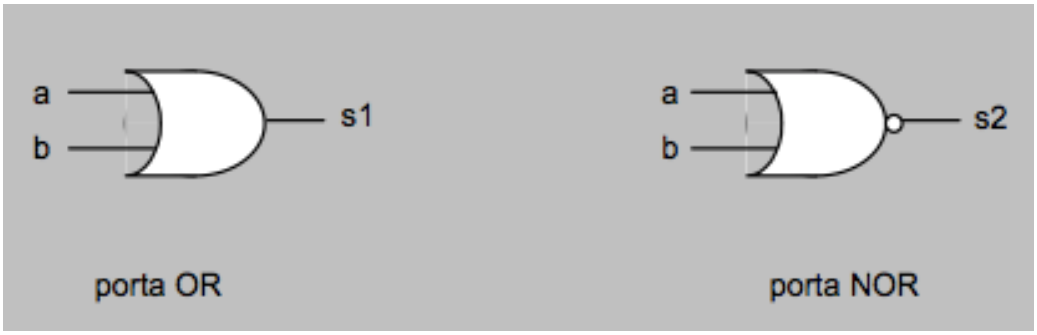
a	***b***	***s*2**
0	0	1
0	1	1
1	0	1
1	1	0

La porta AND implementa la funció lògica AND, és a dir: la sortida *s*1 val 1 només si les dues entrades *a* i *b* valen 1. Per tant, $s1 = a \cdot b$.

La porta NAND implementa la funció lògica NAND. Al punt *s*2 hi ha un 1 sempre que en alguna de les dues entrades *a* o *b* hi hagi un 0. És a dir, $s2 = (a \cdot b)'$.

Portes OR i NOR

Es representen gràficament amb aquests símbols:



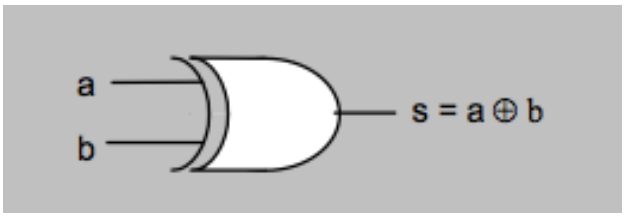
a	***b***	***s*1**
0	0	0
0	1	1
1	0	1
1	1	1

a	***b***	***s*2**
0	0	1
0	1	0
1	0	0
1	1	0

La porta OR implementa la funció lògica OR, és a dir: a la sortida *s*1 hi ha un 1 si qualsevol de les dues entrades està a 1. Per tant, *s*1 = a + *b*.
La porta NOR implementa la funció lògica NOR. Al punt *s*2 hi trobarem un 1 només quan a totes dues entrades hi hagi un 0, és a dir: *s*2 = (a + *b*)'.

Porta XOR

Implementa la funció lògica XOR, és a dir: la sortida *s* val 1 si alguna de les dues entrades val 1, però no si valen 1 totes dues alhora. Es representa gràficament amb aquest símbol:



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

SÍNTESI I ANÀLISI

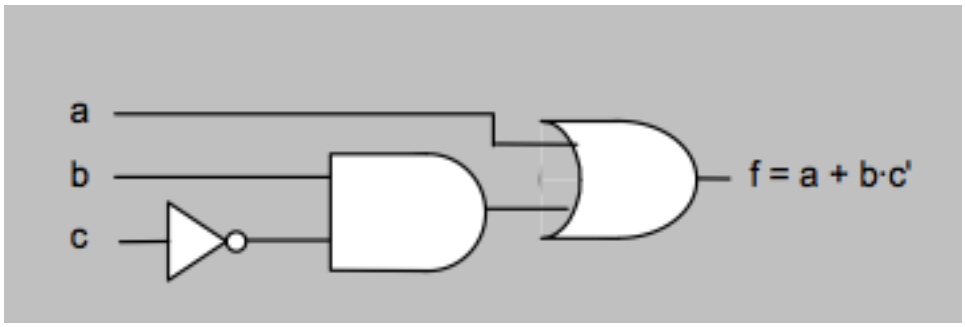
Qualsevol funció lògica es pot implementar usant aquestes portes; és a dir, es pot construir un circuit que es comporti com la funció.

El procés d'obtenir una expressió d'una funció a partir del circuit que la implementa s'anomena *anàlisi*.

Per a **sintetitzar** (o implementar) una funció a partir de la seva expressió algebraica només cal substituir cada operador de la funció per la porta lògica adequada.

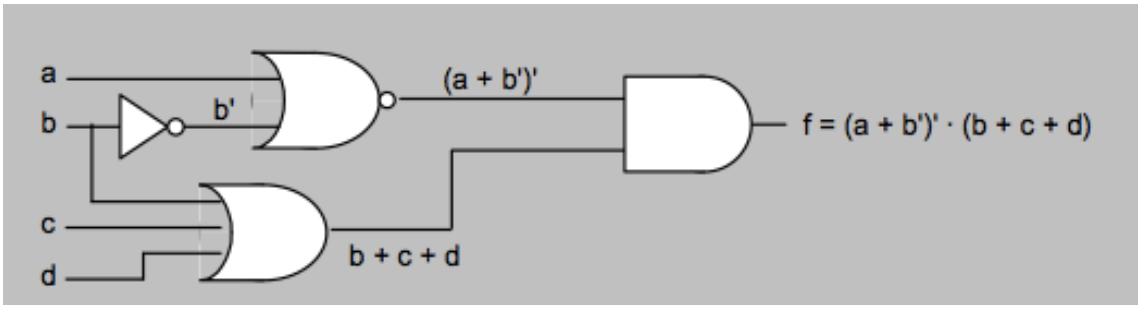
Per exemple, un terme producte de tres variables s'implementarà amb una porta AND de tres entrades. Les portes han d'estar interconnectades entre si i amb les entrades segons el que indiqui l'expressió.

Per exemple, el circuit que implementa la funció $f(a,b,c) = a + bc'$ és el següent:



Un altre exemple. El circuit següent implementa la funció:

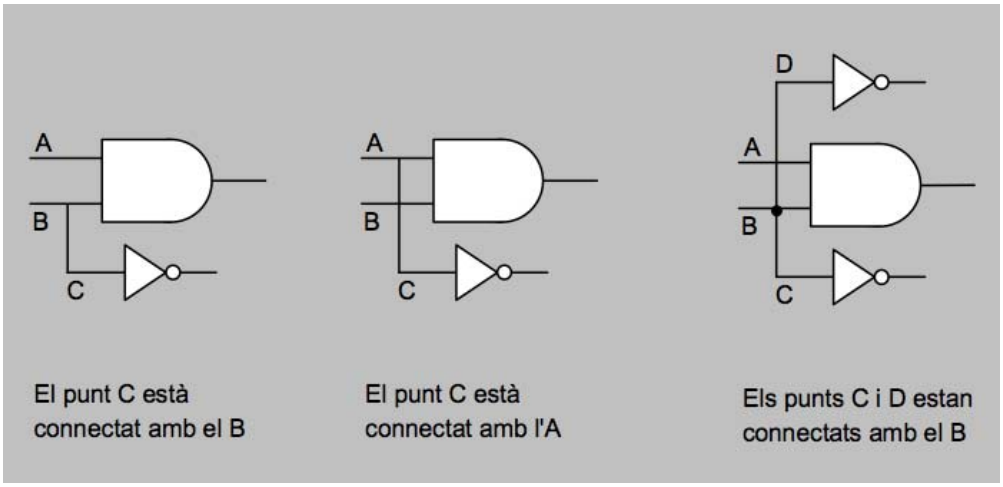
$$f(a,b,c,d) = (a + b')'(b + c + d).$$



En un circuit, si una línia comença sobre una altra línia perpendicular, s'entén que les línies estan connectades (i, per tant, que tenen el mateix valor lògic). Si dues línies perpendiculars es creuen, s'entén que no estan connectades. Si damunt

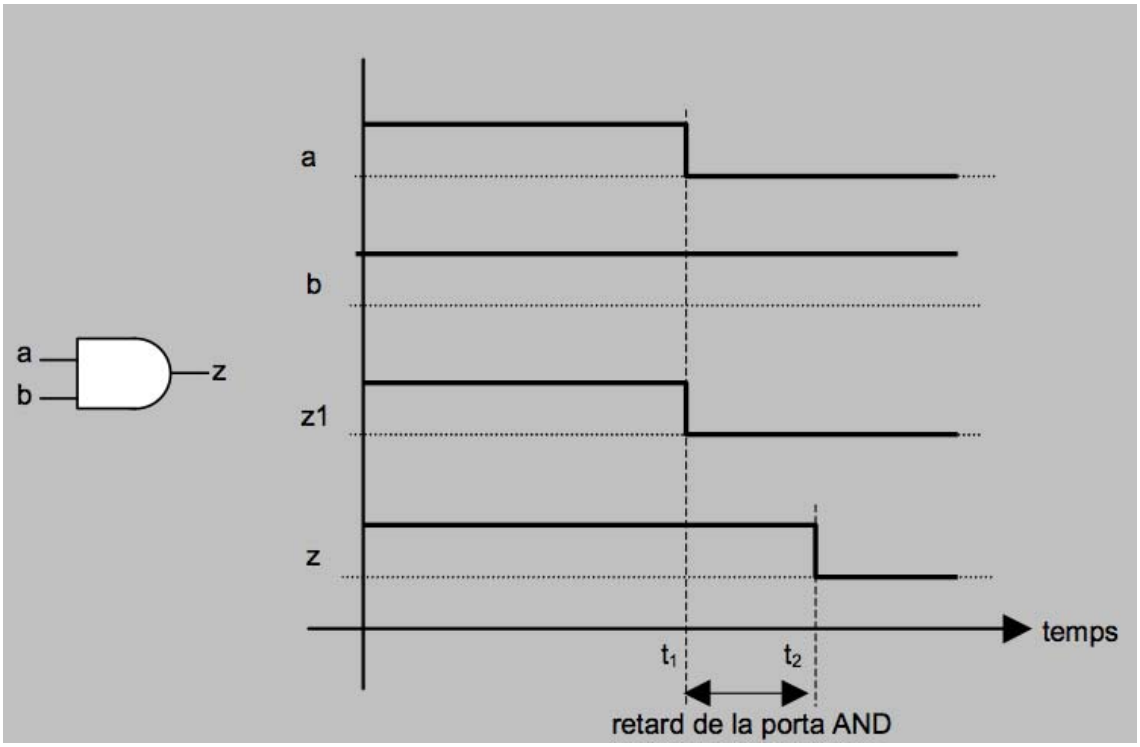
d'una línia s'hi posa un punt, s'entén que totes les línies que el toquen estan connectades.

A continuació en mostrem alguns exemples:



1.2.2 RETARDS. CRONOGRAMES. NIVELLS DE PORTES

Les portes lògiques no responen instantàniament a les variacions en els senyals d'entrada, sinó que tenen un cert retard. La millor manera d'entendre aquest concepte és mirant la figura següent, en la qual hi ha un circuit senzill (només una porta AND) i un cronograma del seu funcionament.

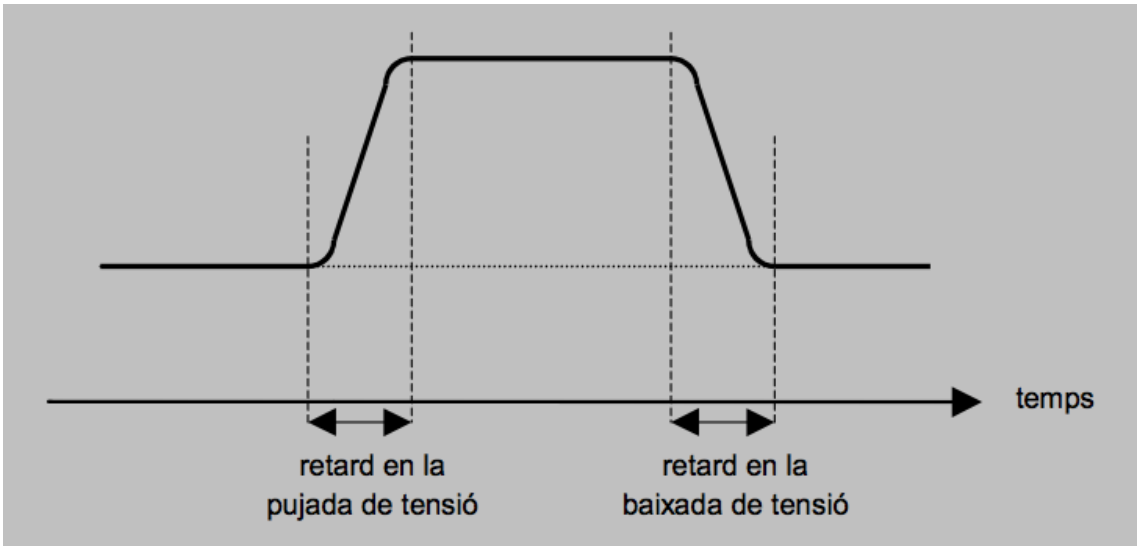


Un cronograma és una representació gràfica de l'evolució dels senyals d'un circuit al llarg del temps.

Suposem que a les entrades *a* i *b* hi tenim connectats dos interruptors que ens permeten en cada moment posar aquests senyals a 0 o a 1. En l'exemple de la figura anterior hem suposat que inicialment tots dos senyals estan a 1, i que en l'instant *t*1 posem el senyal a 0. En aquest moment, el senyal *z* s'hauria de posar a 0, perquè $0 \cdot 1 = 0$. Aquesta situació hipotètica és la que es mostra en el cronograma amb el senyal z_1. No obstant això, en la realitat *z* no es posa a 0 fins a l'instant t2, perquè els dispositius electrònics interns de la porta AND tarden un cert temps a reaccionar. Direm que la porta AND té un retard de $t_2 - t_1$.

Cada porta té un retard diferent, que depèn de la tecnologia que s'hagi usat per a construir-la. Els retards són molt petits, de l'ordre de nanosegons, però cal tenir-los en compte a l'hora de construir físicament un circuit.

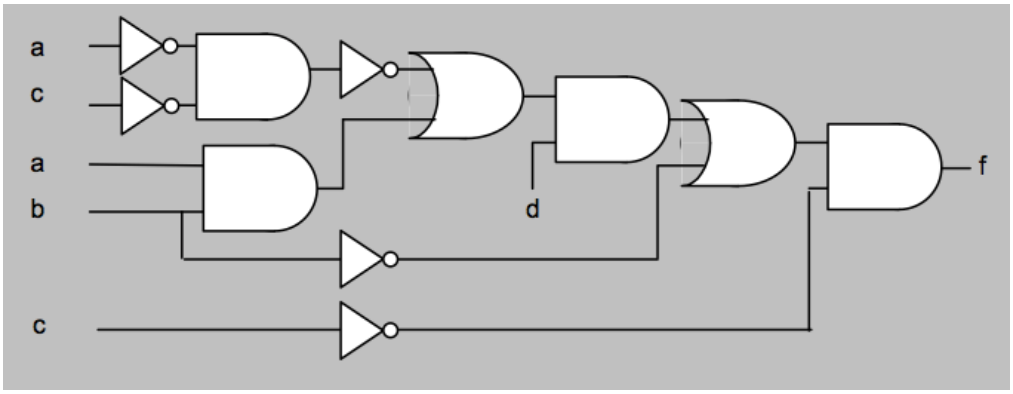
De fet, la transició entre diferents nivells de tensió d'un senyal tampoc no és instantània, sinó que es produeix tal com mostra la figura següent:

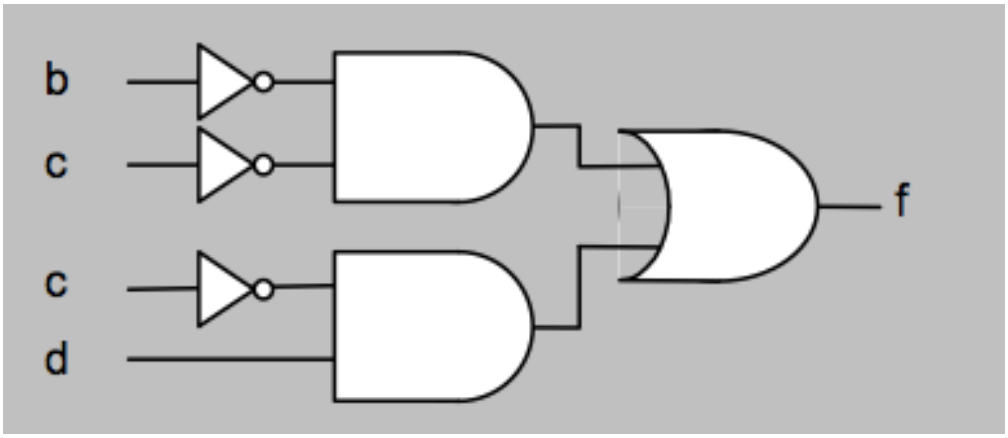


Un dels objectius dels enginyers electrònics que construeixen circuits és que el temps de resposta del circuit sigui tan petit com sigui possible. Com que cada porta té un cert retard, un circuit és en general més ràpid com menys nivells de portes hi ha entre les entrades i les sortides.

El **nombre de nivells de portes** d'un circuit és el màxim nombre de portes que un senyal ha de travessar consecutivament per a generar el senyal de sortida. En comptabilitzar el nombre de nivells de portes d'un circuit no es tenen en compte les portes NOT (per raons que no veurem en aquest curs).

Per exemple, la figura següent mostra el nombre de nivells de portes de dos circuits diferents (podeu comprovar que la funció que implementen és la mateixa):

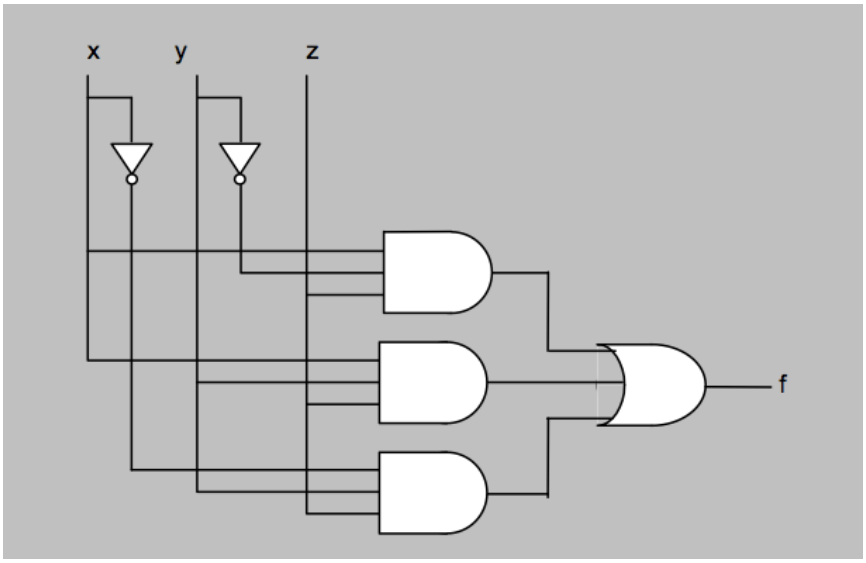




Un bon enginyer escolliria el circuit de la segona figura per a implementar aquesta funció, ja que és més ràpid.

No hi ha una fórmula universal per a trobar l'expressió d'una funció que donarà lloc al circuit més ràpid possible. No obstant això, sabem una manera de garantir que un circuit no tindrà més de dos nivells de portes: partir de l'expressió de la funció en suma de mintermes. En efecte, el circuit corresponent a una suma de mintermes té, a més de les portes NOT, un primer nivell de portes AND (que computen els diferents mintermes) i un segon nivell en el qual hi ha una única porta OR, amb tantes entrades com mintermes té l'expressió. Per exemple, la figura següent mostra el circuit que s'obté en sintetitzar aquesta funció:

$$f = xy'z + xyz + x'yz$$



Els circuits que s'obtenen a partir de les sumes de mintermes s'anomenen *circuits a dos nivells*. Anomenarem *síntesi a dos nivells* el procés d'obtenir-los.

Tenir en compte els retards de les diferents portes i de les transicions entre nivells de tensió és fonamental a l'hora de construir circuits. No obstant això, en aquest curs considerarem que els circuits són ideals, de manera que no es tindran mai en compte els retards, és a dir, s'assumirà sempre que són 0.

1.2.3 BLOCS COMBINACIONALS

Un bloc combinacional és un circuit lògic combinacional amb una funcionalitat determinada. Està construït a partir de portes, com els circuits que hem vist fins ara.

Fins aquest moment, les "peces" que hem fet servir per a sintetitzar circuits han estat portes lògiques. Després d'estudiar aquest capítol, podrem fer servir també els blocs combinacionals com a peces per a dissenyar circuits més complexos, com per exemple un computador, que no es poden pensar si treballem amb portes però sí amb blocs.

MULTIPLEXOR

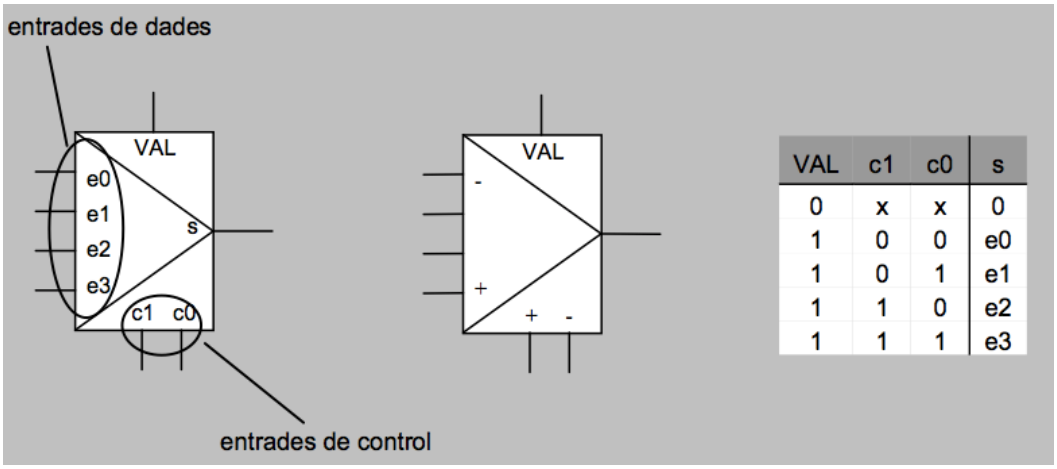
Imaginem que en una ciutat hi ha tres carrers que conflueixen en un altre carrer d'un sol carril. Farà falta un guàrdia urbà o algun tipus de senyalització per a controlar que en cada moment circulin cap al carrer de sortida els cotxes provinents d'un únic carrer confluent.

Un multiplexor és un bloc que fa la funció "d'urbà" en circuits electrònics. Té un cert nombre de senyals d'entrada que "competeixen" per connectar-se a un únic senyal de sortida, i uns senyals de control que serveixen per a determinar quin dels senyals d'entrada es connecta en cada moment amb la sortida.

Més concretament, les entrades i sortides d'un multiplexor són les següents:

- 2*m* entrades de dades, identificades per la lletra *e* i numerades des de 0 fins a 2*m* – 1. Direm que l'entrada de dades numerada amb el 0 és la de menys pes, i la numerada amb el 2*m* – 1, la de més pes.
- Una sortida de dades, *s*.
- *m* entrades de control o de selecció, identificades per la lletra c i numerades des de 0 fins a *m* – 1. Direm que l'entrada de control numerada amb el 0 és la de menys pes, i la numerada amb *m* – 1, la de més pes.
- Una entrada de validació, que anomenem *VAL*.

La figura següent mostra dues representacions gràfiques possibles d'un multiplexor de 4 entrades de dades (*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 diferència és que en la primera hi posem els noms de les entrades, mentre que en la segona només indiquem amb els signes "+" i "–" quines són les de més i menys pes (la resta s'assumeix que estan ordenades en ordre de pes). Totes dues representacions són igualment vàlides.

Per a especificar la mida d'un multiplexor direm quantes entrades de dades té, o bé quantes entrades de control té. Per exemple, un multiplexor de 8 entrades de dades és el mateix que un multiplexor de 3 entrades de control.

Si no dibuixem l'entrada de validació en un multiplexor, assumirem que aquesta està a 1.

El funcionament del multiplexor, és el següent:

- Quan l'entrada de validació val 0, la sortida del multiplexor es posa a 0 (i per tant el valor de les entrades de dades és indiferent, tal com reflecteixen les *x* a la taula de veritat).
- Quan l'entrada de validació val 1, llavors les entrades de control determinen quina de les entrades de dades es connecta amb la sortida, de la manera següent: s'interpreten les variables connectades a les entrades de control (c_1 i c_0 a l'exemple) com un nombre codificat en binari amb *m* bits (l'entrada de més pes correspon al bit de més pes); si el nombre codificat és *i*, l'entrada de dades que es connecta amb la sortida és la numerada amb el nombre *i*.

CODIFICADORS I DESCODIFICADORS

La funció d'un **codificador** és generar la codificació binària d'un nombre donat.

Els codificadors disposen dels senyals següents:

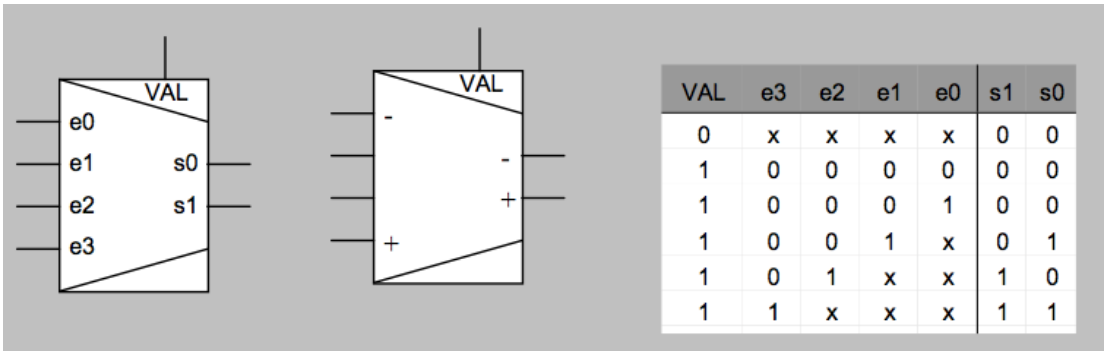
- Una entrada de validació, VAL, que funciona igual que en el cas dels multiplexors: si val 0 totes les sortides valen 0 (quan no dibuixem l'entrada de validació, assumirem que val 1).
- 2*m* entrades de dades (d'un bit), identificades per la lletra *e* i numerades de 0 a 2*m* – 1 (la de nombre més alt és la de més pes)
- *m* sortides de dades (d'un bit), identificades per la lletra *s* i numerades de 0 a *m* – 1, que s'interpreten com si formessin un nombre codificat en binari amb *m* bits (la sortida de més pes correspon al bit de més pes).

  El funcionament d'un codificador és el següent:

  Quan l'entrada de validació val 1, si l'entrada de més pes entre les que estan a 1 és la numerada amb el nombre *i*, llavors les sortides codifiquen en binari el nombre *i*.

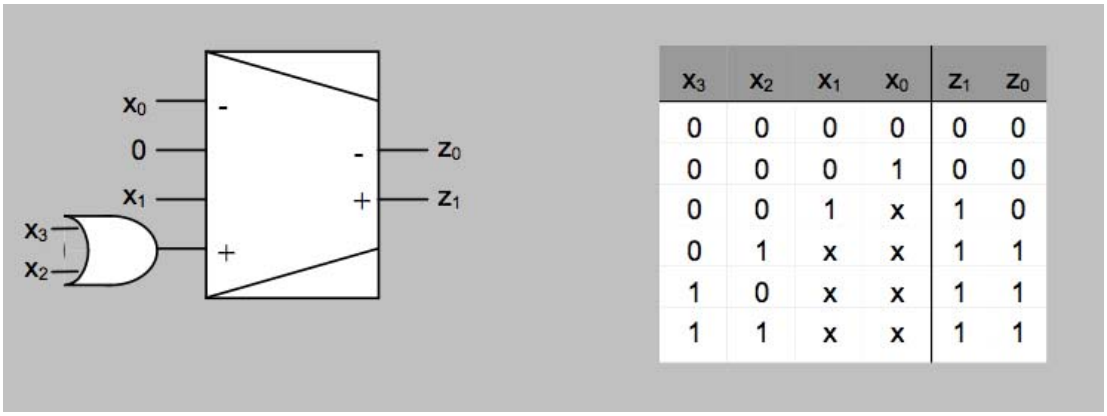
La figura següent mostra la representació gràfica d'un codificador 4-2 i la taula de veritat que n'explica el funcionament (fixem-nos que, com en el cas dels

multiplexors, podem fer servir els símbols "+" i "–" en lloc dels noms de les entrades i sortides).



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 següent mostra un exemple d'ús d'un codificador, i la taula de veritat que descriu el funcionament del circuit.



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

Recordem que si no dibuixem l'entrada de validació en un codificador, assumirem que aquesta està a 1.

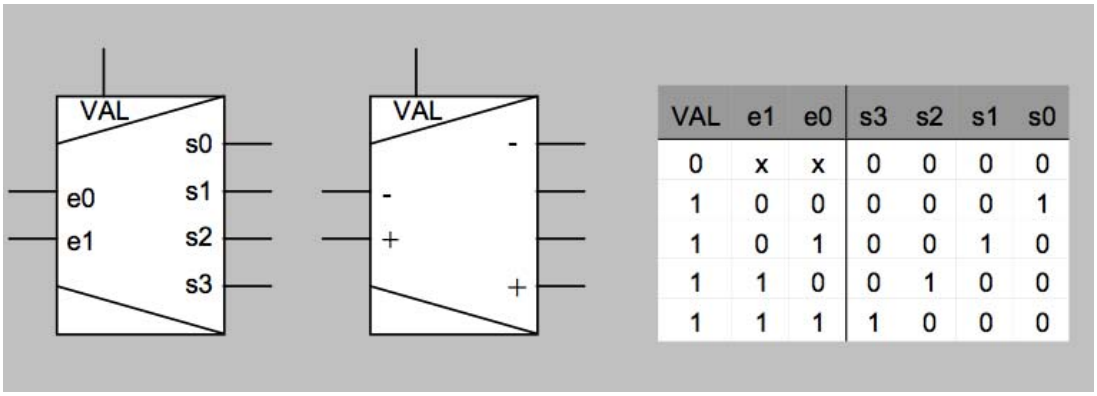
Els **descodificadors** fan la funció inversa als codificadors: donada una combinació binària present a l'entrada, indiquen a quin nombre decimal correspon.

Els descodificadors disposen dels senyals següents:

- Una entrada de validació.

- *m* entrades de dades, identificades per la lletra *e* i numerades de 0 a *m* – 1, que s'interpreten com si formessin un nombre codificat en binari (l'entrada de més pes correspon al bit de més pes).
- 2*m* sortides, identificades per la lletra *s* i numerades de 0 a 2*m* – 1, de les quals només una val 1 en cada moment (si l'entrada de validació està a 0, llavors totes les sortides valen 0).

El funcionament d'un descodificador és el següent:
Quan l'entrada de validació val 1, si les entrades codifiquen en binari el nombre *i*, llavors es posa a 1 la sortida numerada com a *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

Els descodificadors també es poden fer servir per a implementar funcions lògiques. Si la funció té *n* variables, usarem un descodificador *n* – 2*n*, i connectarem les variables a les entrades en ordre de pes. D'aquesta manera, la sortida i del descodificador es posarà a 1 quan les variables, interpretades com a bits d'un nombre en binari, codifiquin el nombre *i*. Per exemple, si connectem les variables [*x*1 x0] a les entrades [*e*1 *e*0] del descodificador de la figura anterior, la sortida *s*2 valdrà 1 quan [*x*1 *x*0] = [1 0].
Per tant, per a implementar la funció només cal connectar a una porta OR les sortides corresponents a les combinacions que fan que la funció valgui 1. Quan alguna d'aquestes combinacions és present a l'entrada del descodificador, la sortida corresponent es posa a 1 i, per tant, de la porta OR en surt un 1. Quan les variables construeixen una combinació que fa que la funció valgui 0, totes les entrades de la porta OR valen 0 i, per tant, també la seva sortida.

DECALADORS LÒGICS I ARITMÈTICS

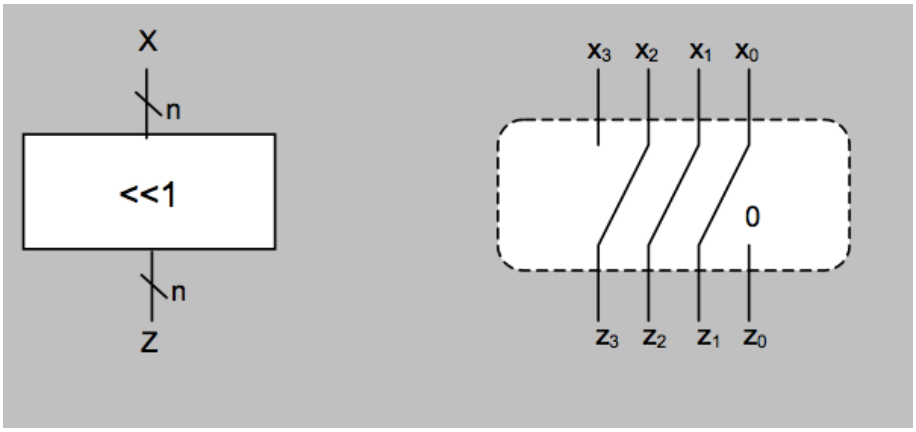
Els decaladors tenen un senyal d'entrada i un senyal de sortida, tots dos de *n* bits. El senyal de sortida s'obté desplaçant els bits d'entrada *m* vegades cap a la dreta o cap a l'esquerra.

• Si el desplaçament és cap a l'esquerra, els *m* bits de menys pes de la sortida es posen a 0.

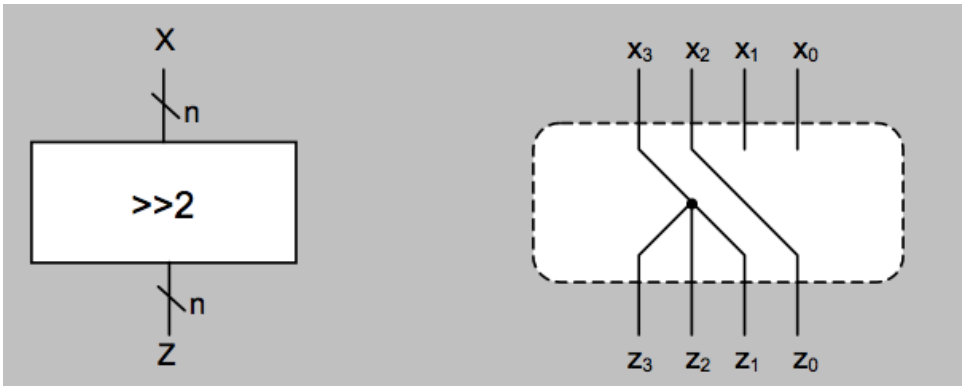
• Si el desplaçament és cap a la dreta, hi ha dues possibilitats per als *m* bits de més pes de la sortida: en els decaladors lògics es posen a 0 en els decaladors aritmètics prenen el valor del bit de més pes de l'entrada.

Fixem-nos que, si interpretem les entrades i sortides com a nombres codificats en complement a dos amb *n* bits, el que fan els decaladors aritmètics és mantenir a la sortida el signe de l'entrada.

Decalador (lògic o aritmètic) d'1 bit a l'esquerra:



Decalador aritmètic de 2 bits a la dreta:



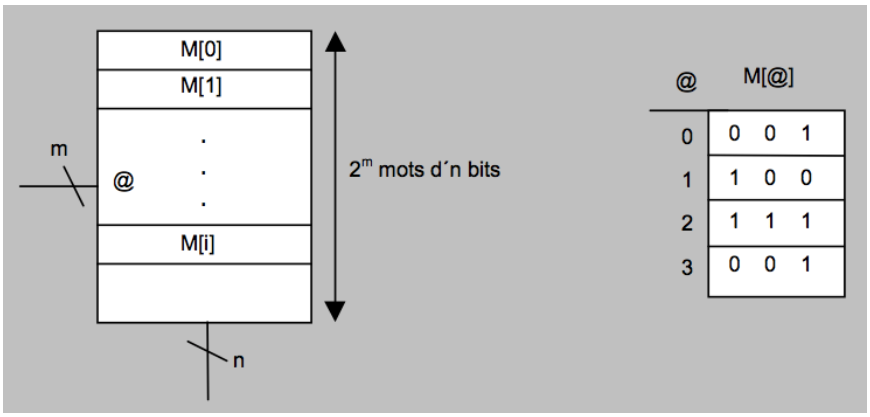
MEMÒRIA ROM

Podem veure una memòria ROM com un arxivador amb calaixos que guarden bits. Cada calaix té una certa capacitat (tots tenen la mateixa), i l'arxivador té un nombre determinat de calaixos, que és sempre una potència de 2.

La memòria ROM té els elements següents:

- 2*m* mots o dades de *n* bits, cadascuna en una posició (calaix) diferent de la memòria ROM. Les posicions que contenen les dades estan numerades des del 0 fins al 2*m* – 1, i aquests nombres s'anomenen *adreces*.
- Una entrada d'adreces de *m* bits, que s'identifica pel símbol @. Els *m* bits de l'entrada d'adreces s'interpreten com a nombres codificats en binari (i per tant cal determinar quin és el pes de cada bit).
- Una sortida de dades de *n* bits.

El funcionament de la ROM és el següent:



Així doncs, només es pot accedir al valor d'un mot (llegir-lo) en cada instant (és com si en cada moment només es pogués obrir un calaix).
La figura mostra un possible contingut d'una memòria ROM de 4 mots de 3 bits. En aquest exemple:

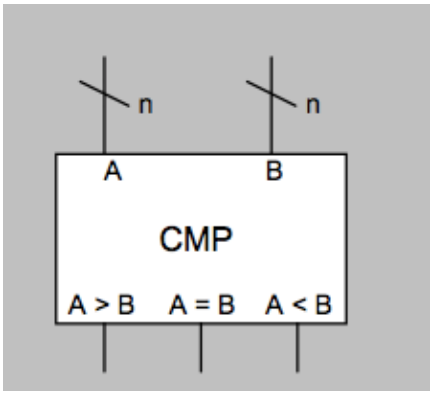
$M[0] = 001$ $M[1] = 100$ $M[2] = 111$ $M[3] = 001$

La memòria ROM és un bloc combinacional que permet guardar el valor de $2m$ mots de n bits.
La denominació **ROM** deriva de l'anglès *read only memory*, perquè es refereix a memòries en les quals no es poden fer escriptures sinó només lectures.
Quan els m bits de l'entrada d'adreces (interpretats en binari) codifiquen el nombre i, llavors la sortida pren el valor de la dada que hi ha emmagatzemada a l'adreça i. Per a referir-nos a aquesta dada usarem la notació $M[i]$, i direm que estem llegint la dada de l'adreça i.

COMPARADOR

El comparador és un bloc combinacional que compara dos nombres codificats en binari i indica quina relació mantenen.
Disposa dels senyals següents:

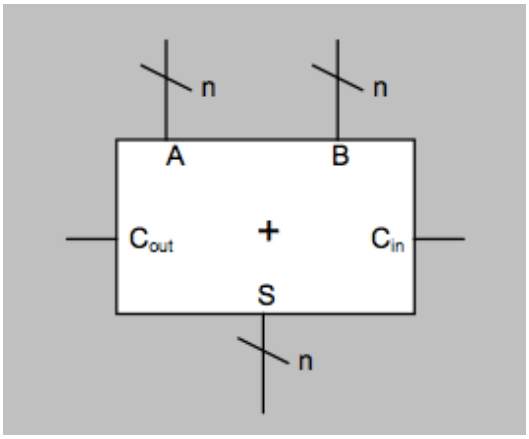
• Dues entrades de dades de n bits, que reben els noms A i B. S'interpreten com a nombres codificats en binari.
• Tres sortides d'un bit, només una de les quals val 1 en cada moment: la sortida $A > B$ val 1 si el nombre que arriba per l'entrada A és més gran que el que arriba per l'entrada B; la sortida $A = B$ val 1 si els dos nombres d'entrada són iguals la sortida; $A < B$ val 1 si el nombre que arriba per l'entrada A és més petit que el que arriba per l'entrada B.



SUMADOR

El sumador és un bloc combinacional que fa la suma de dos nombres codificats en binari o bé en complement a dos.

La figura següent mostra la representació gràfica d'un sumador de *n* bits.



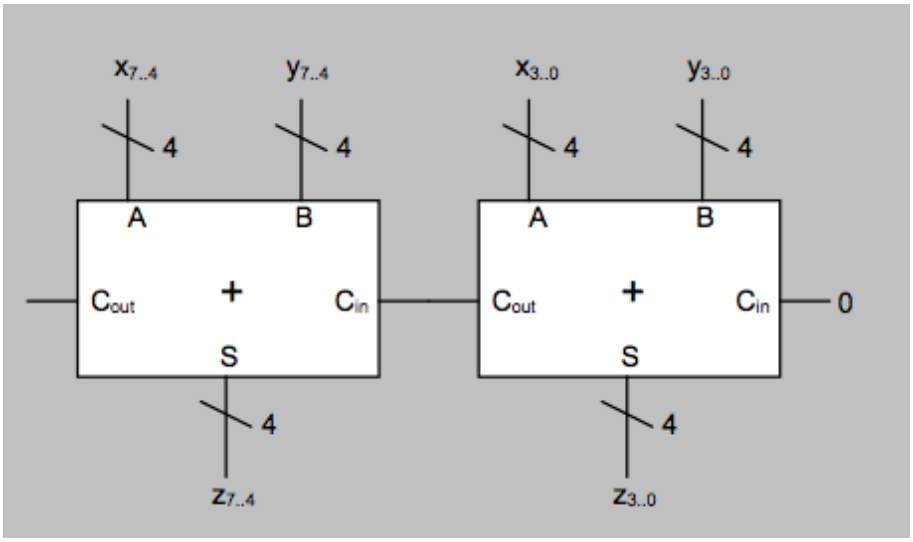
Els senyals de què disposa són els següents:

- Dues entrades de dades de *n* bits, que anomenarem *A* i *B*, per on arribaran els nombres que es volen sumar.
- Una sortida de *n* bits, *S*, que prendrà el valor de la suma dels nombres *A* i *B*.
- Una sortida d'un bit, C_{out}, que valdrà 1 si, en fer la suma, es produeix transport en el bit de més pes.
- Una entrada d'un bit, C_{in}, per on arriba un transport d'entrada.

L'entrada C_{in} és útil quan es volen sumar nombres de $2 \cdot n$ bits i només es disposa de sumadors de *n* bits. En aquest cas s'encadenen dos sumadors: el primer suma

els *n* bits més baixos dels nombres, i el segon, els *n* bits més alts. La sortida C_{out} del primer sumador es connecta amb l'entrada C_{in} del segon, per tal que el resultat sigui correcte.

La figura següent mostra un exemple per al cas $n = 4$, en què fem la suma $Z = X + Y$, en què X, Y i Z són nombres de 8 bits.



Fixem-nos que el sumador que suma els bits més baixos el dibuixem a la dreta, perquè així els bits queden ordenats de la manera com estem acostumats a veure'ls.

Si no necessitem tenir en compte un transport d'entrada, connectarem un 0 a l'entrada C_{in}.

Com ja sabem, el fet de limitar la longitud dels nombres a un nombre de bits determinat (*n*) té com a conseqüència que el resultat d'una suma no sigui sempre correcte (és incorrecte quan es produeix sobreeiximent, és a dir, quan el resultat requereix més de *n* bits per a ser codificat). En les sumes binàries, sabem que si es produeix transport en el bit de més pes llavors el resultat és incorrecte. En canvi, en les sumes en complement a dos no hi ha cap relació entre el transport i el sobreeiximent.

UNITAT LÒGICA ARITMÈTICA (ALU)

Una unitat aritmètica i lògica és un aparell capaç de fer un conjunt determinat d'operacions aritmètiques i lògiques sobre dos nombres d'entrada codificats en binari o en complement a dos.

Les unitats aritmètiques i lògiques s'anomenen *UAL* o, també, *ALU* (de l'anglès *arithmetic and logic unit*).
Per a dissenyar una ALU cal especificar el conjunt d'operacions que volem que faci. Per exemple, una ALU pot fer la suma, la resta, l'AND i l'OR dels operands d'entrada. En cada moment se li ha d'especificar quina és l'operació que ha de fer.

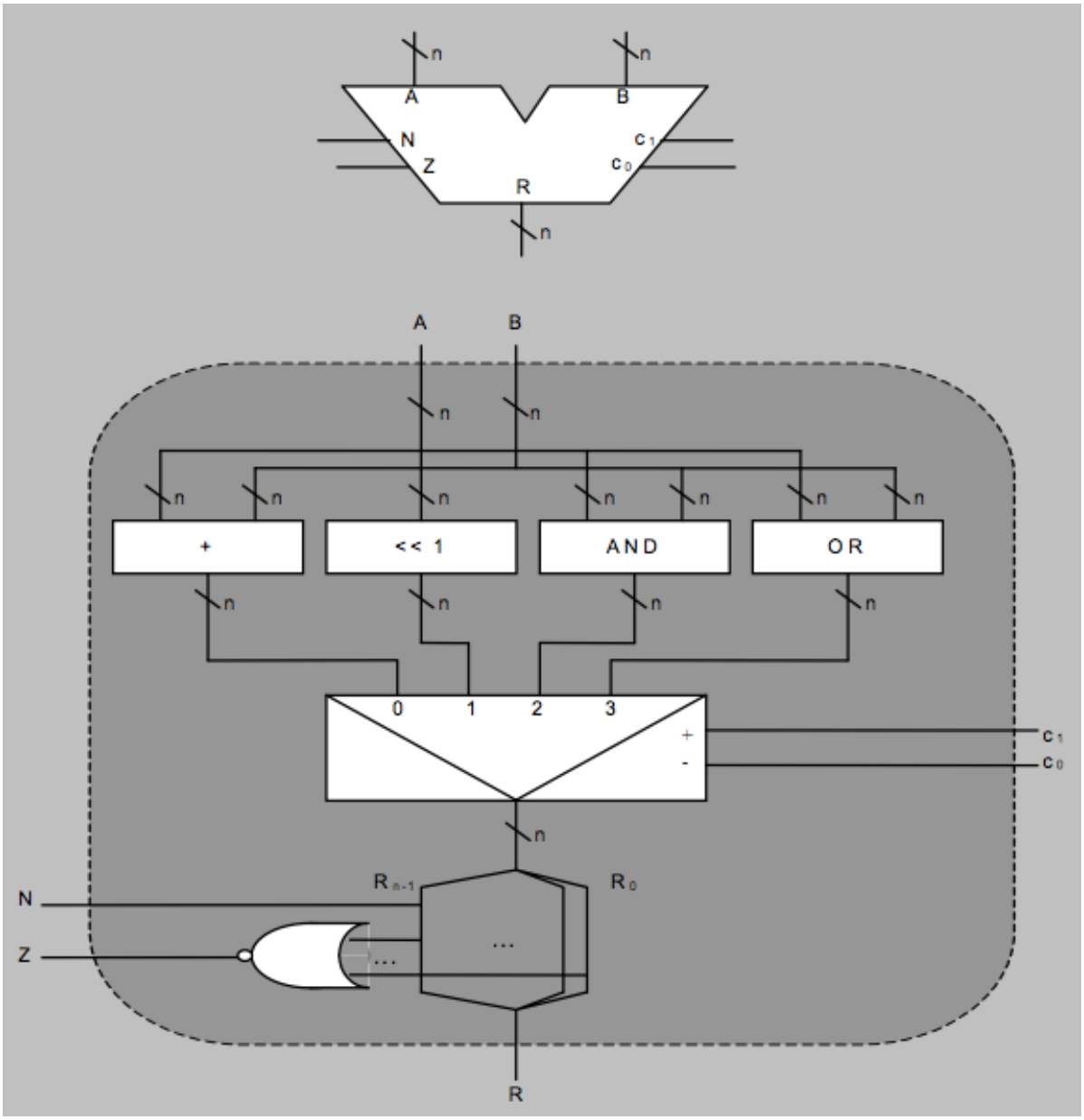
Els senyals de què disposa una ALU són els següents:

- Dues entrades de dades de *n* bits, *A* i *B*, per on arribaran els nombres sobre els quals s'han de fer les operacions.
- Una sortida de dades de *n* bits, *R*, on s'obtindrà el resultat de l'operació.
- Un cert nombre d'entrades de control, *ci*, d'1 bit cadascuna. Si l'ALU és capaç de fer 2*m* operacions diferents, ha de tenir *m* entrades de control. Cada combinació de les entrades de control indica a l'ALU que faci una operació concreta.
- Un cert nombre de sortides d'1 bit, que s'anomenen *bits d'estat*, i que tenen la funció d'indicar algunes circumstàncies que es poden haver produït durant el càlcul.

Els bits d'estat més habituals són els que indiquen si s'ha produït transport en l'últim bit (s'anomena *C*), si s'ha produït sobreeiximent (*V*), si el resultat de l'operació ha estat negatiu (*N*) o si el resultat de l'operació ha estat zero (*Z*).

El bit de sobreeiximent se sol identificar amb les lletres *O* o *V*, que deriven de la paraula anglesa per sobreeiximent, *overflow*.
Per exemple, l'ALU de la figura següent pot fer 4 operacions:

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



En aquesta figura hem disgregat els bits del bus corresponent al senyal de sortida R per a poder dibuixar la implementació dels bits *N* i *Z*. S'ha d'interpretar que tots els bits de *R* es connecten a la porta NOR (de *n* entrades) que computa *Z*.

1.3 CIRCUITS LÒGICS SEQÜÈNCIALS

En el mòdul "Els circuits lògics combinacionals" s'ha vist que els circuits computen funcions lògiques dels senyals d'entrada: el valor dels senyals de sortida en un instant determinat depenen del valor dels senyals d'entrada en el mateix moment. Quan els senyals d'entrada varien, llavors els de sortida també variaran en conseqüència (després del retard introduït per les portes i blocs, que en aquest curs no tenim en compte).

Ara bé, en algunes aplicacions es necessita que el valor dels senyals de sortida no depengui només de les entrades en el mateix moment, sinó que tingui també en compte els valors que han pres les entrades amb anterioritat. En els circuits que hem conegut fins ara, això no és possible: fan falta els elements que formen els circuits lògics seqüencials.

En aquest mòdul coneixerem el concepte de sincronització i estudiarem els biestables o també anomenats *flip-flop* –que són els dispositius seqüencials més bàsics–, i els blocs seqüencials –que es construeixen a partir de biestables i tenen una funcionalitat determinada.

L'objectiu fonamental d'aquest mòdul és conèixer els circuits lògics seqüencials; és a dir, saber com estan formats i poder utilitzar-los amb agilitat.

Per a arribar a aquest punt cal haver satisfet els objectius següents:

- A partir de la funcionalitat que es vol que tingui un circuit lògic, saber discernir si el circuit ha de ser de tipus seqüencial o combinacional.
- Entendre el concepte de memòria, la necessitat d'una sincronització en els circuits lògics seqüencials i el funcionament del senyal de rellotge.
- Conèixer el funcionament del biestable *D* i de totes les entrades de control de què pot disposar.
- Conèixer la funcionalitat dels diferents blocs seqüencials, i saber utilitzar-los en el disseny de circuits.

Després de l'estudi d'aquest mòdul hem de ser capaços de construir un circuit qualsevol que combini tant dispositius seqüencials com combinacionals, i també entendre la funcionalitat d'un circuit que contingui tots aquests elements.

1.3.1 NECESSITAT DE MEMÒRIA EN ELS CIRCUITS LÒGICS

Tenim un circuit amb un senyal d'entrada *X* i un de sortida *Z*, tots dos de *n* bits, que interpretem com a nombres representats en complement a dos. Suposem que volem que $Z = X + 2$. Amb els elements estudiats en el mòdul "Els circuits lògics combinacionals" sabem com fer-ho, i fins i tot de moltes maneres diferents. Quan el valor present a l'entrada *X* varia, llavors *Z* també canvia de valor conseqüentment.

Suposem ara que volem que el valor de *Z* correspongui a la suma de tots els valors que han estat presents a l'entrada *X* durant un interval de temps determinat (durant el qual el valor de *X* ha variat). Amb els dispositius lògics que coneixem fins ara no ho podem aconseguir, perquè en canviar el valor de *X*, el valor anterior ha "desaparegut" i ja no el podem fer servir per a calcular la suma.

Cal que aquest circuit sigui capaç de recordar o de retenir els valors anteriors d'alguns senyals; és a dir, ha de tenir memòria. Aquesta és la funcionalitat que distingeix els **circuits lògics seqüencials** dels **combinacionals**.

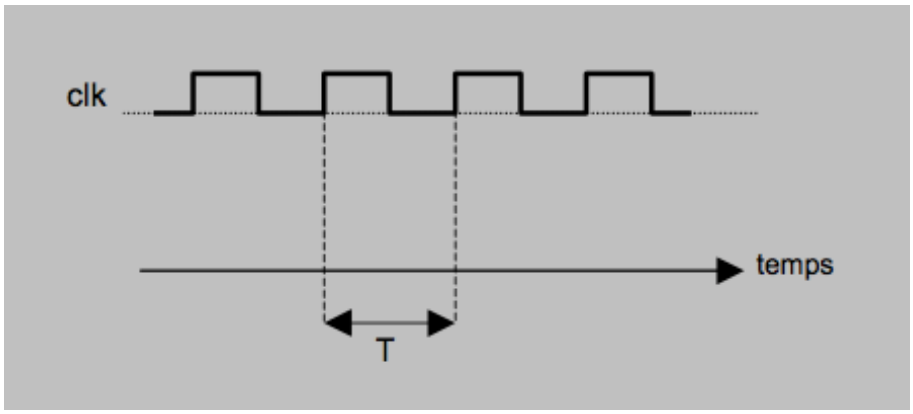
1.3.2 RELLOTGE. SINCRONITZACIÓ

En els circuits combinacionals, l'única noció temporal que intervé és el "present". En canvi, en els circuits seqüencials es té en compte l'evolució temporal dels senyals (i apareix, com veurem més endavant, la noció de "futur").
Ara bé, en la descripció del circuit de l'exemple anterior, què vol dir exactament que "tots els valors que han estat presents a l'entrada X durant un temps determinat"? El senyal *X* pot anar canviant de valor aleatòriament en el temps: pot valdre 13 durant 4 ns; després –25, durant 10 ns; després 0, durant 1 ns, etc. Com pot determinar el circuit en quin moment *X* canvia de valor, és a dir, en quin moment ha de considerar que "*X* ha deixat de tenir el valor antic" i "comença a tenir el valor nou"? Per a poder determinar-ho, el circuit ha de disposar d'un mecanisme de sincronització. En els circuits seqüencials que estudiarem en aquest mòdul s'utilitza un senyal de rellotge com a forma de sincronització.

El **rellotge** és un senyal que serveix per a determinar els instants en què un circuit seqüencial "veu" o "és sensible" al valor dels senyals i respon en conseqüència. Aquesta tasca que duu a terme el senyal de rellotge s'anomena *sincronització dels circuits*.

Concretament, el senyal de rellotge pren els valors 0 i 1 de manera cíclica i contínua des de la posada en marxa d'un circuit i fins que aquest s'atura. Usualment es fa servir la notació ***clk*** per a fer referència al senyal de rellotge (deriva de l'anglès *clock*).

La figura següent mostra el cronograma del senyal de rellotge:



El cicle que forma la seqüència de valors 0 i 1 té una durada determinada i constant, *T*, que s'anomena *període*. Es mesura en segons o, més habitualment, en nanosegons (mil milionèsimes de segon).

Els instants en què el senyal de rellotge passa de 0 a 1 s'anomenen *flancs ascendents*. L'interval de temps que hi ha entre un flanc i el següent s'anomena *cicle* o *cicle de rellotge*. Per tant, la duració d'un cicle és un període, *T* segons.

La freqüència del rellotge és la inversa del període, és a dir, és el nombre de cicles de rellotge que tenen lloc durant un segon. Es mesura en hertzs (cicles per segon); el més habitual és usar el múltiple megahertzs (milions de cicles per segon), que s'abreuja *MHz*. Per exemple, si tenim un rellotge amb un període de 2 ns, la seva freqüència és la següent:

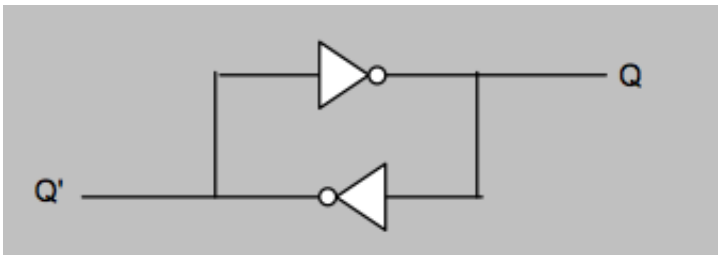
$$\frac{1\,cicle}{2{\cdot}10^{-9}\,segons} = 0'5{\cdot}10^{9}\ \text{cicles/segon} = 500{\cdot}10^{6}\ \text{cicles/segon} = 500\ \text{MHz}$$

El senyal de rellotge pot sincronitzar els circuits de diverses maneres. En aquest curs només veurem la que s'usa més habitualment, anomenada *sincronització per flanc ascendent*. Aquesta forma de sincronització estableix que els dispositius seqüencials d'un circuit seran sensibles als valors dels senyals en els instants dels flancs ascendents, tal com veurem a l'apartat següent.

EL BIESTABLE *D* (o *flip-flop D*)

A l'apartat anterior hem vist la necessitat que els circuits lògics tinguin capacitat de memòria. En aquest apartat veurem com es construeixen els dispositius que poden "recordar" els valors dels senyals.

Examinant el circuit que es mostra a la figura següent:



Veiem que el valor que hi hagi en els punts *Q* i *Q*' (0 o 1) s'hi mantindrà indefinidament, ja que la sortida de cada inversor està connectada amb l'entrada de l'altre. Per tant, podem dir que aquest circuit és capaç de "recordar", o de mantenir en el temps, un valor lògic.

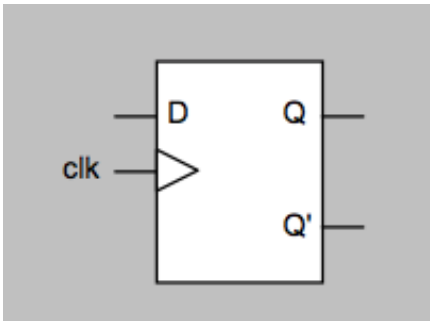
Ara bé, aquest circuit no és gaire útil, perquè no admet la possibilitat de modificar el valor recordat. Interessa dissenyar un circuit que tingui aquesta mateixa capacitat de memòria, però que a més permeti que el valor en el punt Q pugui canviar

segons els requeriments de l'usuari. Un circuit amb aquestes característiques s'anomena *biestable*.

Els **biestables** són els dispositius de memòria més elementals: permeten guardar un bit d'informació.
Un **biestable** té dues sortides, *Q* i *Q*'. Es diu que *Q* és "el valor que guarda el biestable" en cada moment, o "el valor emmagatzemat al biestable", i que *Q*' és la seva negació.

Hi ha diferents tipus de biestables. En aquest curs només en veurem un, **el biestable *D***.

La figura següent en mostra la representació gràfica:



Podem observar que disposa d'una entrada de rellotge, ja que es tracta d'un dispositiu seqüencial.

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

La sortida *Q* pren el valor que hi ha a l'entrada *D* en cada flanc ascendent de rellotge. Durant la resta del cicle, el valor de *Q* no canvia.

És a dir, el biestable només és sensible al valor present en l'entrada *D* en els instants dels flancs ascendents.

La figura següent mostra la taula de veritat que descriu el comportament del biestable *D* (no hi posem la columna corresponent a *Q*', perquè és la negació de *Q*).

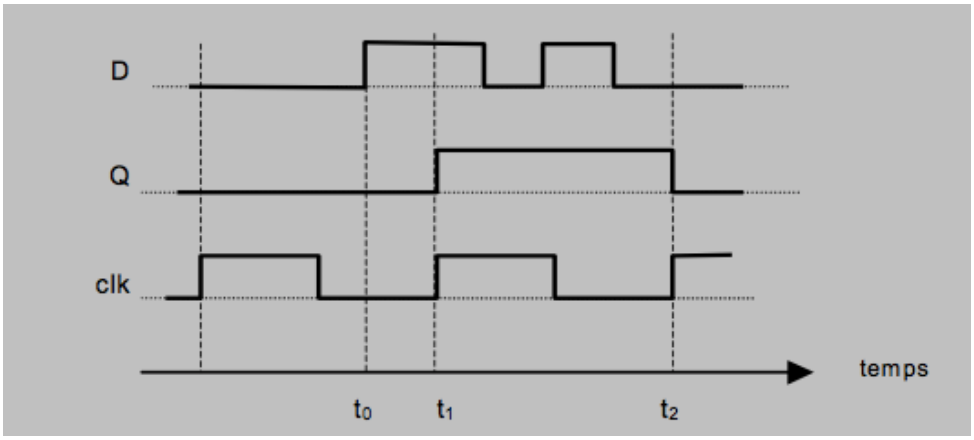
D	clk	Q^+
0	↓	0
1	↓	1

Per tant, *Q*+ no identifica cap senyal del circuit, sinó el valor del senyal *Q* en un instant futur: a partir del moment en què es produeixi el proper flanc. Aquesta notació, doncs, ens permet descriure amb precisió l'evolució temporal dels senyals en un circuit lògic seqüencial.

En aquesta figura s'introdueixen algunes notacions que s'usaran d'ara en endavant:

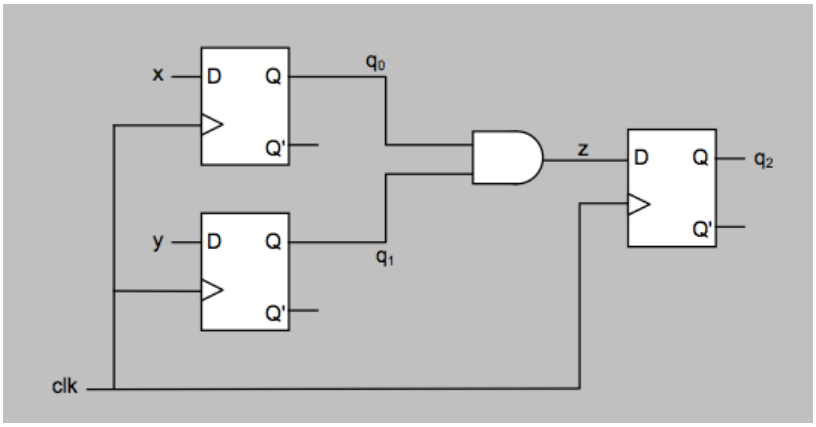
- el símbol "↓" representa un flanc ascendent de rellotge
- el símbol "$^+$" a la dreta del nom d'un senyal es refereix *al valor que prendrà aquest senyal quan es produeixi el proper flanc ascendent de rellotge.*

La figura següent mostra el cronograma del comportament d'un biestable D:

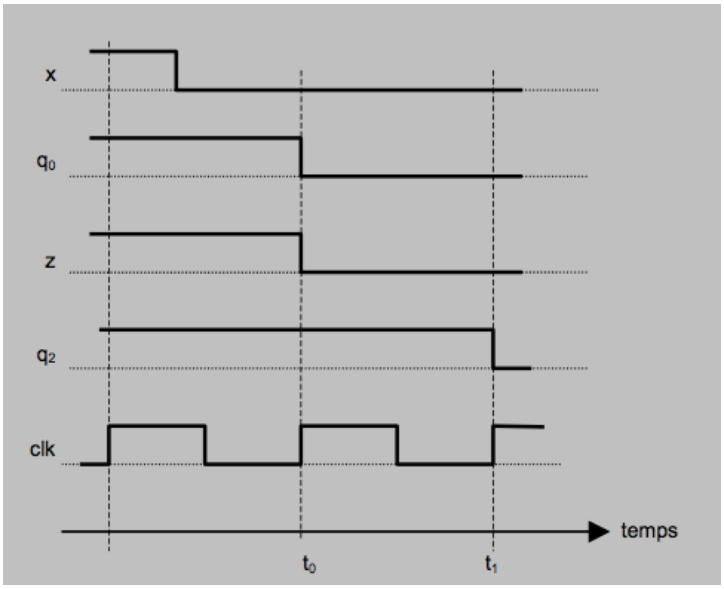


Es pot observar que, tot i que *D* es posa a 1 en l'instant t_0, *Q* no canvia de valor fins a l'instant t_1, perquè no és fins en aquest moment que es produeix un flanc ascendent de rellotge.
Malgrat les subseqüents variacions de *D*, *Q* es manté inalterat fins a l'instant t_2, en què es produeix el proper flanc de rellotge.

En els circuits reals, és habitual fer alguna funció combinacional sobre la sortida d'un biestable o més i connectar el resultat a l'entrada d'un altre biestable. La figura següent en mostra un exemple:



El cronograma de l'evolució d'aquest circuit durant un cert interval de temps (per tal de simplificar el dibuix, no hi hem posat ni *y* ni q_1; assumim que tots dos senyals es mantenen a 1 tota l'estona) és el següent:



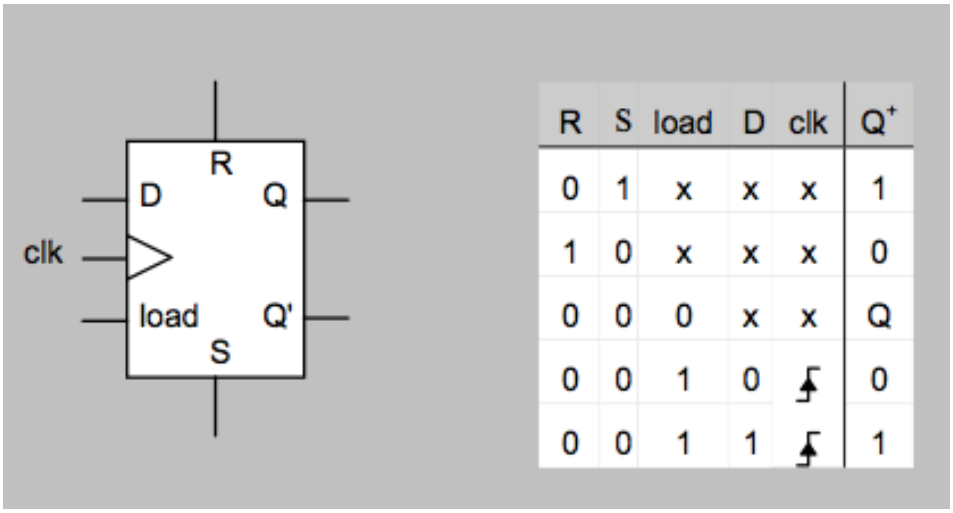
En l'instant t_0, q_0 es posa a 0, perquè a l'entrada *D* del biestable corresponent hi ha un 0 (x = 0); en conseqüència, z també es posa a 0. En dibuixar la línia del cronograma corresponent a q_2, podríem dubtar de si s'ha de posar a 0 en aquest mateix instant, ja que sobre la línia vertical corresponent a t_0, z està tant a 1 com a 0.

El valor d'un biestable *D* pot variar en els instants de flancs ascendents segons el valor que hi ha a les entrades *D* i *clear*. Ara bé, cal tenir la capacitat de donar-li un valor inicial: el valor que prendrà en posar-se en marxa un circuit.
Les **entrades asíncrones** d'un biestable permeten modificar-ne el valor instantàniament, independentment del valor del senyal de rellotge i de les entrades *D* i *clear*. Es diu que les entrades asíncrones tenen més prioritat que la resta d'entrades.

Els biestables solen disposar de dues entrades asíncrones:

- *R* (de l'anglès *reset*): quan es posa a 1, el biestable es posa a 0.
- *S* (de l'anglès *set*): en el moment en què es posa a 1, el biestable es posa a 1.

La figura següent mostra la representació gràfica d'un biestable *D* amb entrades asíncrones i senyal de càrrega, i la taula de veritat que descriu el seu comportament:



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

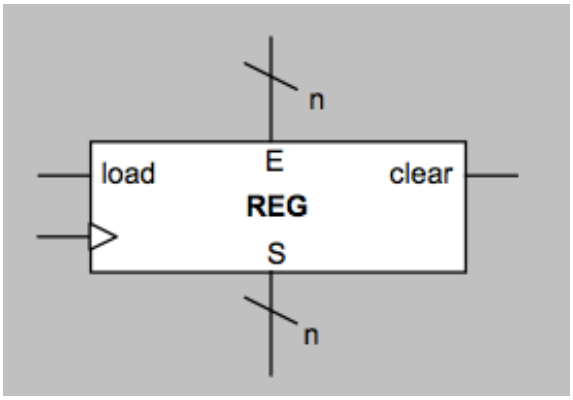
Quan en un circuit no dibuixem els senyals *clear*, *R* i *S* d'un biestable, assumirem per defecte que valen 1, 0 i 0 respectivament.

Els circuits seqüencials solen disposar d'un senyal que actua de manera asíncrona i que té per missió inicialitzar el circuit. Aquest senyal, que anomenarem *Inici*, està connectat a les entrades asíncrones dels biestables (a *R* o *S* segons si el valor inicial ha de ser 0 o 1). Quan el circuit es posa en funcionament, el senyal Inici val 1 durant un cicle de rellotge (es diu que fa un pols a 1), i després baixa a 0 i s'hi manté fins al final.

BLOCS SEQÜENCIALS

Registre

Hem vist que un biestable permet desar el valor d'un bit. Per a desar el valor d'un mot de *n* bits, caldran *n* biestables *D*.
La figura següent mostra la representació gràfica d'un registre.



Es pot veure que disposa dels senyals següents:

- Una entrada de dades de *n* bits, E. Cadascun dels bits d'aquest bus està connectat amb l'entrada *D* d'un dels *n* biestables que formen el registre.
- Una sortida de dades de *n* bits, *S*, que és un bus format per les sortides *Q* dels *n* biestables que formen el registre.

- Dues entrades de control d'un bit, *clear* i *clear*. Aquests dos senyals estan connectats respectivament al senyal *clear* i a l'entrada asíncrona *R* de cadascun dels biestables del registre.
- Una entrada de rellotge, connectada a les entrades de rellotge de tots els biestables.

El funcionament d'un registre és el següent:

El senyal *clear* serveix per a posar el contingut del registre a 0. Com que es connecta amb les entrades *R* dels biestables, és un senyal asíncron, és a dir, actua independentment del rellotge, i és el més prioritari: quan està a 1 els *n* bits del registre es posen a 0, independentment del valor dels altres senyals. Quan *clear* està a 0, llavors els *n* biestables que formen el registre es comporten com a *n* biestables *D* amb senyal de càrrega.

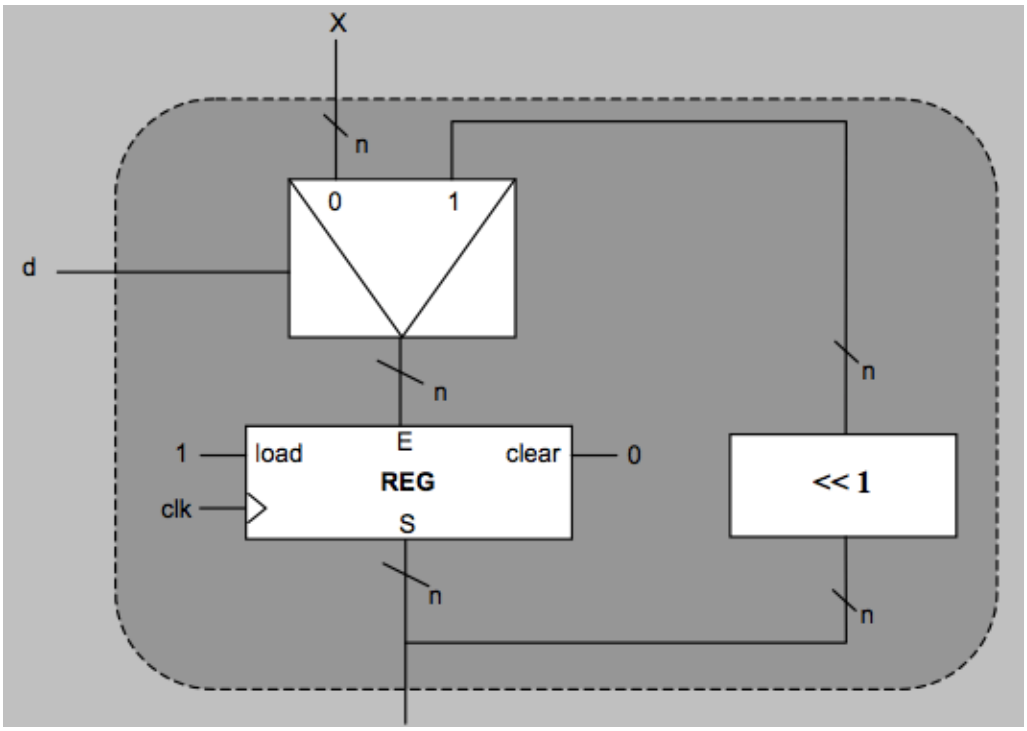
Aquest funcionament es pot expressar mitjançant aquesta taula de veritat:

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

Quan modifiquem el valor d'un registre i fem que es carregui amb el valor que hi ha a l'entrada *E*, diem que fem una escriptura.
Quan analitzem el contingut d'un registre a partir de la sortida *S*, diem que fem una lectura.

A partir d'un registre i blocs combinacionals, podem dissenyar circuits amb una funcionalitat determinada.

Per exemple, el circuit de la figura següent permet que el registre es pugui carregar amb el valor de l'entrada X o que pugui desplaçar el seu contingut 1 bit a l'esquerra, depenent del senyal *d*.



En els circuits seqüencials, assumirem sempre que hi ha un únic senyal de rellotge (*clk*). En les figures, però, a vegades no es connecten totes les entrades de rellotge amb una mateixa línia, per tal de clarificar el dibuix.
En general, si en un circuit hi ha més d'un punt identificat per un mateix nom de senyal s'entén que els punts estan connectats, encara que no estiguin units per una línia.

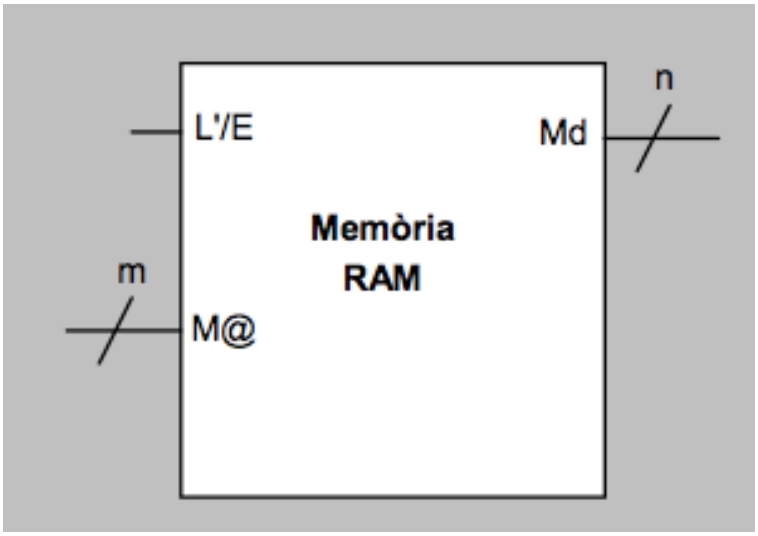
LA MEMÒRIA RAM

La memòria RAM és un bloc seqüencial que permet guardar el valor d'un cert nombre de mots (2^m) d'un cert nombre de bits (n).

La denominació *RAM* prové de l'anglès *random access memory* (memòria d'accés aleatori). Va rebre aquest nom perquè el temps que es tarda a fer una lectura o una

escriptura no depèn del mot al qual s'estigui accedint (a diferència del que passava en altres dispositius de memòria que es feien servir en els primers computadors).

La figura següent mostra la representació d'una memòria RAM amb un sol port de lectura/escriptura.



Es pot veure que disposa dels senyals següents:

- Una entrada d'adreces, M@. Si la memòria té capacitat per a 2*m* mots, l'entrada d'adreces tindrà *m* bits.
- Una entrada/sortida de dades, *Md*, de *n* bits (si els mots que guarda la memòria són de *n* bits).
- Una entrada de control, *L'/E*, que indica en tot moment si s'ha de fer una lectura o una escriptura.

El funcionament d'una memòria RAM és el següent:

- Si *L'/E* = 0, llavors es fa una lectura: pel bus *Md* surt el valor del mot que està guardat a l'adreça indicada per M@.
- Si *L'/E* = 1, llavors es fa una escriptura: el mot indicat per M@ pren el valor que hi ha a *Md* (un cert interval de temps després que *L'/E* s'hagi posat a 1).

La taula de veritat següent resumeix el funcionament de la memòria RAM.

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

La capacitat d'una memòria RAM se sol mesurar en bytes (mots de 8 bits). Com hem dit, sol contenir diversos milions de mots, i per això per a indicar la capacitat que té se solen fer servir les lletres *k*, *M* i *G*, que tenen els significats següents:

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. SIMULACIÓ DE SISTEMES DIGITALS

2. 1. INTRODUCCIÓ AL DISSENY DIGITAL: LLENGUATGES DESCRIPTORS DE MAQUINARI

Els dissenyadors de maquinari fan servir eines programari per a dissenyar circuits digitals. Els dissenys normalment comencen amb especificacions descrites amb llenguatges de descripció de maquinari. En aquest curs estudiarem un dels llenguatges més populars: **VHDL**. A partir de descripcions en VHDL, es poden construir circuits de gran complexitat fent servir eines que transformen la descripció del circuit en un conjunt de transistors que finalment formaran un circuit integrat, també anomenat *xip*.

VHDL és un resultat del programa VHSIC, suportat pel Departament de Defensa dels EUA durant els anys setanta i vuitanta. Inicialment el llenguatge va ser desenvolupat amb l'objectiu de servir com a eina d'intercanvi de disseny entre diferents dissenyadors. Les diferents descripcions havien de poder ser enteses sense equívocs per les dues parts. Més tard va ser utilitzat com a eina de modelització de dissenys, totalment adequat per a ser una descripció d'entrada a simuladors. L'any 1987 VHDL passa a ser l'estàndard IEEE 1076. L'any 1988 MilStd454 exigeix que tots els ASIC relacionats amb el Departament de Defensa estiguin descrits en VHDL. L'any 1993 el llenguatge va ser revisat i es va formar l'estàndard IEEE 1164. Finalment, l'any 1996, el llenguatge VHDL va ser definit com un estàndard de síntesi de circuits (IEEE 1076.3).

Avantatges principals de fer servir VHDL:

- Disseny independent de dispositiu. Permet incloure dissenys (IP, *intelectual properties*) externs. Facilitat de crear biblioteques. No cal coneixement del dispositiu o tecnologia.
- Portabilitat. Permet la transferència de dissenys a diferents entorns CAD, simulació i/o síntesi.
- Potència i flexibilitat. Atesa la seva capacitat de descripció funcional o comportamental, permet una eficàcia elevada en entorns de síntesi automàtica.

- Facilitat de migració. Els dissenys fets amb tecnologies PLD o FPGA poden ser migrats a ASICS. La descripció d'alt nivell de VHDL servirà com a element de verificació de la implementació final. També és interessant com a vehicle de generació de vectors de test en alt nivell.
- Capacitats de referenciació. Permet comparar fàcilment diferents implementacions fetes amb diferents descripcions i/o sintetitzadors.
- Ràpid Time-to-Market. Combinat amb eines de síntesi automàtica permet portar un disseny al mercat en temps rècord.

2.2 DESCRIPCIÓ DE SISTEMES DIGITALS AMB VHDL

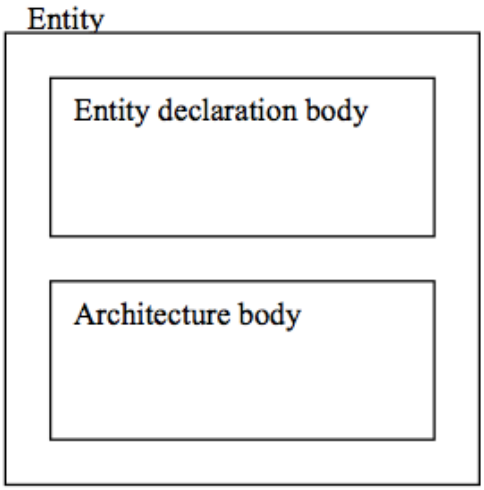
Comencem amb un exemple elemental

A continuació mostrem la descripció VHDL d'un comparador de dues paraules de 8 bits:

```
-- comp8 és un comparador de dues paraules de 8 bits

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

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

2.2.1 ESTRUCTURA I ORGANITZACIÓ DEL LLENGUATGE

Ja sigui un sistema complet o una part (disseny jeràrquic), un element (*entity*) sempre està descrit a partir de dos blocs: una **declaració d'interfície** (*entity declaration body*), en què es declaren les variables d'entrada/sortida i el seu tipus, i una **descripció del seu funcionament** (*architecture body*), que pot contenir qualsevol combinació de descripcions comportamental (*behavioral*) i estructural.



Concepte d'***entity declaration body*** i ***architecture body***

En l'exemple anterior les paraules de significat especial per al llenguatge (i, per tant, reservades) estan ressaltades i subratllades, i els identificadors definits pel dissenyador es mostren en cursiva (i així es fa al llarg d'aquests apunts). La descripció VHDL, en l'exemple, comença amb un comentari indicat pel símbol --. Després trobem, en les quatre següents línies de descripció, l'entity declaration, en què es declaren les variables d'interfície i el seu tipus.

ENTITY DECLARATION

L'**entity declaration** comença sempre per:

entity *nom_element* **is**

I acaba per:

end *nom_element;*

Cada senyal d'entrada/sortida de l'entitat està referit com un **port**. A l'**entity declaration** cal declarar aquestes variables i el seu mode (*mode*) i tipus (*types*). Els possibles modes són els següents:

- **in**: aquests senyals entren a l'entitat. Corresponen a aquest tipus els senyals rellotge, senyals d'entrada de control i els senyals de dades unidireccionals.
- **out**: correspon a senyals que surten de l'entitat, el seu controlador forma part de l'entitat i no es realimenten dins d'aquesta (són sortides netes de l'entitat; és a dir, aquests senyals no es fan servir a dins de l'entitat).
- **buffer**: correspon a un senyal de sortida però que és utilitzat també per l'entitat mateix. Quan un senyal de sortida es fa servir internament, la declaració ha de ser **buffer** i no **out**.
- **inout**: per a senyals, normalment de dades, bidireccionals.

Les variables són identificades amb una combinació de caràcters alfabètics, numèrics i guions inferiors (*underscores*) amb les restriccions següents:

- El primer caràcter ha de ser una lletra.
- L'últim caràcter no pot ser un guió inferior.
- No hi pot haver dos guions inferiors seguits.

Les variables d'entrada/sortida han de ser també declarades pel que fa al tipus (*type*) de variables. Aquests tipus són els següents:

- **bit**: variables binàries que poden prendre els valors 0 i 1.
- **bit_vector**: correspon a un vector binari; en l'exemple –*a*, *b* **in** bit_vector(7 **down** 0)–, indica que les variables d'entrada (in) *a* i *b* són vectors de 8 bits en què la coordenada 7 correspon al MSB. Els bits individuals (components del vector) són assenyalats com *a*(0), *a*(1), ... *a*(7) i *b*(7) seran els MSB.
- **std_logic**: correspon a una extensió de **bit,** definida en la versió 1164 de 1993 del VHDL; aquestes variables binàries poden prendre els valors següents:

'U'	no inicialitzat
'X'	desconegut
'0'	0
'1'	1
'Z'	alta impedància
'W'	desconegut feble (*weak*)
'L'	0 feble

'H' 1 feble

'-' no importa (*don't care*)

Per a utilitzar aquest tipus cal introduir les dues línies següents en la descripció:

library ieee;
use ieee.std_logic_1164.all;

- **std_logic_vector**: correspon a un vector de variables std_logic.
- **boolean**: variables booleanes que prenen els valors "true" i "false". S'utilitzen,
  normalment, com a variables de retorn de funcions.
- **integer**: correspon a la declaració de variables organitzades com a nombres enters decimals multidígit; l'amplada d'aquestes variables ha de ser declarada. Exemple:…… integer **range** 0 **to** 1023.

  També es poden utilitzar numeracions en diferents bases i en coma flotant, com també organitzacions matricials. Finalment el domini d'una variable discreta també pot ser definida per l'usuari, per exemple:
  type *format_meu* **is** (alt, baix, mitjà); defineix un nou tipus (*type*) discret determinat per *format_meu*.

De vegades, en l'**entity declaration** també es defineixen algunes variables que no són ports (és a dir entrades/sortides del bloc), sinó que s'utilitzen per a parametritzar la descripció dels ports. Això es fa mitjançant la indicació **generic**.

Un exemple pot 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 bloc **architecture body** es defineix la funció de l'element declarat a **entity declaration**. Les variables dels ports (definits ja a **entity declaration**) no han de ser declarades un altre cop aquesta secció; per tant, les variables dels ports es poden utilitzar directament en l'**architecture body**. Normalment, per a definir la funció d'un element cal introduir variables noves que hauran de ser declarades en les primeres línies de la descripció de l'**architecture body**.
Aquests modes nous són **constant, signal, variable**, i n'haurem de declarar el tipus.

Exemple:

```
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';
```

(fixeu-vos que per a assignar un valor a una constant o a un paràmetre es fa servir el símbol :=).

Podem entendre el bloc **entity declaration body** com una descripció de l'element en forma de capsa negra, entrades, sortides i un nom descriptor. El bloc **architecture body** ens fa veure què hi ha a dins d'aquesta capsa, i això tant pot

ser com es comporta (*behavioral*) o com està fet (estructural) o una combinació de totes dues coses.

En el nostre exemple:

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

La paraula *dataflow* és una elecció nostra per a definir de quina manera estem definint l'element *comp8.* Qualsevol identificador seria vàlid. Concretament la descripció donada com a exemple al principi del component *comp8* és de tipus comportamental. Indiquem què fa, o com es comporta, però no com està fet. Hem elegit l'identificador *dataflow* per a aquesta descripció perquè especifiquem com es comporta mitjançant el flux de les dades. En aquesta descripció l'operador <= indica assignació.
A continuació mostrem una altra descripció comportamental possible, ara en forma d'un procediment seqüencial (aquesta seria la manera com faríem una versió "programada" del component, possiblement a partir d'un µcontrolador; concretament, en VHDL, si no s'introdueixen retards específics, l'execució de la seqüència és instantània).

```
-- 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;
```

Fixem-nos que tota descripció via procediment comença així:

nom_proces: **process** (llista)

I acaba amb el següent:

end process *nom_proces;*

Les variables mostrades a *llista* s'anomenen *variables sensibles*. El procediment s'executa la primera vegada i després només quan hi ha algun canvi en alguna d'aquestes variables). L'engegada del procediment és "insensible" a variacions de variables no contingudes a *llista.*

A continuació veurem una tercera manera comportamental, ara en forma de procés booleà:

```
-- 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;
```

Operadors booleans

Els operadors booleans a VHDL són **AND**, **OR**, **NAND**, **NOR** i **XOR**.

Finalment presentem una descripció estructural en un entorn d'una certa llibreria *work.gatespkg* que tingués les portes (components) *xnor2* i *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;
```

Assignacions i temporització

L'assignació **<=** que hem comentat abans, molt utilitzada en descripcions comportamentals, pot tenir assignat un retard.

igual <= ''1'' **after** 8 ns;

indica l'assignació del valor 1 a la variable *igual* amb un retard de 8 ns, que d'alguna manera ens podria modelar el retard o temps de propagació d'un component físic.

Comparació de diferents descripcions

Un entorn com MAX Plus II sintetitza de manera automàtica una implementació sobre PLD o FPGA a partir d'una descripció VHDL. Si bé les descripcions poden

orientar diferents implementacions, el sintetitzador de l'entorn és molt eficient i la variació de recursos utilitzats davant de diferents implementacions és molt minsa.

Indicacions concurrents

Hem vist que les sentències d'un procediment es fan seqüencialment. En disseny electrònic moltes vegades volem emprar accions concurrents en el temps. Les descripcions estructurals, tota llista d'equacions lògiques o aritmètiques, una llista de procediments, estructures de decisió com **when-else** o **case-then,** totes són executades concurrentment sense que l'ordre en què estan definides tingui cap efecte.

Operadors aritmètics

VHDL fa servir els operadors aritmètics següents: *****, ******, **+**, **–**, **/**, **abs**, **mod**, **rem**.

Operadors relacionals

VHDL fa servir els operadors relacionals següents: **=**, **<**, **>**, **/=**, **<=**, **>=**.

Com a exemple:

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

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

Fixeu-vos en el significat diferent dels símbols <= o >= depenent de si es tracta d'una condició o d'una assignació.

Elements de control de seqüència

Per tal d'implementar bucles en procediments, VHDL permet els elements de control de seqüència següents:

- 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;
```

Elements de decisió a procediments

Aquests són **if-the-else** i **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;
```

Llista de paraules reservades

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 DISSENY VHDL A LA SÍNTESI DE SISTEMES DIGITALS

EXEMPLES BÀSICS AMB 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;
```

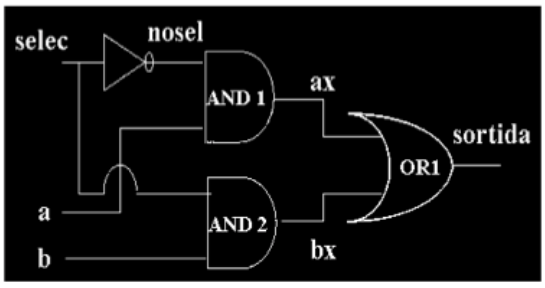
OR

```
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

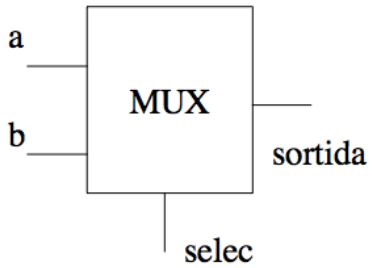
```
entity porta_and is
  port(x,y : in bit; z : out bit);
end porta_and;
architecture rtl of porta_and is
begin
  z <= x and y;
end rtl;
```

- ARQUITECTURA: exemple: multiplexor: estil estructural



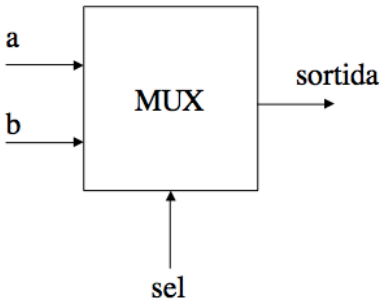

```
architecture estructural of mux is
component inv port(x: in bit; y: out bit);
end component;
component porta_or port(x,y : in bit; z : out bit);
end component;
component porta_and port(x,y : in bit; z : out bit);
end component;
signal ax, bx, nosel: bit;

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

DESCRIPCIÓ DE LÒGICA SÍNCRONA

En els circuits digitals síncrons, les actuacions de molts blocs estan condicionades pel senyal rellotge (*clock*), que pot ser actiu per nivell o per flanc.

Vegem un exemple de descripció d'un *flip-flop* tipus *D* actuat per flanc de pujada (*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;
```

Després de declarar la biblioteca IEEE i l'estàndard 1164, la descripció declara els ports dins de l'entity **declaration body**: es declaren dues entrades, *d* i *clk* i una sortida, *q*.

La descripció comportamental definida en l'architecture **body** es fa mitjançant un procediment (*process*) en què es declara *clk* com a variable sensible.

Mitjançant un **IF** (*condition*) es condiciona l'assignació instantània de la variable *d* a *q* (estat). La condició és doble: d'una banda, *clk'event*, que és una funció actuant sobre *clk* que retorna "true" si hi ha algun canvi de valor a *clk*; i d'una altra, la condició *clk* = 1. L'**AND** de les dues condicions equival a la condició de flanc de pujada.

En el cas d'un *flip-flop* actiu per nivell alt, la condició de l'**IF** seria simplement (*clk* = 1). Observeu la importància que *clk* estigui declarada a la llista de variables sensibles del procediment.

Quan s'utilitza *std_logic_1164*, es poden fer servir les funcions *rising_edge(variable)* i *falling_edge(variable)*.

La funció *rising_edge(clk)* és exactament equivalent a *clk 'event* **and** *clk* = 1' i la funció *falling_edge(clk)* a *clk'event* **and** *clk* = 0'.

INICIALITZACIÓ ASÍNCRONA

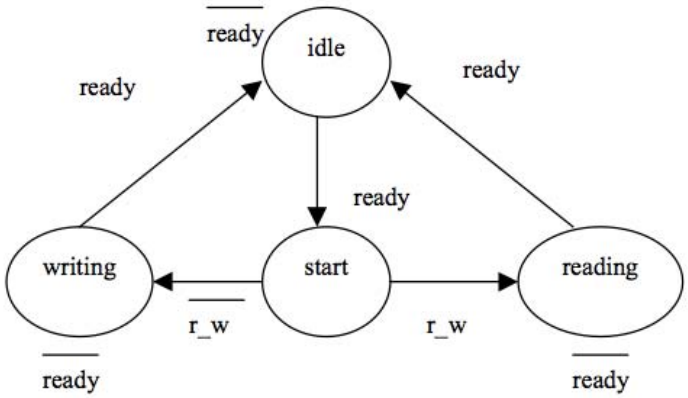
Molts dispositius *flip-flop* disposen d'una entrada d'inicialització asíncrona. A continuació veiem una descripció d'un *flip-flop* tipus D actiu per flanc de pujada amb capacitat d'inicialització asíncrona. Fixeu-vos com s'aconsegueix la doble sensibilitat de *clk* i *reset* i com l'actuació de *reset* és independent de *clk* i prioritària fent servir l'ordre d'execució en un element **if-elseif** d'un procediment:

```
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;
```

DESCRIPCIÓ DE MÀQUINES D'ESTATS

Molts blocs lògics constituents d'un sistema es defineixen a partir d'una màquina d'estats finits (FSM). VHDL permet diverses tècniques per a descriure-les. Moltes són molt directes en la seva descripció.

Considerem el cas d'una FSM que actuaria com a element parcial d'un control d'un bloc de memòria. La FSM rep dues variables d'entrada: *ready*, que indica quan la memòria està preparada, i *read_write* (*r_w*), que indica si es pretén llegir o escriure. La FSM genera dues variables: *oe* i *we* que s'aplicaran als senyals *output enable* i *write enable* del bloc de memòria. El diagrama de transició d'estats i la taula de variables de sortida segons l'estat estan indicades a continuació:



Taula de sortides

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

Una descripció d'aquesta màquina és la següent:

```
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;
```

DISSENY JERÀRQUIC A VHDL

El disseny de circuits i sistemes complexos acostuma a ser fragmentat en parts (components, subcircuits) manejables pel dissenyador, cosa que dóna lloc al concepte de disseny jeràrquic. L'entorn **Max Plus II** disposa d'una capacitat flexible de jerarquia de parts, dins d'un mateix disseny, que poden ser definides amb diferents tècniques.

Alguns dels avantatges més importants del disseny jeràrquic són els següents:

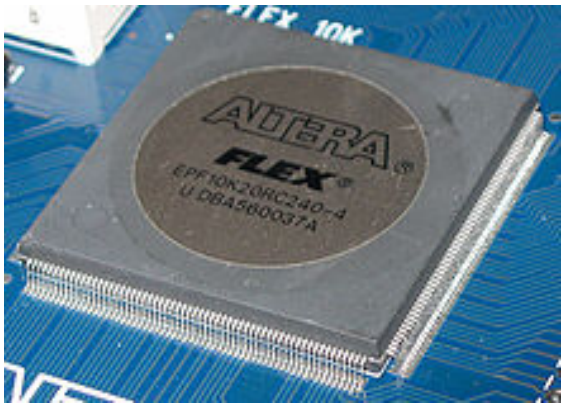
- Una jerarquització d'un disseny comporta una bona estructuració i usualment una millor definició.
- Cada part o component pot ser verificada de manera separada, i simplificar la posada al punt, verificació, simulació.
- En un entorn, les parts, si estan adequadament definides, poden formar part d'una llibreria, poden ser reutilitzades per diferents dissenyadors.
- Permet un disseny paral·lel, és a dir en col·laboració en equip amb altres enginyers.

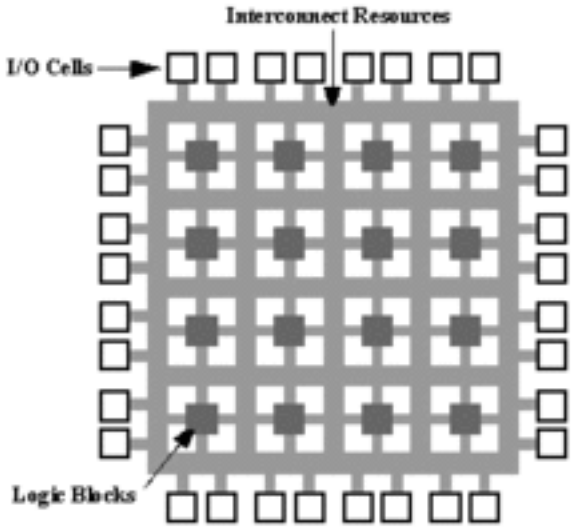
3. IMPLEMENTACIÓ DE SISTEMES DIGITALS SOBRE DISPOSITIUS PROGRAMABLES

3.1 INTRODUCCIÓ ALS DISPOSITIUS PROGRAMABLES: FPGA

Una **FPGA** (*field programmable gate array*) és un dispositiu semiconductor que conté blocs de lògica la interconnexió i funcionalitat dels quals pot ser configurada *in situ* amb un llenguatge de programació especialitzat. La lògica programable pot reproduir des de funcions tan senzilles com les que fa una porta lògica fins a sistemes complexos en un xip.

Exemples físics de FPGA:





Arquitectura d'una FPGA

Les FPGA s'utilitzen en aplicacions similars als ASIC encara que són més lentes, consumeixen més potència i no poden comprendre sistemes tan complexos com els altres. Encara així, les FPGA tenen els avantatges de ser reprogramables (cosa que afegeix una gran flexibilitat al flux de disseny), els seus costos de desenvolupament i adquisició són molt més baixos per a petites quantitats de dispositius i el temps de desenvolupament també és menor.

Certs fabricants tenen FPGA que només es poden programar un cop, i això fa que els seus avantatges i inconvenients siguin a mig camí entre els ASIC i les FPGA reprogramables.

Tradicionalment, els enginyers han utilitzat les FPGA amb eines de programació fetes per experts. No obstant això, com que les FPGA s'han tornat més ràpides i més rendibles, els enginyers i investigadors amb poca experiència o sense en disseny de maquinari digital volen aprofitar les FPGA per a crear solucions personalitzades. Per a abastar aquest interès creixent, els proveïdors estan creant eines de més alt nivell que fan més fàcil programar FPGA i brindar els beneficis de la tecnologia FPGA a aplicacions noves.

Les FPGA són el resultat de la convergència de dues tecnologies diferents, els dispositius lògics programables (**PLD**, *programmable logic devices*) i els circuits

integrats d'aplicació específica (**ASIC**, *application-specific integrated circuit*). La història dels PLD va començar amb els primers dispositius PROM (*programmable read-only memory*), i se'ls va afegir versatilitat amb els PAL (*programmable array logic*), que van permetre més entrades i la inclusió de registres. Aquests dispositius han continuat creixent en grandària i potència. Mentrestant, els ASIC sempre han estat dispositius potents, però el seu ús ha requerit tradicionalment una inversió considerable tant de temps com de diners. Alguns intents de reduir aquesta càrrega han provingut de la modularització dels elements dels circuits, com els ASIC basats en cel, i de l'estandardització de les màscares, tal com Ferranti va ser pioner amb la ULA (*uncommitted logic array*). El pas final era combinar les dues estratègies amb un mecanisme d'interconnexió que pogués programar utilitzant fusibles, antifusibles o cel RAM, com els innovadors dispositius Xilinx de mitjans dels vuitanta. Els circuits resultants són semblants en capacitat i aplicacions als PLD més grans, encara que hi ha diferències puntuals que delaten avantpassats diferents. A més de computació reconfigurable, les FPGA s'utilitzen en controladors, codificadors/decodificadors i en el prototipatge de circuits VLSI i microprocessadors a mida.

El primer fabricant d'aquests dispositius va ser Xilinx i els dispositius de Xilinx es mantenen com un dels més populars en companyies i grups de recerca. Altres venedors en aquest mercat són Atmel, Altera, AMD i Motorola.

CARACTERÍSTIQUES

Una jerarquia d'interconnexions programables permet als blocs lògics d'un FPGA ser interconnectats segons la necessitat del dissenyador del sistema, molt semblant a un *breadboard* (placa d'ús genèric reutilitzable o semipermanent) programable. Aquests blocs lògics i interconnexions poden ser programats després del procés de manufactura per l'usuari/dissenyador, així que el FPGA pot exercir qualsevol funció lògica necessària.

Una tendència recent ha estat combinar els blocs lògics i interconnexions dels FPGA amb microprocessadors i perifèrics relacionats per a formar un sistema programable en un xip. Exemple d'aquestes tecnologies híbrides poden ser trobats

en els dispositius Virtex-II PRO i Virtex-4 de Xilinx, els quals inclouen un processador PowerPC embegut juntament amb la lògica del FPGA, o més d'un. El FPSLIC d'Atmel és un altre dispositiu similar que utilitza un processador AVR en combinació amb l'arquitectura lògica programable d'Atmel. Una altra alternativa és fer ús de nuclis de processadors implementats fent ús de la lògica del FPGA. Aquests nuclis inclouen els processadors MicroBlaze i PicoBlaze de Xlinx, Nens i Nens II d'Altera, i els processadors de codi obert LatticeMicro32 i LatticeMicro8.

Molts FPGA moderns suporten la reconfiguració parcial del sistema, i permeten que una part del disseny sigui reprogramada, mentre les altres parts segueixen funcionant. Aquest és el principi de la idea de la computació reconfigurable, o els sistemes reconfigurables.

Arquitectura interna d'una FPGA

Les FPGA van ser inventades l'any 1984 per Ross Freeman, cofundador de Xilinx, i van sorgir com una evolució dels PLA i dels CPLD.

Un **CPLD** (de l'acrònim anglès *complex programmable logic device*) és un dispositiu electrònic.

Els CPLD estenen el concepte d'un PLD (de l'acrònim anglès *programmable logic device*) a un nivell d'integració més alt, perquè permeten implementar sistemes més eficaços, ja que utilitzen menys espai, milloren la fiabilitat del disseny, i redueixen costos. Un CPLD es forma amb múltiples blocs lògics, cada un semblant a un PLD. Els blocs lògics es comuniquen entre si utilitzant una matriu programable d'interconnexions, la qual cosa fa més eficient l'ús del silici, i això condueix a una eficiència millor a un cost més baix.

Tant els CPLD com les FPGA contenen un gran nombre d'elements lògics programables. Si mesurem la densitat dels elements lògics programables en portes lògiques equivalents es podria dir que en un CPLD trobaríem entorn de desenes de

milers de portes lògiques equivalents, i en una FPGA, entorn de cents de milers fins a milions.

A part de les diferències en densitat entre tots dos tipus de dispositius, la diferència fonamental entre les FPGA i les CPLD és la seva arquitectura. L'arquitectura dels CPLD és més rígida i consisteix en una suma o més de productes programables els resultats de la qual van a parar a un nombre reduït de biestables síncrons (també denominats *flip-flop*s). L'arquitectura de les FPGA, per altra banda, es basa en un gran nombre de petits blocs utilitzats per a reproduir senzilles operacions lògiques que alhora tenen biestables síncrons. L'enorme llibertat disponible en la interconnexió d'aquests blocs concedeix a les FPGA una gran flexibilitat.

Una altra diferència important entre FPGA i CPLD és que en la majoria de les FPGA es poden trobar funcions d'alt nivell (com sumadors i multiplicadors) intrínseques en la matriu d'interconnexions, com també blocs de memòria.

Les FPGA s'utilitzen en aplicacions similars als ASIC però tenen els inconvenients i avantatges següents respecte d'aquests:

Inconvenients

- Són més lents.
- Consumeixen més potència.
- No poden realitzar sistemes tan complexos perquè contenen portes lògiques amb un comportament reprogramable que permet la implementació de dispositius lògics.

Avantatges

- Són reprogramables.
- Tenen costos de desenvolupament i d'adquisició molt més baixos.
- El temps de desenvolupament és molt menor.

Històricament, les FPGA sorgeixen com una evolució dels conceptes desenvolupats pels CPLD.

Una altra diferència és que les FPGA utilitzen tecnologies de memòria diferents:

- **Volàtils**: basades en RAM. La seva programació es perd en treure l'alimentació. Requereixen una memòria externa no volàtil per a configurar-les en arrancar (abans o durant la reinicialització).
- **No volàtils**: basades en ROM. N'hi ha de dos tipus: les reprogramables i les no reprogramables.

    1. **Reprogramables**: basades en EPROM o Flash. Aquestes es poden esborrar i tornar a reprogramar, encara que amb un límit d'uns 10.000 cicles.
    2. **No reprogramables**: basades en fusibles. Només es poden programar una vegada, cosa que les fa poc recomanables per a treballs en laboratoris.

Pel que fa a la **seguretat**, les FPGA tenen avantatges i desavantatges en comparació amb els ASIC o microprocessadors segurs. La flexibilitat FPGA fa que les modificacions malicioses que hi pugui haver durant la fabricació siguin d'un risc menor. Per moltes FPGA, el disseny carregat està exposat mentre es carrega (en general en cada encesa del dispositiu). Per a abordar aquesta qüestió, alguns FPGA suporten el xifratge *bitstream encryption*.

3 .2 DEL LLENGUATGE DE DESCRIPCIÓ A LA SÍNTESI DEL DISPOSITIU

PROGRAMACIÓ FPGA

La feina del programador és definir la funció lògica que farà cada un dels CLB, seleccionar el mode de treball de cada IOB i interconnectar-lo.

El disseny de circuits i sistemes complexos acostuma a ser fragmentat en parts (components, subcircuits) manejables pel dissenyador, cosa que dóna lloc al concepte de **disseny jeràrquic**. L'entorn **Max Plus II** disposa d'una capacitat flexible de jerarquia de parts, dins d'un mateix disseny, que poden ser definides amb tècniques diferents.

En un flux de disseny típic, un desenvolupador d'aplicacions FPGA simularà el disseny en diverses etapes durant el procés de disseny. Inicialment, la descripció RTL en VHDL o Verilog se simula amb la creació de bancs de proves per a simular el sistema i observar els resultats. Després que el motor de síntesi ha traçat el disseny a una llista d'interconnexions (o *netlist*), aquesta llista es tradueix en una descripció del nivell de la porta en què la simulació es repeteix per a confirmar la síntesi a terme sense errors. Finalment, el disseny es presenta a la FPGA en el moment en què les demores de propagació es poden afegir i executar la simulació de nou amb aquests valors de *back* anotats a la llista d'interconnexions. El dissenyador té l'ajuda d'entorns de desenvolupament especialitzats en el disseny de sistemes que s'implementen en una FPGA. Un disseny pot ser capturat ja sigui esquemàtic o fent ús d'un llenguatge de programació especial. Aquests llenguatges de programació especials són coneguts com a *HDL* o *hardware description language* (llenguatges de descripció de maquinari). Els HDL més utilitzats són els següents:

- **VHDL**: tal com hem vist en el mòdul anterior és un llenguatge definit per l'IEEE usat per enginyers per a descriure circuits digitals.
- **Verilog**: és un llenguatge de descripció de maquinari usat per a modelar sistemes electrònics.
- **ABEL**: és un llenguatge de descripció de maquinari i un conjunt d'eines de disseny per a programar dispositius lògics programables.

En un intent de reduir la complexitat i el temps de desenvolupament en fases de prototipatge ràpid, i per a validar un disseny en HDL, hi ha diverses propostes i nivells d'abstracció del disseny. Entre d'altres, **National Instruments LabVIEW FPGA** proposa un llenguatge de programació gràfica d'alt nivell.

APLICACIONS FPGA

Qualsevol circuit d'aplicació específica pot ser implementat en una FPGA, sempre que aquest disposi dels recursos necessaris. Les aplicacions en què més comunament s'utilitzen les FPGA, inclusivament els **DSP** (processament digital de senyals), són ràdio definida per programari, sistemes aeroespacials i de defensa,

prototipus d'ASC, sistemes d'imatges per a medicina, sistemes de visió per a computadors, reconeixement de veu, bioinformàtica, emulació de maquinari de computadora. Hem de saber que cada cop es fa servir més en altres àrees, sobretot en les aplicacions que requereixen un alt grau de paral·lelisme.

Les FPGA especialment troben aplicacions en qualsevol àrea o algorisme que pugui fer ús de l'alt grau de paral·lelisme ofert per la seva arquitectura. Un exemple és el trencament de codis, en particular atacs de força bruta a algorismes criptogràfics.

A més, cada cop són més usades en aplicacions convencionals d'alt rendiment en què els nuclis computacionals com FFT o convolució són implementats en una FPGA en comptes d'un processador d'ús general.

Hi ha codi font disponible (amb llicència GNU GPL) de sistemes com microprocessadors, microcontroladors, filtres, mòduls de comunicacions i memòries, entre d'altres. Aquests codis s'anomenen *cores*.

FABRICANTS DE FPGA

A principi de 2007, el mercat de les FPGA s'havia col·locat en un estat amb dos grans productors de FPGA de propòsit general i un conjunt d'altres competidors que es diferenciaven perquè oferien dispositius de capacitats úniques.

- **Xilinx** és un dels dos grans líders en la fabricació de FPGA.
- **Altera** és l'altre gran líder.
- Lattice Semiconductor va treure al mercat dispositius FPGA amb tecnologia de 90nm. Lattice és un proveïdor líder en tecnologia no volàtil, FPGA basades en tecnologia Flash, amb productes de 90nm i 130nm.
- Actel té FPGA basades en tecnologia Flash reprogramable. També ofereix FPGA que inclouen mescladors de senyals basats en Flash.
- QuickLogic disposa de productes basats en antifusibles (són programables només un cop).

- Atmel és un dels fabricants que ofereix productes reconfigurables.(Xilinx XC62xx va ser un d'aquests, però actualment ja no se'n fabriquen). Es van enfocar a proveir microcontroladors AVR amb FPGA, tot en el mateix encapsulatge.
- Achronix Semiconductor tenen FPGA molt ràpides en desenvolupament. Actualment tenen FPGA que funcionen a 1,5 GHz.
- MathStar, Inc. ofereixen FPGA que anomenen *FPOA* (*field programmable object arrays*).
- Tabula, el març del 2010, va anunciar una nova tecnologia FPGA que utilitza la lògica de temps multiplexatge i la interconnexió de més potencial d'estalvi per a aplicacions d'alta densitat.

4. SISTEMES DE PROPÒSIT ESPECÍFIC

4.1 INTRODUCCIÓ ALS SISTEMES DE PROPÒSIT ESPECÍFIC.

En aquest mòdul ens centrarem en el processador de senyals digitals, **DSP**.

El **processador de senyals digitals**, conegut en anglès com a ***DSP*** (*digital signal processor*), és un processador o microprocessador que incorpora el maquinari capaç d'executar els algoritmes per al processament digital d'un senyal d'entrada en temps real, com pot ser l'entrada del senyal d'un fitxer d'àudio, per a obtenir les operacions corresponents i extreure'n la sortida. Com que treballa amb senyals digitals necessita un convertidor dels senyals analògics a digitals (ADC) a l'entrada, i un convertidor digital analògic (DAC) a la sortida, normalment. Com tots els sistemes basats en processadors programables, necessita una memòria per a guardar les dades amb les quals treballarà i el programa que executa.

Els DSP també tenen diverses solucions via maquinari i/o programari per a les instruccions de treball, variable de gran transcendència en el moment de tractar un senyal mostrejat. Aquestes eines fan que els DSP s'imposin moltes vegades a la construcció d'un dispositiu especialitzat només per a aquest procés en particular. S'observa llavors que l'avantatge principal d'aquest sistema és que està dissenyat per a treballar amb la major quantitat de contratemps possibles i en una quantitat de temps determinada.

Si es té en compte que un DSP pot treballar amb diverses dades en paral·lel i un disseny i instruccions específiques per al processament digital, es pot veure la potencialitat que arriba a tenir per a aquest tipus d'aplicacions. Aquestes característiques constitueixen la diferència principal d'un DSP i d'altres tipus de processadors.

S'utilitzen en circuits relacionats amb la imatge, el so, les telecomunicacions i la regulació i control, com telèfons mòbils, reproductors MP3, càmeres digitals, sintonitzadors TDT, regulació de velocitat, posicionament de precisió, etc., per a les funcions de processament de senyal en temps real: reducció de soroll, filtratge en general, compressió, descompressió, detecció i correcció d'errors, etc.

L'avantatge principal que tenen és la potència que els proporciona la seva estructura, la qual els permet treballar en paral·lel, amb una memòria de dades d'accés ràpid i gran capacitat, gran poder d'execució gràcies a les unitats MAC i ALU, i tot en temps real.

S'han desenvolupat sostingudament durant els últims quaranta anys, des que la disponibilitat de computadors va fer possible l'aplicació pràctica d'algoritmes que abans només podien ser avaluats de manera manual.
Les millores tecnològiques contínues han permès substituir els circuits analògics per circuits digitals que ocupen menys volum, i que estan lliures de problemes de tolerància dels components, calibratge, i deriva tèrmica que afecten els analògics.

4.2 CARACTERÍSTIQUES DELS PROCESSADORS DIGITALS DE SENYAL (DSP)

ARQUITECTURA

Els DSP no utilitzen l'arquitectura de Von Neumann, en què les dades i els programes són a la mateixa memòria, sinó que fan ús de l'arquitectura Harvard, en què dades i programes són en memòries diferents.

Cada memòria és adreçada amb diferents busos, i fins i tot és possible que la memòria de dades tingui una amplada diferent que la de programes.

Amb la tecnologia Harvard, com que tenim memòries diferents, aconseguim accelerar l'execució de les instruccions, ja que mentre estem executant una instrucció (que fa servir la memòria de dades) podem començar a descodificar la instrucció següent (que fa servir la memòria de programa).

Normalment els DSP utilitzen una arquitectura Harvard modificada, i fan servir tres busos, un per al programa i dos de dades. Així, la CPU pot llegir una instrucció i dos operands alhora; d'aquesta manera es guanya en temps. Aquests busos poden ser com en els processadors convencionals de 16, 32 o 64 bits; o amplades de bus tan diferents com 24, 48 o 56 bits. Com en els processadors convencionals, també inclou un comptador de programa (*program counter*) i un punter de pila (*stack pointer*).

Els elements bàsics d'un DSP, a part de la memòria de dades i de programa, són convertidors A/D a les entrades i D/A a les sortides, i, dins del processador DSP, multiplicadors i acumuladors, una ALU i registres:

- **ALU**: unitat lògica algorítmica que s'encarrega d'executar els càlculs algorísmics.
- **DMA**: memòria d'accés directe que treballa a una freqüència tan ràpida com el processador.
- **MAC**: multiplicador acumulador encarregat de fer el producte i d'acumular el resultat.
- **Busos de dades**: en paral·lel per a donar més potència.
- **Registres de desplaçament**
- **Estructura Harvard**: basada en l'emmagatzemament per separat usant més d'un bus.
- **Pipeline**: execució en cadena.
- **Coma fixa/coma flotant**

El 1979, Bell va introduir el primer xip DSP, era el Mac 4 Microprocessor, capaç de processar senyals digitals.
Un altre avanç en PDS va ser l'Altamira DX-1, el qual treballava utilitzant *pipelines* (cadenes d'execucions amb retard entre si) i que permetia una gran potència d'execució.
Però el primer PDS fabricat per Texas Instrument (TI), presentat el 1983, va ser un dels èxits més importants. Actualment Texas Instrument és un dels fabricants de PDS més importants.

PROGRAMACIÓ

Per a programar un DSP, s'utilitza un programa que es guarda com un codi màquina a dins del DSP. Si un programador escrivís un programa de DSP utilitzant codi màquina li seria molt difícil. Per això es va desenvolupar un llenguatge assemblador per a programar DSP. Les seves instruccions mnemòniques són simbòliques i en correspondència una a una amb les instruccions de màquina. S'utilitzen un assemblador, un enllaçador i un compilador que tradueix els codis font. Tot això serveix per a traduir el programa escrit en llenguatge assemblador en els codis de màquina del DSP. Són els casos de **LabVIEW** i **Matlab**.

APLICACIONS DSP

Les aplicacions més habituals en què s'utilitzen els DSP són el processat d'àudio i vídeo, i qualsevol altra aplicació que requereix processament en temps real. Amb aquestes aplicacions es pot eliminar l'eco en les línies de comunicacions, aconseguir fer més clares les imatges d'òrgans interns en els equips de diagnòstic mèdic, xifrar converses en telèfons portàtils per a mantenir la privacitat, analitzar dades sísmiques per a trobar reserves noves de petroli, fer possible les comunicacions sense fil LAN, el reconeixement de veu, els reproductors digitals d'àudio, els mòdems sense fil, les càmeres digitals, i una llarga llista d'elements que poden estar relacionats amb el processament de senyals.

Algunes altres aplicacions dels DSP són les següents:

• **Verificació de la qualitat del subministrament elèctric**: mesura de valor efectiu, potència, factor de potència, contingut harmònic i *flicker*.
• **Radars**: mesura de la distància i de la velocitat dels contactes. Comprensió del pols, la qual cosa permet incrementar la longitud dels polsos per a augmentar l'abast, mantenint la resolució en distància.
• **Sonars**: formació de feixos, per a orientar electrònicament la reparació de transductors; en mode actiu, mesura de la distància, la demarcació i la velocitat dels contactes; en mode passiu, classificació dels contactes segons el soroll que produeixen.
• **Medicina**: reducció de soroll i diagnòstic automàtic d'electrocardiogrames i electroencefalogrames; formació d'imatges en tomografia axial computeritzada (escàner), ressonància magnètica nuclear i ecografia (ultrasò).
• **Anàlisi de vibracions en màquines**: per a detectar prematurament el desgast de rodaments o engranatges, comparant l'anàlisi espectral de les vibracions amb un espectre de referència obtingut quan la màquina no té defectes.
• **Oceanografia**: alerta prematura de sismes submarins o tsunamis quan es propaguen en l'oceà obert, segons les característiques d'aquestes ones, que es diferencien de les onades i de les marees; anàlisi harmònica i predicció de marees; mesura de l'energia de les onades amb l'objectiu de dimensionar molls i altres estructures submergides.
• **Astronomia**: detecció de planetes en estrelles llunyanes, segons el moviment oscil·latori que indueixen en les estrelles al voltant de les quals orbiten.
• **Radioastronomia**: cerca de patrons en els senyals rebuts pels radiotelescopis, per a detectar intel·ligència extraterrestre (SETI).
• **Imatges**: millora de la llum, contrast, color i nitidesa, restauració d'imatges borroses pel moviment de la càmera o de l'element fotografiat. Compressió de la informació.

TRANSFORMADA

Un dels beneficis principals del DSP és que les transformacions de senyals són més fàcils de fer. La **transformada de Fourier discreta** (TFD), és una de les més

importants. Aquesta transformada ens permet convertir un senyal de domini de temps en una de domini de freqüència. La TFD permet una anàlisi més senzilla i eficaç de la freqüència, sobretot en aplicacions d'eliminació de soroll i en altres tipus de filtratge (filtres passa baixos, passa alts, passa banda...).

Una altra de les transformades importants és la transformada del cosinus discreta, és similar a l'anterior pel que fa als càlculs necessaris per a poder obtenir-la, però converteix els senyals en components del cosinus trigonomètric. Aquesta transformada és una de les bases de l'algoritme del compressor d'imatges JPEG.

FONTS I AUTORIES

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

Armstrong, J. R. (1988). *Chip-level modeling with VHDL.* Englewoods Cliffs, NJ: Prentice Hall.

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

Gené Pujols, E. Material propi de l'autor.

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

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

http://logic.ly/

Viquèdia: es.wikipedia.org

weble.upc.es/dcise/Practiques

Tutorial en línia VHDL en anglès : **http://www.vhdl-online.de/tutorial/**

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

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