

FUNDAMENTOS DE COMPUTADORES

Juan Diego Barrado Daganzo
1º de Carrera

Basado en apuntes de Jose Manuel Mendíaz Cuadros (UCM)

13 de septiembre de 2021

REPRESENTACIÓN DIGITAL DE LA INFORMACIÓN

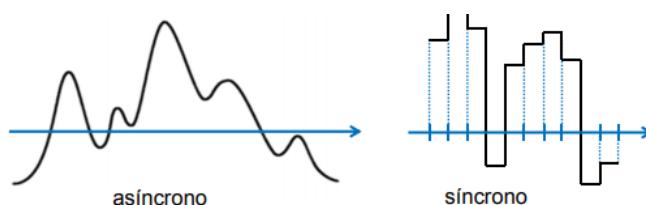
SISTEMAS

Caja "negra" que a lo largo del tiempo, recibe información por sus entradas, procesa dicha información según una cierta función y genera información por sus salidas, $z(t)$.

- **Analógicos:** pueden tomar valores en un espectro continuo de ellos
- **Digitales:** pueden tomar valores restringidos a un conjunto discreto
- **Combinacionales:** la salida en cada instante depende exclusivamente de la combinación de entrada
- **Secuenciales:** la salida en cada instante depende del valor de la entrada en ese instante y de todos los valores que la entrada ha tomado anteriormente. En el secuencial ha de introducirse el sistema de memoria o tiempo



- **Síncronos:** las entradas y salidas solo pueden cambiar en un conjunto discreto de instantes definidos por una señal de reloj
- **Asíncronos:** las entradas y salidas pueden cambiar en cualquier momento



La **especificación** de un sistema es la descripción del comportamiento y funcionalidades que ese sistema posee (¿qué hace?).

La **implementación** de un sistema es la descripción en base al conjunto de elementos que forman ese sistema (¿cómo es?).

SISTEMAS DE NUMERACIÓN

Es un mecanismo que permite dar representación gráfica a cada número. Se define por:

- **Dígitos:** conjunto discreto de símbolos, lo que se conoce como la BASE
- **Notación:** conjunto discreto de reglas de generación que permite representar números mayores a los dígitos
- **Aritmética:** conjunto de reglas de manipulación de símbolos que permite realizar operaciones entre números coherentemente.

Además del sistema decimal, es posible utilizar el sistema binario, cuya base es 2, para expresar un número. El **sistema binario**, al igual que el decimal, es un sistema numérico POSICIONAL, es decir, en el que la posición del dígito influye en el valor del número en su totalidad. El **sistema hexadecimal**, a su vez también es un sistema numérico posicional, pero cuya base numérica es 16. Se utiliza este sistema sobre todo por su correspondencia con 4 bits, lo que le aporta la capacidad de poder escribir grandes números binarios de forma abreviada.

Rango de representación

Con un número de n dígitos decimales se pueden representar 10^n números distintos pertenecientes al rango: $(0, 10^n - 1)$

$$n = 3 \Rightarrow 10^3 \text{ números distintos entre } (0, 999)$$

Con un número de n dígitos binarios se pueden representar 2^n números distintos pertenecientes al rango: $(0, 2^n - 1)$

$$n = 3 \Rightarrow 2^3 \text{ números distintos entre } (0, 7)$$

Con un número de n dígitos hexadecimales se pueden representar 16^n números distintos pertenecientes al rango: $(0, 16^n - 1)$

$$n = 3 \Rightarrow 16^3 \text{ números distintos entre } (0, 4095)$$

Conversión entre sistemas

- $10^n \rightarrow 2^n$

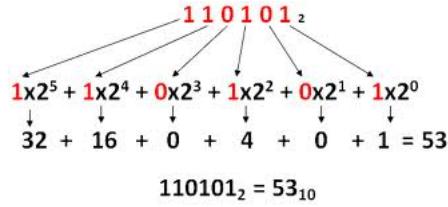
Dividimos el número a convertir entre 2, 16, 8, ..., anotamos el resto de la división y dividimos el cociente resultante hasta que este sea 1. Despues, anotamos en orden inverso los sucesivos restos de las divisiones hasta INCLUIR el ÚLTIMO 1 DEL COCIENTE FINAL.

$$\begin{array}{r}
 105 \mid 2 \\
 05 \quad 52 \quad | 2 \\
 \textcircled{1} \quad 0 \quad 12 \quad 26 \quad | 2 \\
 \textcircled{0} \quad \textcircled{0} \quad 06 \quad 13 \quad | 2 \\
 \textcircled{1} \quad \textcircled{1} \quad \textcircled{1} \quad 6 \quad | 2 \\
 \textcircled{0} \quad \textcircled{0} \quad \textcircled{0} \quad 3 \quad | 2 \\
 \textcircled{1} \quad \textcircled{1} \quad \textcircled{1} \quad 1 \quad | 2 \\
 \textcircled{1} \quad \textcircled{1} \quad \textcircled{1} \quad 0
 \end{array}$$

$105_{10} = 1101001_2$

- $2^n, 16^n, 8^n, \dots \rightarrow 10^n$

Elevamos cada dígito a la potencia de $2^n, 8^n, 16^n, \dots$ ¹ correspondiente siendo x^0 el dígito más a la derecha y x^{n-1} el de más a la izquierda. El valor n es el número de dígitos que tiene el valor binario.



- $2^n \rightarrow 16^n, 8^n, \dots$

Se agrupan de derecha a izquierda los dígitos binarios de i en i , siendo $i: \text{base } R \rightarrow \text{base } S = R^i$, es decir, $8 = 2^3 \Rightarrow i = 3, 16 = 2^4, i = 4$ posteriormente se traduce el valor de ese dígito binario (que como máximo su cifra será 15 por lo explicado en el rango de representación) al dígito hexadecimal correspondiente:

Dígito Hexadecimal	Decimal Equivalente	Binario Equivalente	Dígito Hexadecimal	Equivalente Decimal	Equivalente Binario
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

- $16^n \rightarrow 2^n$

Mismo proceso, pero a la inversa.

REPRESENTACIÓN DE LA INFORMACIÓN

Un sistema digital solo procesa información codificada en binario.

Codificación: es un convenio que asocia a cada elemento una representación en binario, por lo que un mismo elemento puede tener más de una representación en función de la codificación que se emplee.

Cada código emplea un número fijo de bits que limita el rango de representación de dicha codificación, cuando un número sobrepasa dicha frontera decimos que se produce un problema de desbordamiento o *overflow*, detectándose en un resultado incorrecto de las operaciones aritméticas realizadas.

Para saber el número necesario de bits que hacen falta para representar un número, elegimos un $k : 2^k - 1 > N$, siendo N mi número en decimal, después si existe bit de signo en nuestra codificación se escoge un $n = k + 1$, siendo k y n el número de bits de nuestra codificación (en función de si tiene bits de signo o no)

¹De la utilización de dichos sistemas nace el término bit, byte y nibbles.

Binario puro

Esta codificación solo representa números naturales.

- **Notación:** k bits codifican la magnitud del natural
- **Rango de representación:** $[0, 2^k - 1]$
- **Aritmética:**
 - Extensión: para pasar a mayor numero de bits completar con 0 por la izq.
 - Suma: suma binaria normal, hay desbordamiento cuando al sumar los bits más significativos se produce acarreo
- **Proceso de codificación y decodificación**

Se siguen los pasos explicados en *Conversión entre sistemas*

MyS

Esta codificación representa números enteros.

- **Notación:** 1 bit codifica el signo (0 es positivo y 1 es negativo) y $n - 1$ bits codifican la magnitud del número.
 - Positivos: $+N_{10} \rightarrow 0(N)_2$
 - Negativos: $-N_{10} \rightarrow 1(N)_2$
- **Rango de representación**²: $[-(2^{n-1} - 1), (2^{n-1} - 1)]$
- **Aritmética:**
 - Cambio de signo: cambio del bit más significativo
 - Extensión: para pasar a mayor numero de bits completar con 0 por la izq. conservando el primer bit intacto (bit de signo).
 - Suma y Resta
- **Proceso de codificación y decodificación**

Para **codificar**, primero se codifica la magnitud de la forma explicada en *Conversión entre sistemas* y después se le agrega a la izquierda el bit de signo correspondiente.

Para **decodificar** el número, primero se decodifica el bit de signo y con los $n - 1$ bits restantes se decodifica la magnitud.

Complemento a 2

Esta codificación representa números enteros.

- **Notación:**
 - Positivos: $+N_{10} \rightarrow 0(N)_2$
 $12_{10} \rightarrow 01100_{C2}$
 - Negativos: $-N_{10} \rightarrow (2^n - N)_2 = C2(N_2)$
 $-12_{10} \rightarrow 2^5 - 12 = 20 \rightarrow 10100_{C2}$

²El 0 posee doble representación (en negativo y positivo)

- **Rango de representación**³: $[-2^{n-1}, +2^{n-1} - 1]$

- **Aritmética**⁴:

- Cambio de signo: hacer el complemento a 2 del número. Formas:
 - Hacer un NOT a todos los bits y sumarle uno a ese número.

$$12_{10} = 01100_2 \rightarrow 10011 \rightarrow 10011 + 1 \rightarrow 10100_{C2} = -12_{10}$$

- Recorrer el número de derecha a izquierda hasta encontrar un 1 y a partir de él invertir los demás bits restantes.

$$12_{10} = 01100_2 \rightarrow 10100_{C2} = -12_{10}$$

- Extensión: replicar el bit más significativo hacia la izquierda.
- Suma: suma binaria, **NO se tiene en cuenta acarreo del bit más significativo**
- Resta: se suele expresar como suma del opuesto, es decir, del complementado a dos del sustraendo.

- **Proceso de codificación y decodificación**

Para **codificar** un *número positivo* se siguen las reglas comentadas en el punto *Conversión entre sistemas*. Para *números negativos* se convierte como si fuese positivo y se realiza el complemento a 2 del resultado.

Para **decodificar**, para *números positivos*, decodificar como en *Conversión entre sistemas*. Para *números negativos*, hacer su complemento a 2 y decodificar como los positivos, añadiéndole después el signo.

Complemento a 1

Esta codificación representa números enteros.

- **Notación:**

- Positivos: $+N_{10} \rightarrow 0(N)_2$
- Negativos: $-N_{10} \rightarrow (2^n - 1 - N)_2 = C1(N_2)$

$$-12_{10} \rightarrow 2^5 - 1 - 12 = 19 \rightarrow 10011_{C1}$$

- **Rango de representación**⁵: $[-(2^{n-1} - 1), +(2^{n-1} - 1)]$

- **Aritmética**⁶:

- Cambio de signo: hacer el complemento a 1 del número. Formas:
 - Hacer un NOT a todos los bits

$$12_{10} = 01100_2 \rightarrow 10011_{C1}$$

- Extensión: replicar el bit más significativo hacia la izquierda.
- Suma: suma binaria, **NO se tiene en cuenta acarreo del bit más significativo**
- Resta: se suele expresar como suma del opuesto, es decir, del complementado a dos del sustraendo.

³El 0 ahora posee una única representación y el rango de representación es asimétrico (hay un negativo más)

⁴Cuando el bit de signo de los sumandos y del resultado coincide NO hay desbordamiento, cuando se suman número de distinto signo NUNCA hay desbordamiento

⁵El 0 posee doble representación

⁶Cuando el bit de signo de los sumandos y del resultado coincide NO hay desbordamiento, cuando se suman número de distinto signo NUNCA hay desbordamiento

■ Proceso de codificación y decodificación

Para **codificar** un *número positivo* se siguen las reglas comentadas en el punto *Conversión entre sistemas*. Para *números negativos* se convierte como si fuese positivo y se realiza el complemento a 1 del resultado.

Para **decodificar**, para *números positivos*, decodificar como en *Conversión entre sistemas*. Para *números negativos*, hacer su complemento a 1 y decodificar como los positivos, añadiéndole después el signo.

Decimal	MyS	C2	C1
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	1000	----	1111
-1	1001	1111	1110
-2	1010	1110	1101
-3	1011	1101	1100
-4	1100	1100	1011
-5	1101	1011	1010
-6	1110	1010	1001
-7	1111	1001	1000
-8	----	1000	----

Existen otros métodos de representación en binario como el **BCD** o el **Exceso⁷ 3** que codifican cada cifra por separado en grupos de 4 dígitos binarios:

$$375_2 \rightarrow 0011|0111|0101 \rightarrow 001101110101_{BCD}$$

$$375_2 \rightarrow 0110|1010|1000 \rightarrow 011010101000_{EX-3}$$

⁷El método de exceso 3 es igual que BCD, pero sumando a la cifra 3 unidades

ESPECIFICACIÓN DE SISTEMAS COMBINACIONALES

En este tema nos centramos en hallar la función que describe el comportamiento de esa “caja negra” que es nuestro sistema, es decir, de la foto de abajo hallar la función \mathbf{F} que describe $Z(t)$:



Para ello tendremos que describir:

- Valores de entrada → Dominio
- Salidas → Codominio
- Función: trabaja con los valores

ESPECIFICACIÓN DE ALTO NIVEL

En un primer momento procederemos a realizar una especificación de alto nivel, esto quiere decir, determinar los valores de entrada y salida que puede tomar nuestro sistema y el comportamiento lógico que la función debe tener a la hora de transformar nuestros valores:

Ej.:

- $x(t) = \{0, 1, 2, 3\}$
- $z(t) = \{0, 1, 2, \dots, 7\}$
- $f(t) = 2 \cdot x(t) + 1$

Ej.:

x	0	1	2	3	4	5	6	7
z	0	1	2	0	1	2	0	1

Ej.:

$$\begin{cases} 1 & \text{si } x(t) \text{ es primo} \\ 0 & \text{si } x(t) \text{ no es primo} \end{cases}$$

Sin embargo, para poder trabajar con estas expresiones es necesario codificarlas en alguna codificación binaria. Esto nos da como resultado en vez de entrada x , un vector de entrada x de n bits, siendo n el mínimo necesario para expresar todos los valores de entrada⁸.

$$x : (x_{n-1}, \dots, x_0)$$

Del mismo modo ocurre con la cantidad de valores de salida que ahora pasa a ser un vector de m bits⁹, siendo m el mínimo necesario para codificar todas las posibles combinaciones de salida.

$$z : (z_{m-1}, \dots, z_0)$$

Y existirán m funciones de salida para codificar una a cada bit de salida y que trabajaran con los n bits de entrada.

$$F = (f_{m-1}, \dots, f_0) : f_i = \{0, 1\} \rightarrow z_i : i = (0, m - 1)$$

Es decir, representando cada bit en una tabla¹⁰ con valores como $m = 3$ y $n = 2$:

x_1	x_0	z_1	z_2	z_3
0	0	0	1	1
0	1	1	1	0
1	0	1	0	0
1	1	0	1	0

En ocasiones las funciones no están completamente especificadas y decimos que están **parcialmente especificadas**. En estos casos los valores de entrada para los que no hay salida especificada se marcan con un “-” en la casilla de la salida correspondiente a la entrada sin asignación.

FUNCIONES DE CONMUTACIÓN

Una función de conmutación es una función que asocia un conjunto de n 0 y 1 a otro conjunto de 0 y 1.

$$f : 0, 1^n \mapsto 0, 1$$

El número de funciones que existen para n variables es 2^{2^n}

x_1	x_0	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

x_1	x_0	nula	and	x_1	x_0	xor	or	nor	xnor	not x_0	not x_1	nand	unidad
-------	-------	-------------	------------	-------	-------	------------	-----------	------------	-------------	------------------	------------------	-------------	---------------

En ocasiones estas están incompletamente especificadas y asocian el valor “-” a su salida.

⁸Expresar todos los valores de entrada no quiere decir expresar su magnitud en binario sino asociarle un número en binario que lo represente por lo que n tendrá que ver con la cantidad de valores de entrada, no con la magnitud que representen dichos valores

⁹No es necesario que m y n coincidan

¹⁰Los valores de z en este caso son aleatorios, debería ser los valores discretos que toman los sistemas en realidad

Expresiones de conmutación

Para representar estas funciones de forma más compacta, manipulable y sintetizable se emplean las expresiones de computación o EC.

$$F = \bar{A} + CD + [B \cdot (A + C)]$$

Estas son fórmulas que describen el comportamiento de la función, en función de los valores de las variables lógicas de entrada y que se rigen por las Leyes del Álgebra de Boole:

Leyes del Álgebra de Boole

1. **Ley de la dualidad:** cualquier expresión o identidad en el álgebra de Boole tiene su expresión dual intercambiando el + por la · y los 0 por los 1.

2. **Teorema de la identidad:** $B \cdot 1 = B$

Dual de la identidad: $B + 0 = B$

3. **Teorema del elemento nulo:** $B \cdot 0 = 0$

Dual de elemento nulo: $B + 1 = 1$

4. **Teorema de la potencia:** $B \cdot B = B$

Dual de la potencia: $B + B = B$

5. **Teorema de la involución:** $\bar{\bar{B}} = B$

6. **Teorema del complemento:** $B \cdot \bar{B} = 0$

Dual del complemento: $B + \bar{B} = 1$

7. **Teorema de la propiedad conmutativa:** $B \cdot C = C \cdot B$

Dual de la propiedad conmutativa: $B + C = C + B$

8. **Teorema de la propiedad asociativa¹¹:** $(B \cdot C) \cdot D = B \cdot (C \cdot D)$

Dual de la propiedad asociativa¹²: $(B + C) + D = B + (C + D)$

9. **Teorema de la propiedad distributiva:** $(B \cdot C) + (B \cdot D) = B(C + D)$

Dual de la propiedad distributiva: $(B + C) \cdot (B + D) = B + (C \cdot D)$

10. **Ley de Morgan:** $\overline{x \cdot y} = \bar{x} + \bar{y}$

Dual de la Ley de Morgan: $\overline{x + y} = \bar{x} \cdot \bar{y}$

Valores de una EC

El valor de una EC para una combinación de entrada o asignación concreta se expresa como:

$$v(E, a)$$

Ej.: $E = A + \bar{B}C + B$, $a = (0, 1, 1)$

$$v(E, a) = v(A + \bar{B}C + B, (0, 1, 1)) = 0 + 0 \cdot 1 + 1 = 1$$

¹¹Quiere decir que el resultado de la aplicación de las dos puertas AND es el mismo que el de la aplicación de una única puerta AND de 3 entradas.

¹²Quiere decir que el resultado de la aplicación de las dos puertas OR es el mismo que el de la aplicación de una única puerta OR de 3 entradas.

Suma de productos

Cualquier expresión de commutación puede ser descrita como suma de mintérminos. Estos se corresponden a cada fila de la tabla de verdad y constituyen un producto (AND) de las variables y sus complementarias.

1. Recogemos en una tabla de verdad todo los posibles casos de entrada y las SALIDAS CORRESPONDIENTES POR OBSERVACIÓN EMPÍRICA (no tenemos por qué saber el por qué de esas salidas).
2. Escribimos en la tabla de los mintérminos el correspondiente en función de los valores de A y de B, siendo el mintérmino un producto de ambas variables y cada variable ser A, si vale 1 en la tabla de verdad, o \bar{A} si su valor es un 0.
3. La función se construye solo con la SUMA DE MINTÉRMINOS VERDADEROS, que son los que están en una fila donde la salida sea 1.

¹³

A	B	F	Mintérmino
0	0	0	$\neg A \cdot \neg B$
0	1	1	$\neg A \cdot B$
1	0	0	$A \cdot \neg B$
1	1	1	$A \cdot B$

Ej.:

$$F = \sum m(1, 3) = \bar{A} \cdot B + A \cdot \bar{B}$$

Simplificación de EC

Simplificación algebraica

Se trata de simplificar una EC dada usando las leyes básicas del álgebra de boole:

1. Pasar de una expresión cualquiera a una SPC

Para pasar de una expresión simplificada a una SPC, damos TODOS los valores posibles a las variables que no están en los mintérminos simplificados.

Ej.:

$$f(x, y, z, w) = \bar{y}z + y\bar{z} \Rightarrow \begin{cases} \bar{y}z &= \bar{x}\bar{y}z\bar{w} + \bar{x}\bar{y}zw = x\bar{y}z\bar{w} + x\bar{y}zw \\ y\bar{z} &= \bar{x}y\bar{z}\bar{w} + \bar{x}y\bar{z}w + xy\bar{z}\bar{w} + xy\bar{z}w \end{cases} \Rightarrow f(x, y, z, w) = \sum m(2, 3, 4, 5, 10, 11, 12, 13)$$

2. Simplificar dicha SPC usando las reglas de las leyes de commutación (las variables que cambien entre minterms con variables comunes se simplificarán al final del proceso)

Mapas de Karnaugh

Para facilitar la realización de este proceso utilizando la mínima cantidad de recursos posibles (cuantos menos términos menores puertas lógicas) usamos los Diagramas de Karnaugh.

Se tratan de una secuencia de celdas, en la que cada celda representa un valor binario de las diferentes entradas. Además, cada celda contiene dentro de ella el valor de salida correspondiente a la combinación de entrada que representa dicha celda en la tabla de verdad.

¹³Se ha puesto el símbolo \neg por confusión del signo $\bar{}$ con las líneas

Es muy importante distribuir las celdas de forma que SOLO SE DIFERENCIEN EN 1 BIT, por ello es por lo que en el dibujo se han cambiado las dos últimas celdas. Pueden usarse para expresiones de como MÁXIMO 6 variables y para n variables, es necesario utilizar 2^n celdas.

- **Diagramas de Karnaugh de tres variables**

- A, B, C: Variables. Conjunto de 8 celdas
- Los valores binarios de A y B se encuentran en la parte izquierda y los de C en la parte superior (puede hacerse al revés)

AB	C	0	1
00		1	0
01		0	0
11		0	0
10		0	1

AB	C	0	1
00		$(A \cdot B \cdot \bar{C})$	$(\bar{A} \cdot B \cdot C)$
01		$(\bar{A} \cdot B \cdot \bar{C})$	$(\bar{A} \cdot B \cdot C)$
11		$(A \cdot B \cdot \bar{C})$	$(A \cdot B \cdot C)$
10		$(A \cdot B \cdot C)$	$(\bar{A} \cdot B \cdot C)$

Ejemplo: La función vale '1' en la celda de la esquina superior izquierda corresponde a un valor de las variables A, B y C de 000 ($/A \cdot /B \cdot /C$) y en la celda de la esquina inferior derecha corresponde a un valor de las variables de 101 ($A \cdot B \cdot C$)

- **Diagramas de Karnaugh de cuatro variables**

- A, B, C, D: Variables. Matriz de 16 celdas
- Los valores binarios de A y B se muestran en la parte izquierda de la tabla y los de C y D en la parte superior

AB	CD	00	01	11	10
00		0	0	0	1
01		1	0	0	0
11		0	0	0	0
10		0	1	0	0

AB	CD	00	01	11	10
00		$(\bar{A} \cdot B \cdot \bar{C} \cdot \bar{D})$	$(A \cdot B \cdot \bar{C} \cdot \bar{D})$	$(\bar{A} \cdot B \cdot C \cdot \bar{D})$	$(A \cdot B \cdot C \cdot \bar{D})$
01		$(\bar{A} \cdot B \cdot \bar{C} \cdot D)$	$(A \cdot B \cdot \bar{C} \cdot D)$	$(\bar{A} \cdot B \cdot C \cdot D)$	$(A \cdot B \cdot C \cdot D)$
11		$(A \cdot B \cdot \bar{C} \cdot \bar{D})$	$(A \cdot B \cdot \bar{C} \cdot D)$	$(A \cdot B \cdot C \cdot \bar{D})$	$(A \cdot B \cdot C \cdot D)$
10		$(A \cdot B \cdot \bar{C} \cdot D)$	$(A \cdot B \cdot \bar{C} \cdot D)$	$(A \cdot B \cdot C \cdot \bar{D})$	$(A \cdot B \cdot C \cdot D)$

Ejemplo: La función vale '1' en la celda de la esquina superior derecha corresponde a un valor de las variables A, B, C y D de 0010 ($/A \cdot /B \cdot C \cdot D$), en la celda de la segunda fila izquierda que corresponde a un valor de 0100 ($/A \cdot B \cdot C \cdot D$) y en la última fila que corresponde a un valor de 1001 ($/A \cdot /B \cdot C \cdot D$).

- **Diagramas de Karnaugh de cinco variables**

- A, B, C, D, E: Variables. Matriz de 32 celdas
- Se consideran dos tablas de cuatro variables (B, C, D y E), una correspondiendo al valor de la variable A=0 y otra a A=1
- Los valores binarios de B y C se muestran en la parte izquierda de la tabla y los de D y E en la parte superior

BC	DE	00	01	11	01	00	01	11	10
00									
01									
11									
10									

$A = 0$ $A = 1$

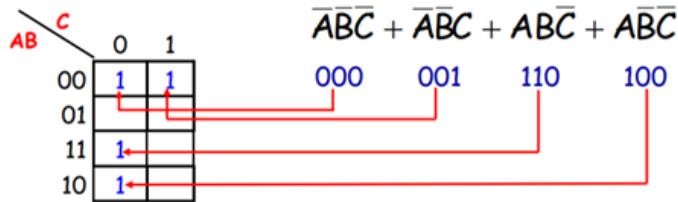
Los mapas de Karnaugh no se usan únicamente como una forma condensada de una tabla de verdad, sino que además permiten por su disposición la MINIMIZACIÓN de la expresión resultante. Para ello es importante tener claro el concepto de adyacencia y las condiciones de formación de grupo:

1. Entre celdas adyacentes solo cambia una variable, es decir, un bit.
2. Solo se consideran adyacentes las que estén situadas inmediatamente por cualquier lado.
3. Las celdas en diagonal no son adyacentes.
4. Las celdas siguen ADYACENCIA CÍCLICA, las del borde de la izquierda lo son con las de la derecha y las del borde de arriba con el de abajo.

Para realizar esta minimización es necesario traducir de Minterms o Maxterms a mapa de Karnaugh si tenemos solo la expresión de la función que se quiere reducir ($F = ABC + \dots$) o simplemente proporcionar el valor 0 o 1 de la tabla de verdad en la combinación de entradas correspondiente:

- **Diagrama de Karnaugh de una suma de productos estándar**

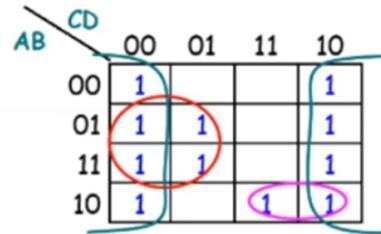
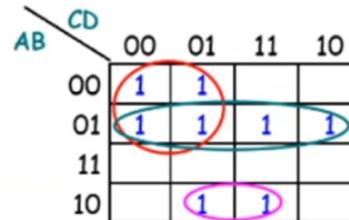
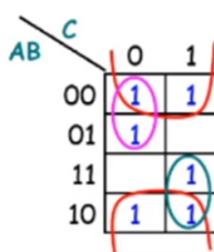
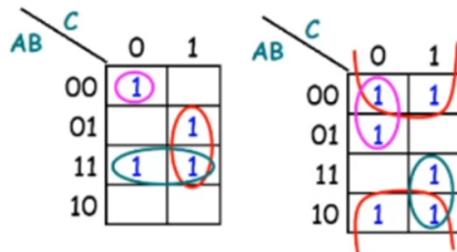
- Por cada término de la expresión suma de productos, se coloca un 1 en el diagrama de Karnaugh en la celda correspondiente al valor del producto



La agrupación de 1 o 0 se hace en base a las siguientes normas:

- Cada grupo debe contener un número de unos correspondiente a un múltiplo de 2 (los grupos siempre serán rectangulitos o cuadrados)
- Cada celda del grupo debe ser adyacente a una o más celdas del grupo
- Todos los 1 o 0 deben estar en algún grupo y los grupos deben tener el máximo de ellos, aunque un elemento este en varios grupos.

- **Agrupación de 1's**



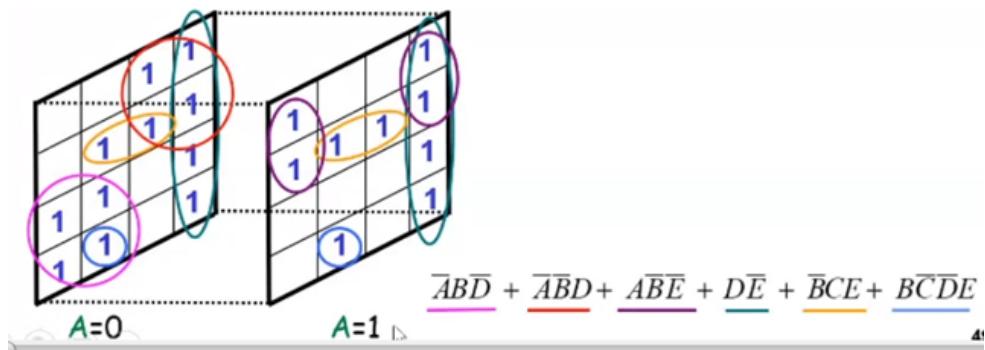
Una vez agrupados los términos de la forma correspondiente:

- Cada grupo constituye un producto de todas las variables que posteriormente se sumará a los demás
- En un mismo grupo, SOLO SE MANTIENEN las variables que NO VARÍAN
- Se escribe como \bar{A} o como A la variable en función de si vale 1 (minterms) o 0 (maxterms)

Para los mapas de 5 variables, las normas son las mismas con la peculiaridad de colocar uno encima del otro:

		DE	BC						
		00	01	11	10	00	01	11	10
		00			1	1			
		01		1	1		1	1	
		11	1	1		1			
		10	1	1		1			
		$A=0$		$A=1$					

$$\underline{\overline{AB}\overline{D}} + \underline{\overline{A}\overline{B}D} + \underline{A\overline{B}\overline{E}} + \underline{D\overline{E}} + \underline{\overline{B}CE} + \underline{B\overline{C}\overline{D}E}$$

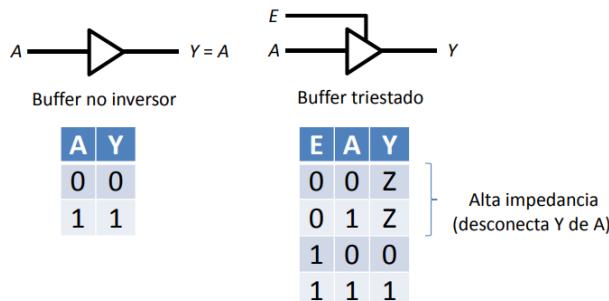


IMPLEMENTACIÓN DE SISTEMAS COMBINACIONALES

PUERTAS LÓGICAS

Son elementos lógicos digitales a partir de los cuales se crean funciones lógicas de forma física:

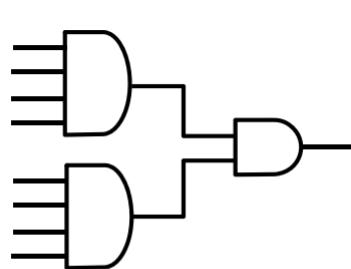
- **BUFFER o Amplificador** → Los datos de entrada son los mismos que los de salida.
 - Buffer triestado: posee una variable auxiliar “enable” que habilita o no el buffer.



- **NOT o Inversor** → Los datos de entrada son contrarios a los de salida.
- **AND o Multiplicación Lógica** → La salida solo vale uno si ambas entradas valen 1.
- **OR o Suma Lógica** → La salida vale 1 si al menos uno de los datos de entrada vale 1 (o todos).
- **NAND** → Funciona como una puerta AND, pero al resultado se le aplica el NOT.
- **NOR** → Funciona como una puerta OR, pero al resultado se le aplica el NOT.
- **XOR** → Es como la puerta OR, pero excluyente: la salida es 1 si solo una de sus entradas vale 1.
- **XNOR** → Es como la puerta XOR, pero cuando solo una de sus entradas vale 0 su salida vale 0.

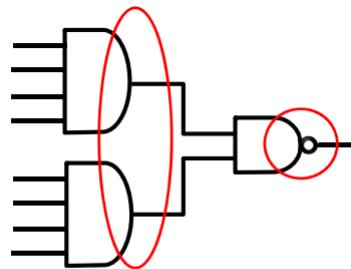
NOT	AND	NAND	OR	NOR	XOR	XNOR
\bar{A}	AB	\bar{AB}	$A+B$	$\bar{A}+B$	$A \oplus B$	$\bar{A} \oplus B$

No es necesario que las puertas tengan únicamente dos entradas, sino que pueden ser las que queramos, aunque en la práctica es habitual encontrar un número elevado de ellas. En lugar de ello se recurre a la implementación en árbol:



Implementación en árbol

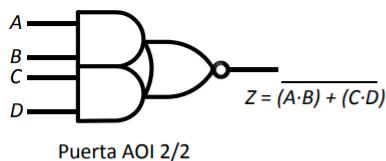
Puerta AND de 8 entradas



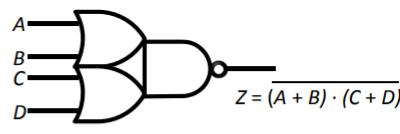
Implementación en árbol

Puerta NAND de 8 entradas

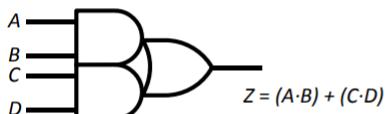
Existen a su vez puertas compuestas homologadas como:



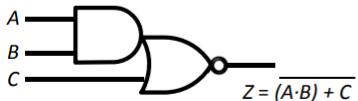
Puerta AOI 2/2



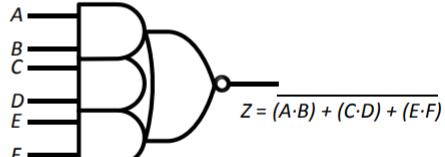
Puerta OAI 2/2



Puerta AO 2/2



Puerta AOI 2/1



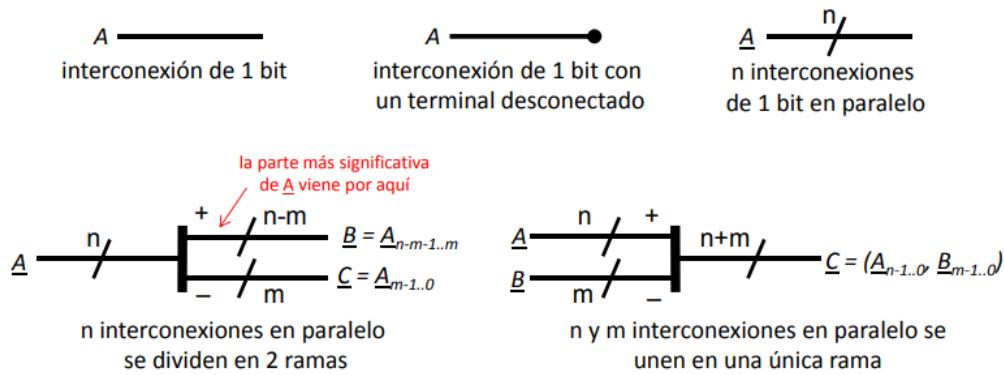
Puerta AOI 2/2

y algunas más...

Definiciones

- **Módulo:** dispositivo que realiza físicamente una función conocida de cualquier complejidad.
Ej.: Puertas lógicas, microchips, memorias RAM...
- **Puerto:** cada línea de entrada o salida que comunica el módulo con el exterior.

- **Interconexión:** unión de dos o más puertos entre sí. Notación:



- **Red:** colección de módulos interconectados de manera que toda ENTRADA SOLO VIENE de UNA SALIDA. (Una salida si puede estar conectada a más de una entrada).
- **Red combinacional:** red de módulos combinacionales en las que no hay retroalimentaciones (porque hay sistemas secuenciales implementados con módulos combinacionales), es decir, que ningún recorrido de la red pasa por el mismo punto dos veces.
- **Nivel de una red:** número máximo de módulos que atraviesa cualquier camino que conecta una entrada con una salida. (Cuando la red es de puertas no solemos contar con los inversores).

Conjunto Universal

Decimos que un conjunto de módulos combinacionales es universal cuando permite implementar cualquier fórmula empleando sólo dichos módulos: Ej.:

- {AND, NOT, OR}

$$a + b = \overline{\overline{a} + \overline{b}} = \overline{a} \cdot \overline{b}$$

- {NAND}

$$\bar{a} = \overline{a \cdot a}$$

$$a \cdot b = \overline{\overline{a} \cdot \overline{b}}$$

$$a + b = \overline{\overline{a} + \overline{b}} = \overline{a} \cdot \overline{b}$$

- {OR, NOT}, {NOR}, {XOR, AND}, ...

SÍNTESIS DE REDES DE PUERTAS

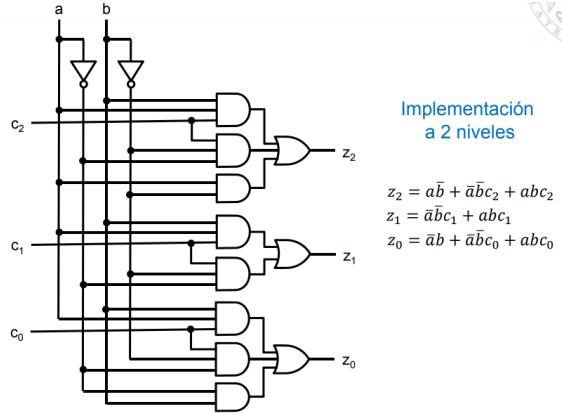
El proceso de automatización de la descripción de un proceso lógico termina con la implementación en un circuito de la expresión booleana resultante, es decir, para crear un sistema físico que describa el comportamiento de mi función lógica seguimos estos pasos:

1. Especificación de alto nivel
2. Codificación binaria de la especificación hecha
3. Simplificación de la codificación a una EC simplificada
4. Implementación del circuito por medio de las puertas lógicas

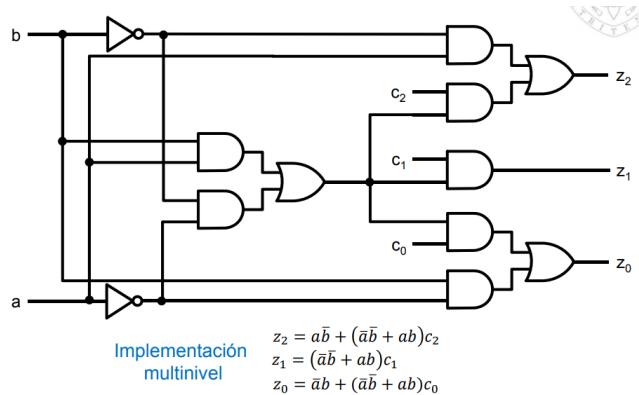
Dentro de este último paso conocemos dos implementaciones posibles distintas:

■ Implementación a dos niveles

- Implementación canónica: implementa la SPC
- Implementación mínima: implementa la EC simplificada

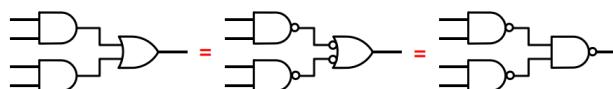


■ Implementación multinivel: se realiza una simplificación heurística de la EC



Para realizar la implementación de un circuito con el empleo exclusivo de puertas NAND, se sigue el siguiente procedimiento:

■ 2 niveles AND-OR equivalen a 2 niveles NAND-NAND



■ Método:

- Obtener una red AND-OR.
- Añadir pares de inversores a las salidas de las puertas AND (o a las entradas de las puertas OR).
- Uniformizar la notación de las puertas NAND.
- Eliminar dobles inversores donde sea posible.
- Reemplazar inversores por su implementación con NAND.

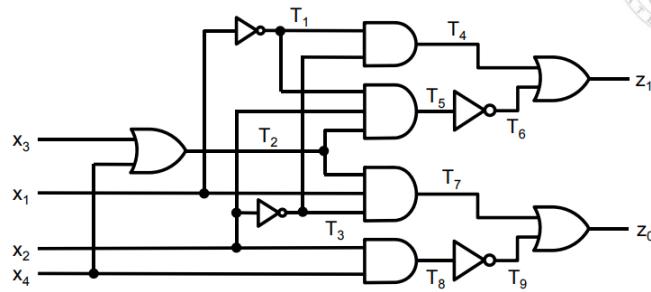
Concretamente, para las expresiones de comutacion, la expresion se transforma negando dos veces la expresion resultante y transformando de la siguiente forma:

ANÁLISIS DE REDES DE PUERTAS

En ocasiones, nuestro trabajo no es implementar un circuito sino deducir la EC que un circuito ha implementado, para ello se siguen los siguientes pasos:

1. Asignar una variable a cada una de las interconexiones
2. Calcular la EC de cada una de esas variables
3. Sustituir su valor en las sucesivas EC hasta obtener una única EC
4. Simplificar dicha expresión

Análisis de redes AND-OR



$$T_1 = \overline{x_1} \quad T_4 = T_1 T_3 = \overline{x_1} \overline{x_2}$$

$$T_2 = x_3 + x_4 \quad T_5 = T_1 x_2 T_2 = \overline{x_1} x_2 (x_3 + x_4)$$

$$T_3 = \overline{x_2} \quad T_7 = T_2 x_1 T_3 = (x_3 + x_4) x_1 \overline{x_2}$$

$$T_8 = x_2 x_4 \quad T_6 = \overline{T_5} = \overline{\overline{x_1} x_2 (x_3 + x_4)}$$

$$T_9 = \overline{T_8} = \overline{x_2} \overline{x_4}$$

$$z_0 = T_7 + T_9 = (x_3 + x_4) x_1 \overline{x_2} + \overline{x_2} \overline{x_4}$$

$$z_1 = T_4 + T_6 = \overline{x_1} \overline{x_2} + \overline{x_1} x_2 (x_3 + x_4)$$

MÓDULOS COMBINACIONALES

En este tema estudiaremos los diferentes módulos combinacionales más básicos y su funcionamiento elemental. En ocasiones, el uso de puertas lógicas para implementación de funciones de conmutación se vuelve engorroso y muy complejo, fruto de facilitar y automatizar dicho trabajo nacen los módulos combinacionales, cuya funcionalidad es la implementación de funciones más complejas con métodos más mecánicos y versátiles (puesto que los módulos son piezas de lego ya definidas y combinables, es decir, más escalables).

Decodificador

Es un módulo combinacional con las siguientes características:

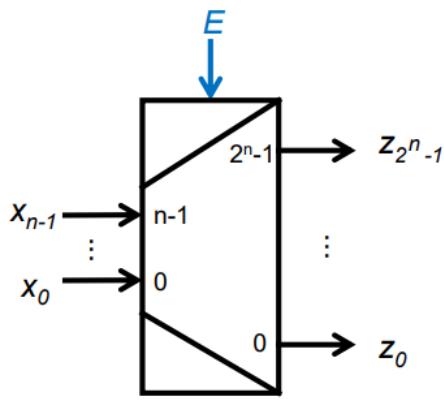
- Posee n entradas y 2^n salidas.
- El valor de la salida se define¹⁴ ¹⁵ de la siguiente forma:

$$(v)_{10} = (x_0, x_1, \dots, x_n)_2 \quad (1)$$

$$z_i = \begin{cases} 1 & \text{si } v = i \\ 0 & \text{si } v \neq i \end{cases} \quad (2)$$

- Puede poseer o no una entrada de control llamada *enable*, que cuando vale 0 convierte todos los bits de salida en 0 y cuando vale 1 sigue la disposición nombrada antes.

El símbolo empleado para denotar a un decodificador es:



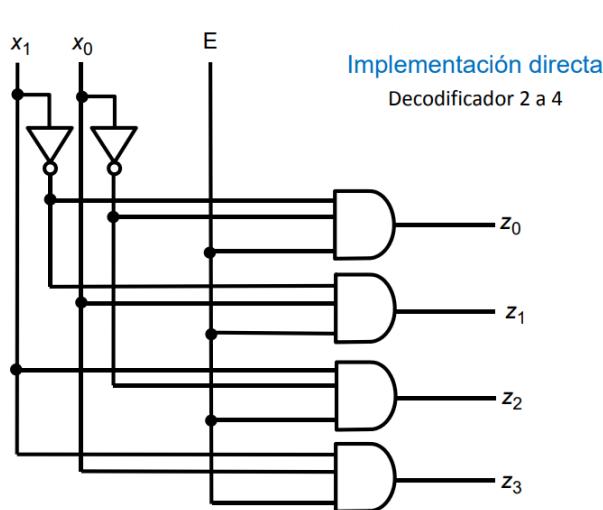
Decodificador n a 2^n

¹⁴Llamamos v al valor del decimal de la entrada codificada en binario.

¹⁵Llamamos z_i al bit del vector de salida que se activa cuando la entrada es concreta.

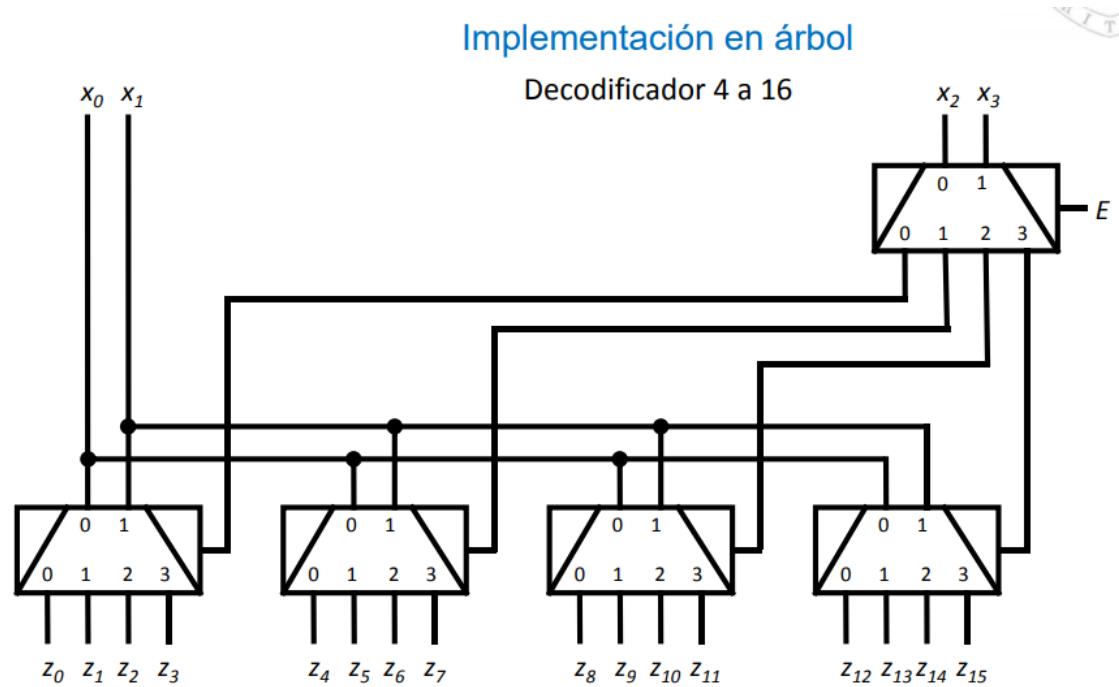
Es decir, cuando el número decimal de entrada es el x , entonces la salida z_x se activa y el resto distintas a esa salida no, si existe un enable, es como si este “apagara” o encendiese el módulo, cuando vale 0 no funciona.

Veamos un ejemplo de implementación de un decodificador de 2 entradas y 4 salidas con Enable. La implementación directa de puertas AND quedaría de la siguiente forma:



E	x_0	x_1	z_0	z_1	z_2	z_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

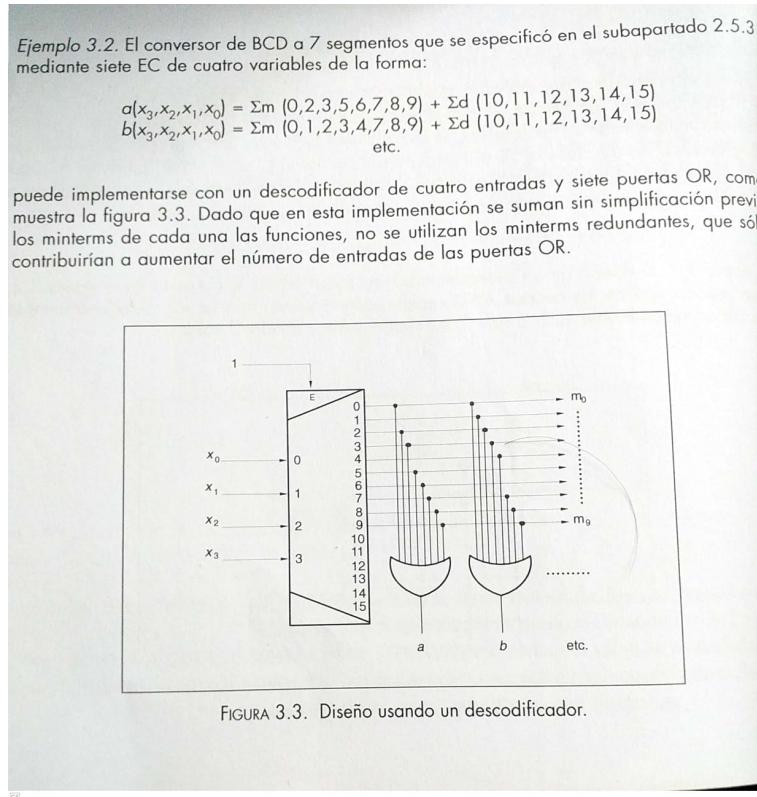
Estos módulos se pueden combinar entre sí para implementar otro del mismo tipo pero con mayor número de entradas, lo que le confiere la ventaja de escalabilidad característica de los módulos combinacionales, por ejemplo:



Aplicaciones de diseño

Se puede observar que un decodificador de n entradas posee 2^n salidas y, en consecuencia, se ve que cada puerta AND conectada a una de las salidas permite implementar todos cada minterm de una función de conmutación concreta.

Dicho esto uno puede pensar que si el decodificador implementa todos los minterms y cualquier función de conmutación se puede escribir como *Suma de Productos Canónica*, entonces si utilizamos unas puertas *OR* para decidir qué minterms implementar y cuáles no, ya tendríamos la posibilidad de implementar cualquier FC de n variables.



Es decir, suele resultar útil para implementar funciones con m salidas dependientes de n entradas porque cada salida del decodificador implementa un minterm y **una puerta OR** por cada salida m implementa la suma de minterms de cada salida.

Codificador

Realiza la función contraria al decodificador, en este caso:

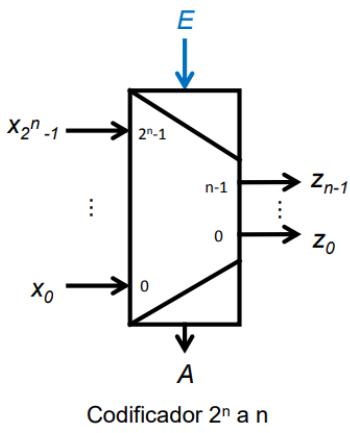
- Posee 2^n entradas y n salidas.
- El valor de la salida se define de la siguiente forma:

$$Z = (z_0, \dots, z_n)_2 = (i)_2 : x_i = 1$$

- **Siempre** debe tener una entrada de control llamada *enable*, que cuando vale 0 inhabilita la salida.¹⁶
- Posee una salida adicional, **la salida de actividad A** que indica si está activo o no, es decir, la salida A vale 1 si y sólo si alguna de las entradas es 1.

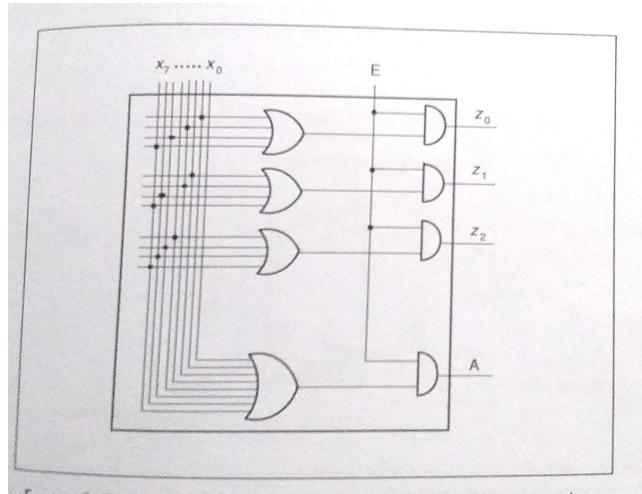
$$A = \begin{cases} 1 & \text{si } E = 1 \wedge \exists i : x_i = 1 \\ 0 & \text{en caso contrario} \end{cases}$$

¹⁶Es fácil ver que para la codificación de entrada de todo 0, se puede confundir con la que está solo x_0 activada



E	x_0	x_1	x_2	x_3	z_0	z_1	A
0	-	-	-	-	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	1	1
1	0	1	0	0	1	0	1
1	1	0	0	0	1	1	1

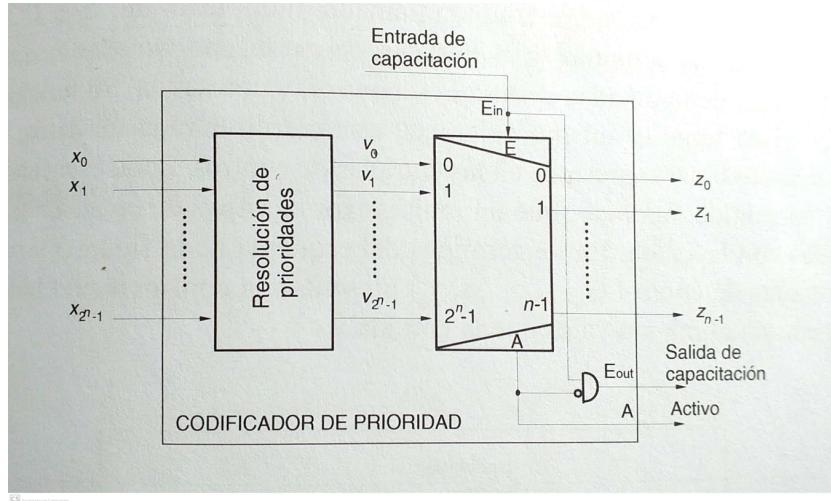
Es decir, la salida codifica en binario el valor del subíndice de la entrada activada. La función de la salida de actividad es distinguir cual de los 3 casos en los que salida es 0 es el válido y real para las entradas que hay en ese momento, es decir, cuando la única variable activa es x_0 y el enable $E = 1$, entonces $A = 1$.



Este es un ejemplo de la implementación directa de un codificador de 8 a 3. Se puede observar que la salida A solo vale 1 cuando alguna de las entradas está activa y que las salidas z_i solo pueden valer 1 si el *enable* se lo permite. De este modo cada puerta *OR* da valor 1 a la salida correspondiente cuando algunas de las entradas x_i con las que se activa, está activa.

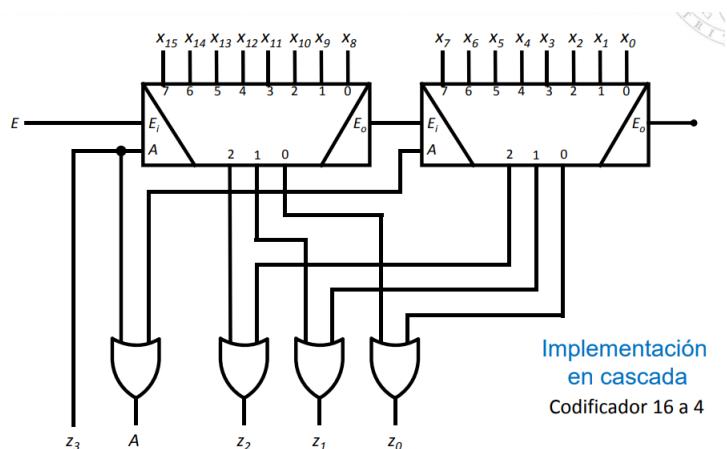
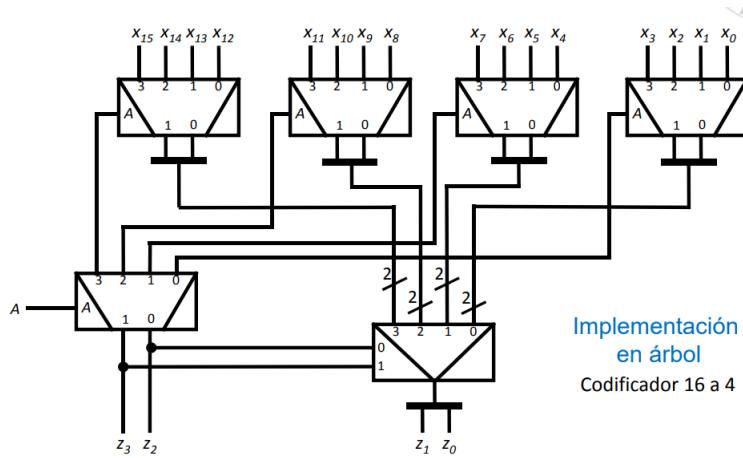
Codificador de prioridad

Se ve de la definición prestada antes que el caso en el que más de una entrada está activa supone un problema de indeterminación en el diseño, para evadir dicho vacío lógico sirve el **codificador de prioridad** cuya función es establecer prioridades entre las diferentes entradas del mismo.

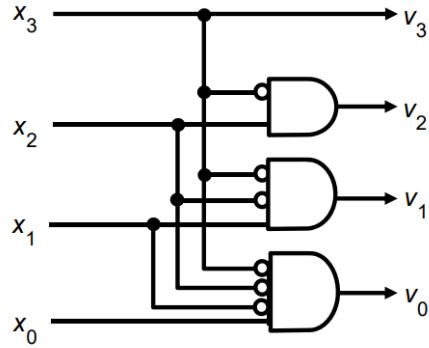


Podemos observar que lo único necesario para poder implementar este tipo de codificador es un módulo anterior que resuelva las prioridades y le pase a un codificador usual las entradas correspondientes de la forma en que se ha explicado en el apartado anterior (todo 0 menos alguna en 1).

Podemos ver en esta nueva definición una nueva salida, E_{out} o *salida de capacitación* que, como vemos, solo vale 1 si la entrada de capacitación es 1, es decir, está habilitado, pero la salida de actividad vale 0, es decir, no está en funcionamiento, a pesar de que está habilitado. La función de esta nueva salida no es más que posibilitar la combinación de varios de estos módulos para obtener uno igual pero con mayor número de entradas.

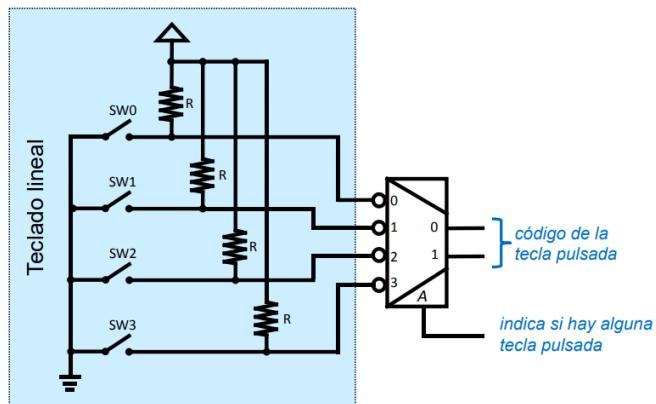


La implementación directa del resolutor de prioridades comentado anteriormente quedaría de la siguiente forma¹⁷:



Aplicaciones de diseño

Una de las posibles aplicaciones es la asociación de un código a cada componente de un vector de entrada. Supongamos que tenemos un teclado con 30 teclas, entonces serían necesarias 2^5 entradas en un decodificador y cada tecla pondría a 1 su x_i correspondiente, esto haría que cada tecla tuviera asociado un vector en binario de salida que identificaría dicha tecla.



Multiplexor

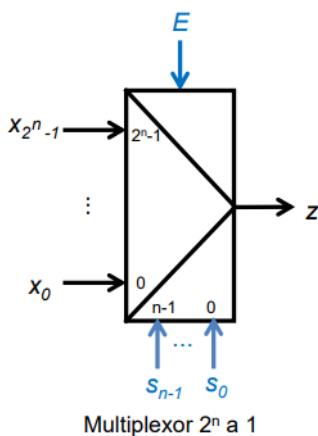
Es un módulo combinacional que redirige el flujo de entrada desde una de las entradas a una de las salidas y posee las siguientes partes:

- 2^n entradas binarias (x_i)
- n entradas de control (s)
- Una única salida (z)
- Puede tener un enable (E)

$z = a_i$ donde i es la el valor en decimal de la codificación del número de n bits de la entrada de control (s):

$$z = E \cdot \sum(x_i \cdot m_i(s))$$

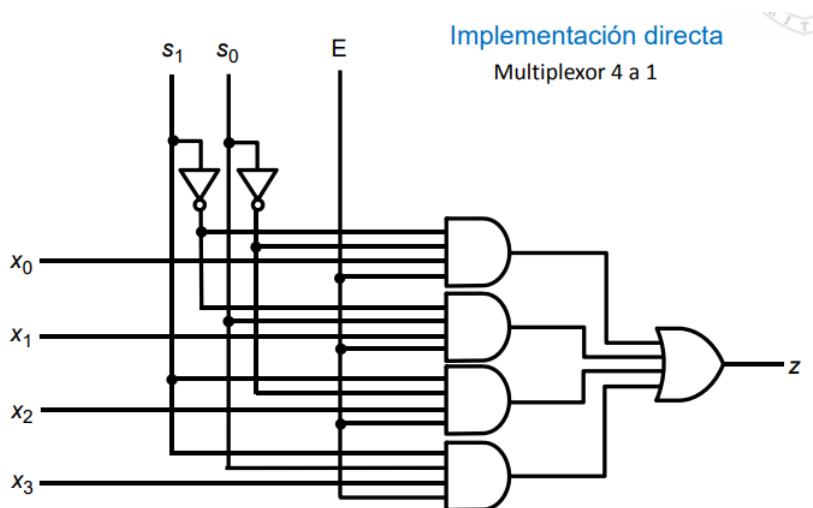
¹⁷Se puede apreciar por las características de su diseño que este es fácilmente escalable al número de entradas deseado por lo que no supone mayor complicación fabricar uno más grande



s	x_0	x_1	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Se observa que cuando la entrada de control vale 1, la salida z toma el valor que tuviese en ese momento la entrada x_1 y cuando vale 0, el valor de x_0 .

La implementación del circuito interno quedaría de la siguiente forma:

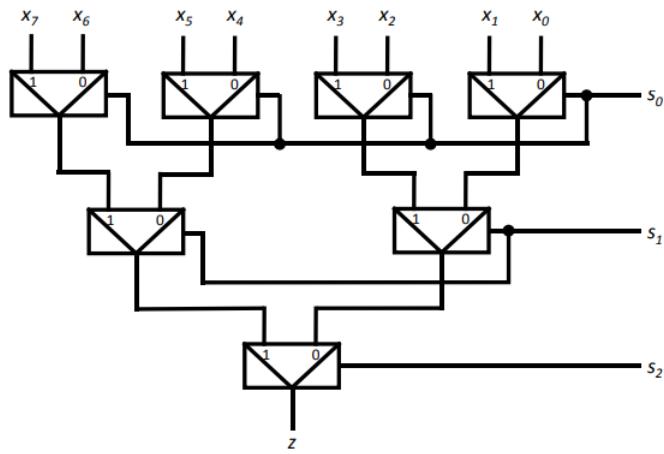


Es fácil ver que la implementación interna del funcionamiento de este dispositivo tiene mucho que ver con la del decodificador, hasta tal punto que la única diferencia entre ambos es la inclusión de una puerta OR en el multiplexor por lo que la implementación de este módulo también puede hacerse a partir de un decodificador cuyas salidas están conectadas a una puerta OR que define la función del multiplexor en contra posición a la del decodificador.

La implementación de un multiplexor de más entradas es posible a través de la combinación de otros más pequeños. El procedimiento más habitual es el siguiente:

Implementación en árbol

Multiplexor 8 a 1

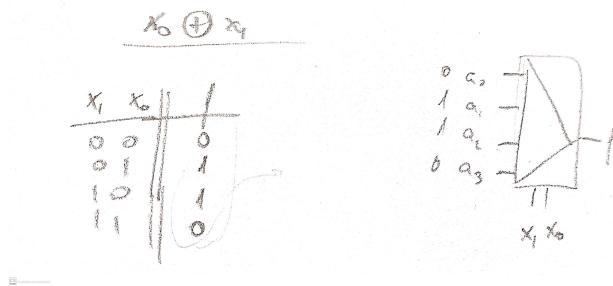


Aplicaciones de diseño

Es razonable pensar que si un decodificador junto con varias puertas *OR* era capaz de implementar cualquier función de conmutación como suma de productos canónica, como el multiplexor lleva ya implementada dicha puerta, entonces un multiplexor es capaz de implementar cualquier FC. Y en efecto, el multiplexor es un **conjunto universal** de los estudiados anteriormente, aunque el diseño con el uso exclusivo de los mismo puede no ser siempre demasiado intuitivo, pero sí posible.

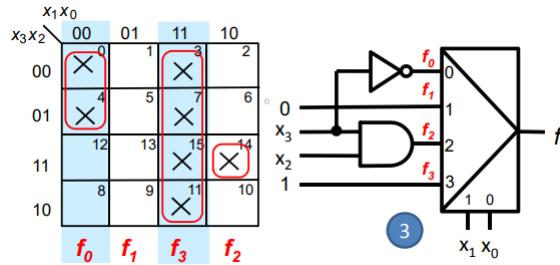
Un procedimiento general para implementar cualquier función de n variables es el siguiente:

- El vector de entradas de control será el vector formado por las variables de la FC.
- En el vector de entradas de datos, cada x_i será el valor de cada minterm de la FC **en orden**, es decir, cada x_i será el valor de la función en ese punto del dominio.



Cuando tenemos que implementar una función de $n + k$ variables y tenemos solo disponible un multiplexor de n entradas de control el procedimiento es:

- Elegimos n variables que serán nuestras entradas de control.
- De la suma de productos canónica, extraemos factor común de esas variables en todas las combinaciones posibles y simplificamos el resultado.
- Cada paréntesis resultante es una función de conmutación auxiliar que va a cada entrada de datos del multiplexor.



Conviene saber que la elección de las variables de control determina si una implementación es más eficiente que otra o la forma de escoger las f auxiliares porque, una vez hechos los grupos de karnaugh según las reglas ya explicadas no podemos separar a miembros del mismo grupo y como se ve en el siguiente ejemplo, el considerar que cada f es una fila porque son x_0 y x_1 las que son variables de control ya no es posible y hay que elegir otras.

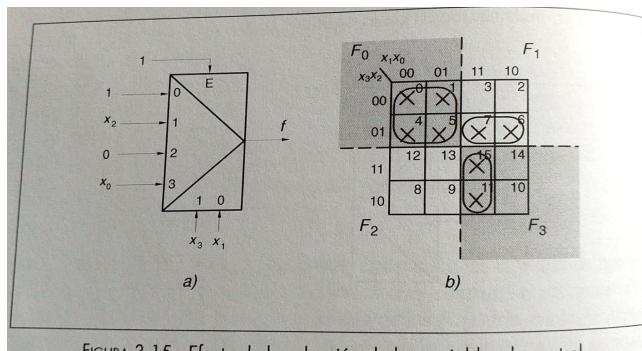
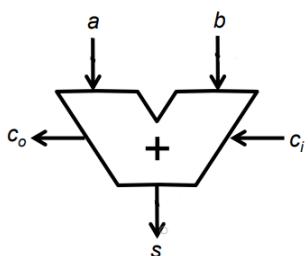


FIGURA 3.15. Efecto de la selección de las variables de control.

Sumador/Restador

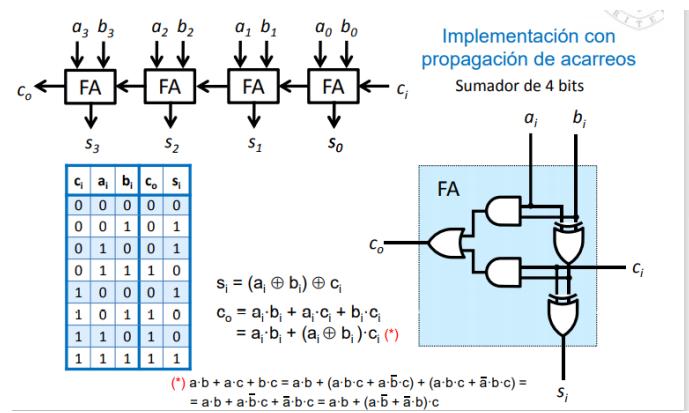
Sumador

Las entradas son 3, dos bits que hay que sumar y un bit que determina si esa operación tiene un acarreo de otra anterior o no (porque se suelen usar combinados), las salidas son 2, una que determina el resultado de la suma de los dos bits de entrada y otra que le añade a la salida un un bit si hubiese acarreo.



c_i	a	b	c_0	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

El funcionamiento es similar para un sumador más de 2 bits. Se colocan en serie varios sumadores simples de modo que el acarreo de salida de uno sea el de entrada del siguiente así que realizan el proceso de sumar a mano que conocemos de toda la vida, el aspecto de la combinación de varios para sumar vectores binarios es:



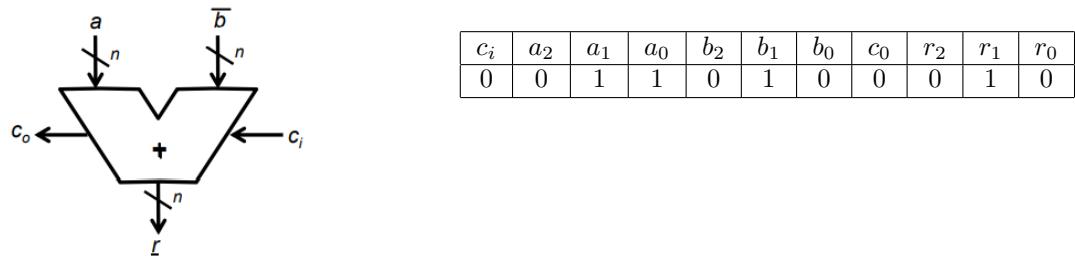
Este sumador sumaría dos números de 4 bits cada uno.

Restador

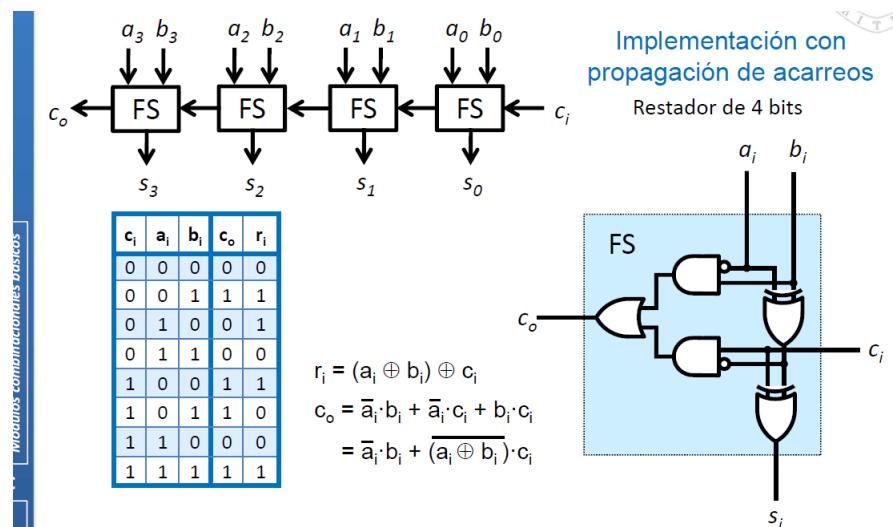
Los ordenadores en general suele funcionar en C_2 , para ello la operación de resta se puede expresar como:

$$A - B = A + B_{C_2} = A + (B_{C_1} + 1) = A + \bar{B} + 1$$

Quedando el sumador resultante como:

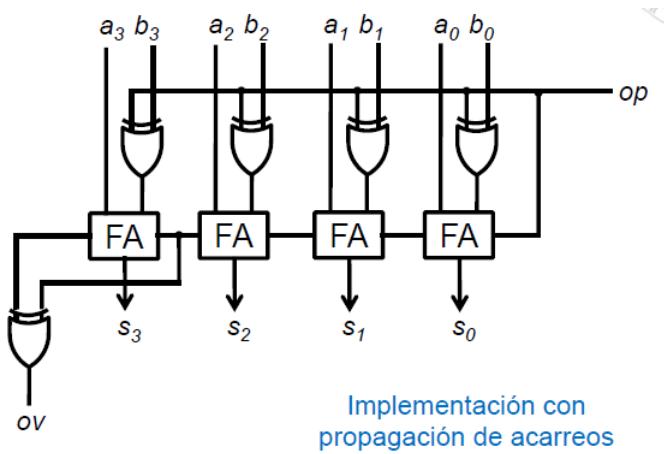
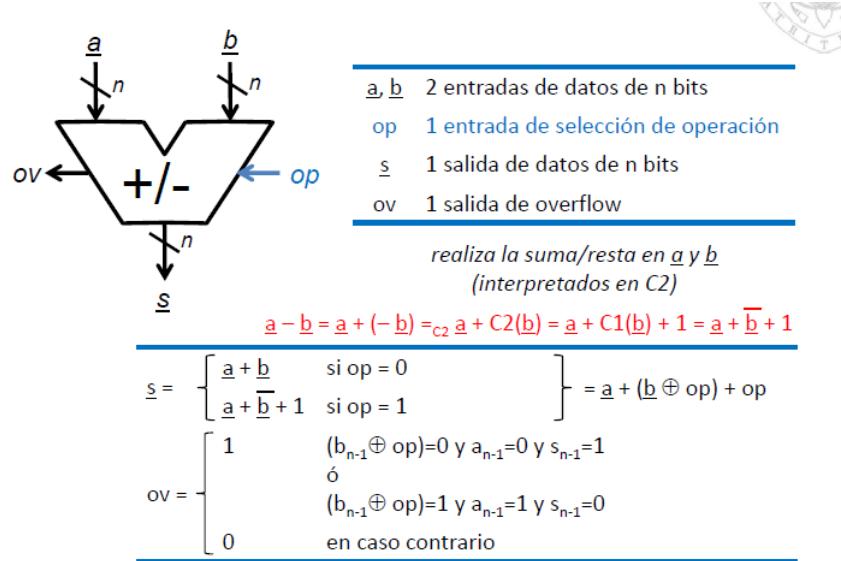


De este modo, el restador no es más que un sumador de opuestos 1. Del mismo modo, varios se pueden componer en serie para conseguir uno de mayor anchura.



Sumador/Restador

Para no tener que distinguir entre ambos módulos y al observar que en esencia ambos son sumadores (aunque uno de ellos sume el opuesto), podemos implementar uno único módulo que realice la función de ambos y que pueda seleccionar que operación hacer.

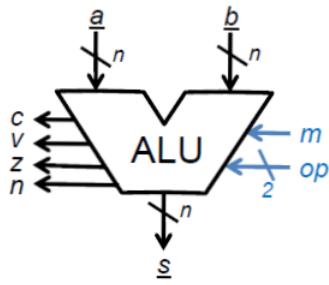


ALU

Esta unidad realiza las operaciones aritmético-lógicas básicas de un microprocesador. Este módulo posee:

- 2 entradas de datos de n bits
- 1 entrada de selección de modo (aritmético o lógico)
- 1 entrada de selección de operación¹⁸
- 1 salida de datos de n bits
- 4 salidas de control de datos como se especifica en la imagen

¹⁸Será de un número n de bits en función del número de operaciones que soporte



a, b 2 entradas de datos de n bits

m 1 entrada de selección de modo

op 1 entrada de selección de operación

s 1 salida de datos de n bits

c 1 salida de acarreo

v 1 salida de overflow

z 1 salida de detección de cero

n 1 salida de detección de negativo

operaciones lógicas

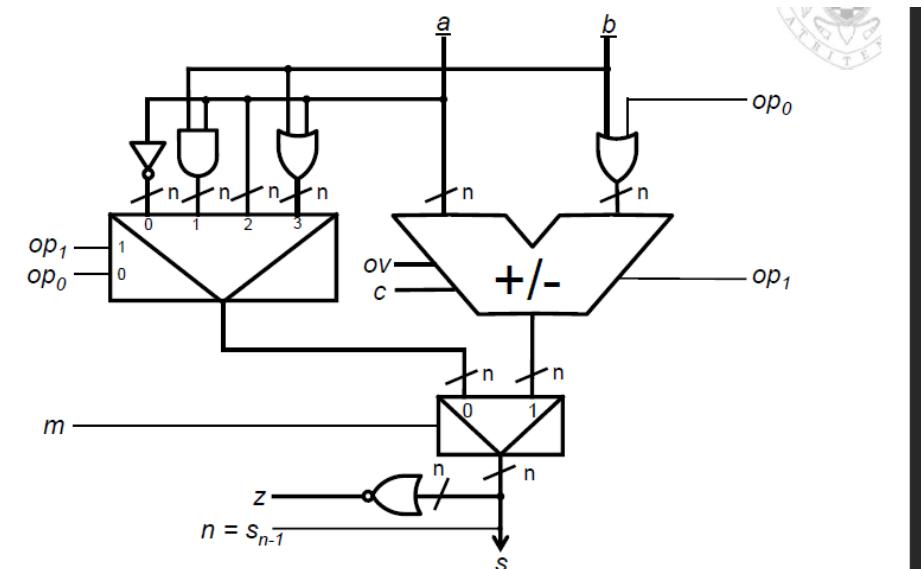
m	op ₁	op ₀	z
0	0	0	not(a)
0	0	1	and(a, b)
0	1	0	<u>a</u>
0	1	1	or(a, b)

operaciones aritméticas

m	op ₁	op ₀	z
1	0	0	<u>a + b</u>
1	0	1	<u>a - 1</u> = <u>a</u> + (-1) = _{C2} <u>a + 1</u>
1	1	0	<u>a - b</u>
1	1	1	<u>a + 1</u> = <u>a</u> - (-1) = _{C2} <u>a - 1</u>

El funcionamiento interno se puede interpretar de distintas formas:

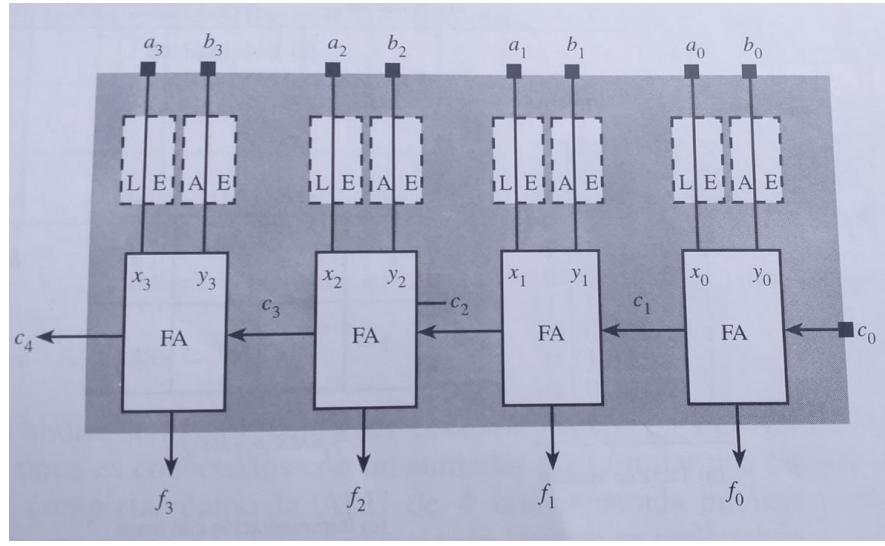
Módulos separados



De esta forma vemos que la parte del sumador de la derecha corresponde a las operaciones aritméticas y la parte del multiplexor de la izquierda corresponde a las operaciones lógicas. Las entradas de selección *op* determinan el tipo de operación que realiza cada una de las dos partes y finalmente la entrada *m* determina cuál de las dos operaciones, la aritmética o la lógica, se muestran finalmente. Es decir, en este caso se evalúan ambas formas y al final es cuando se decide cuál mostrar.

Módulos adicionales a un sumador total

En este caso, lo que se hace es añadir dos módulos conocidos como ampliadores lógicos (*LE*) y aritméticos (*AE*) a un sumador total. De esta forma, si se emplea la función aritmética, ya hemos estudiado como debería funcionar un sumador/restador, pero si se emplea la función lógica, entonces los sumadores no son más que meras conexiones de retraso fijo, es decir, no suman nada.



Podemos ver en la tabla de verdad de abajo que el funcionamiento descrito en el módulo *AE* es completamente análogo a como funcionaría en el ejemplo anterior, siendo la foto de abajo el funcionamiento de cada uno para un bit concreto.

<i>M</i>	<i>S₁</i>	<i>S₀</i>	NOMBRE DE LA FUNCIÓN			<i>F</i>	<i>X</i>	<i>Y</i>	<i>c₀</i>
1	0	0	Decremento			$A - 1$	A	Todo unos (1)	0
1	0	1	Suma			$A + B$	A	B	0
1	1	0	Resta			$A + B' + 1$	A	B'	1
1	1	1	Incremento			$A + 1$	A	Todo ceros (0)	1

(a) Tabla funcional

<i>M</i>	<i>S₁</i>	<i>S₀</i>	<i>b_i</i>	<i>y_i</i>
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(b) Tabla de verdad

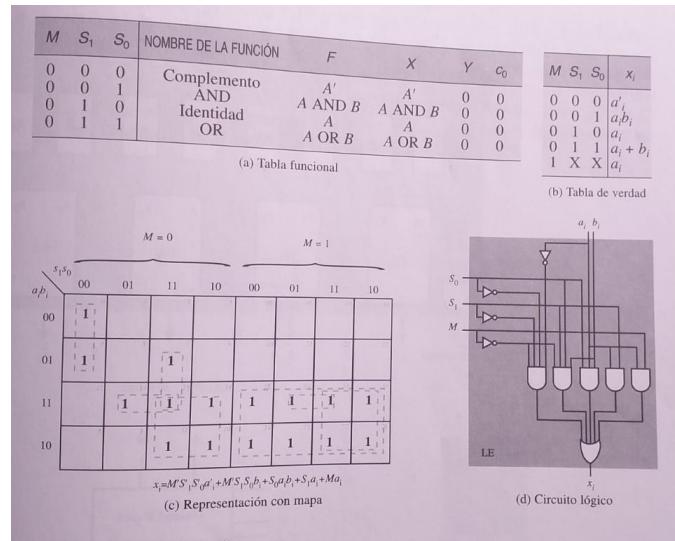
<i>b_i</i>	00	01	11	10
0	1	-	-	1
1	-	1	1	-

$y_i = M S'_1 b_i + M S'_0 b'_i$

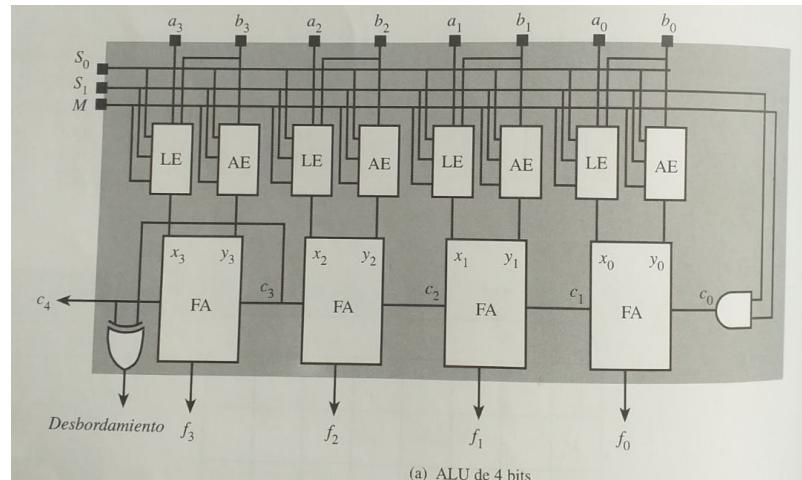
(c) Representación con mapa

(d) Esquema lógico

Para la parte lógica podemos observar que la parte de *Y* es siempre 0 en la tabla de verdad. Esta es la diferencia fundamental con respecto al ejemplo anterior de tipo de ALU puesto que en este caso **toda la condición se pasa como vector a las entradas *a***. Con lo cual, es fácil ver que en este caso el sumador no suma nada, porque suma 0 con algo, y que en consecuencia la salida es directamente la condición evaluada.



La ALU resultante según este tipo quedaría de la siguiente forma:

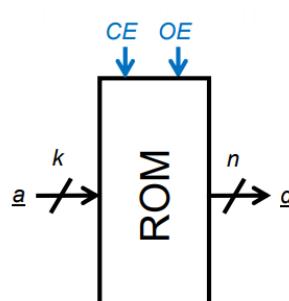


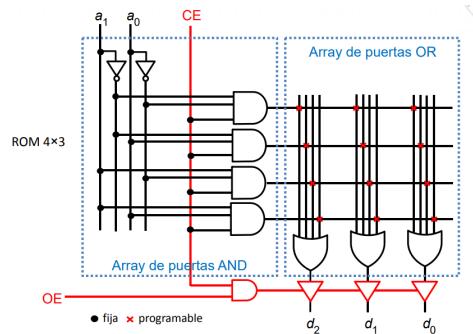
ROM

La ROM (*Read Only Memory*) es un módulo combinacional formado por los siguientes elementos:

- Entradas de dirección (x)
- Salidas de datos (z)
- Entrada de capacitación (CE)
- Entrada de capacitación de lectura (OE)

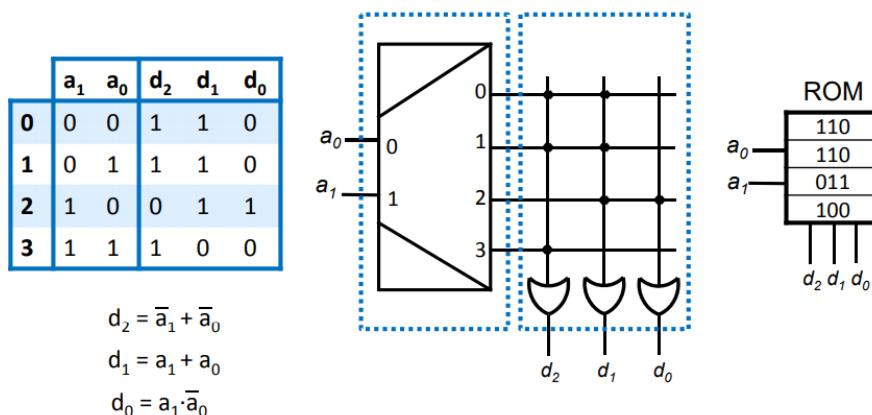
La entrada de capacitación es su *enable* personal y determina el funcionamiento o no del módulo, la entrada de capacitación de lectura es necesaria para implementar ROMs de más tamaño y sirve para pasar un valor que hará de enable en el resto de ROM que formen una más grande.





A diferencia de los módulos anteriores, este módulo no posee una fórmula específica para la descripción de los valores de salida. Se trata de un dispositivo que al igual que el decodificador, implementa todos los minterms en la matriz AND y a cada uno le asigna una dirección de entrada, posteriormente y en función de la configuración interna, devuelve los valores que están guardados en cada dirección, por este motivo decimos que es una memoria, porque en el fondo guardamos en cada dirección un array de bits que posteriormente se reproducen en las salidas de datos al haber introducido una dirección dada.

- Implementar directamente FC almacenando su tabla de verdad.



Es decir, observamos, sobre todo tras hacer la representación con un decodificador¹⁹, que las entradas a indican el minterm o dirección en el cuál mirar y las salidas d devuelven los valores de ese minterm para cada salida de la FC o podemos llamarlos los “datos” guardados en esa dirección.

Tenemos también varias modificaciones del concepto de ROM para atribuirle distintas funcionalidades:

■ Mask Programmable ROM

- Se programa durante la fabricación del chip.
- No puede borrarse/reprogramarse.

■ PROM (Programmable ROM)

- Se programa eléctricamente usando un programador.
- No puede borrarse/reprogramarse.

■ EPROM (Erasable Programmable ROM)

¹⁹Por cuestión de claridad, al haber muchas entradas para cada puerta OR el dibujo puede complicarse así que en vez de poner todas las líneas se pone una única que representa a todas y se marcan en los que puede valer 1

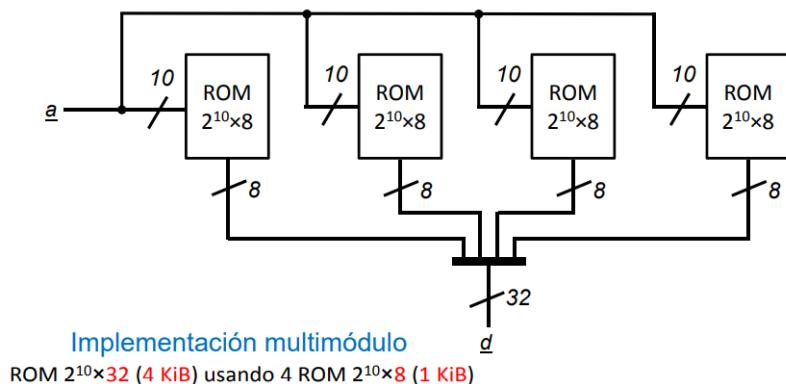
- Se programa eléctricamente usando un programador.
 - Se borra (chip completo) exponiéndola a luz ultravioleta.
- **EEPROM (Electrically Erasable Programmable ROM)**
- Se programa/borra (palabra) eléctricamente usando un programador.
- **Flash memory**
- Se programa/borra (bloque) eléctricamente sin requerir programador.

Aplicaciones de Diseño

A parte de que podemos implementar (según la imagen introducida más arriba) n FC de k variables cada una, podemos combinar varias de ellas para conseguir dos funcionalidades:

■ **Anchura de palabra:**

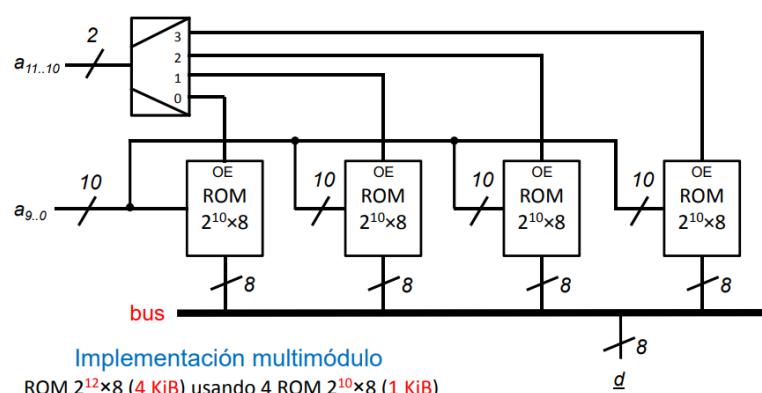
Esto nos permite tener más valores guardados en cada dirección para el mismo número de direcciones.



Es decir, para crear dicha combinación de módulos lo que hacemos es que las entradas de dirección se pasan a todas las ROM, el enable de la 1^a se pasa a las demás y las salidas de cada una se juntan todas en un único vector de salida ORDENADO que se hace usando el BUS de abajo.

■ **Profundidad:**

Esto nos permite tener más direcciones posibles pero con el mismo número de valores que se pueden guardar en cada una.

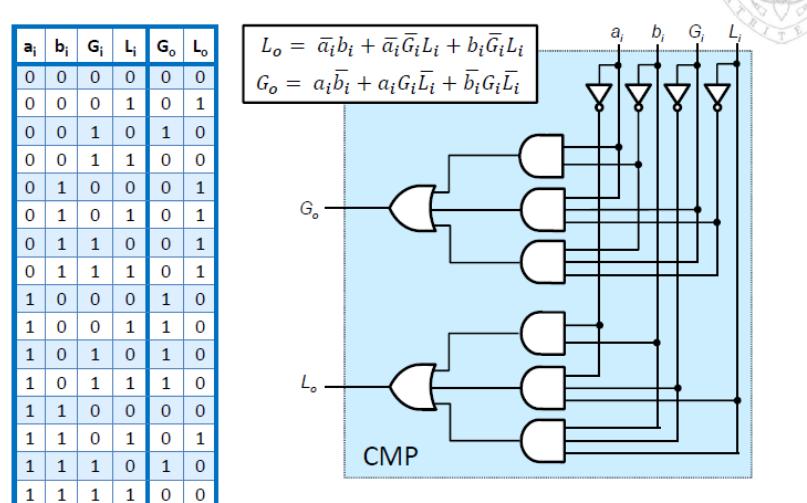
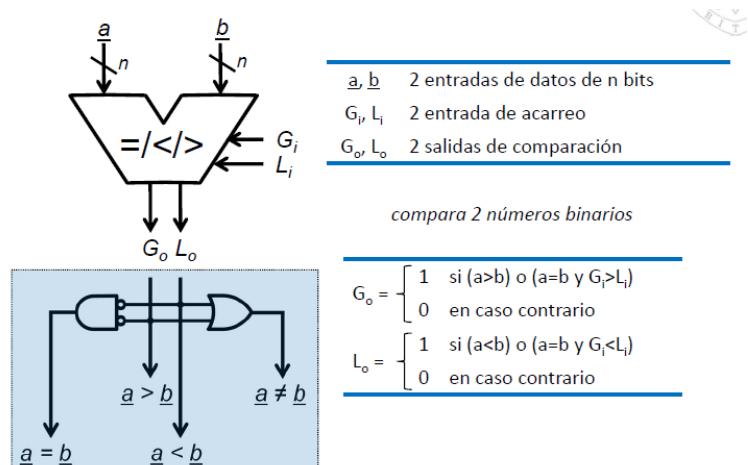


En el fondo es más útil pensar como si se dispusiesen en vertical para ver que son más entradas de dirección y la misma longitud de palabra. El decodificador hace que se distingan las ROM que hay que utilizar en función de los dos valores de más peso siendo las direcciones que comienzan por 00 las de la primera ROM y 11 las de la última. Cuando la combinación de los valores más significativos es ese, la entrada de capacitación de lectura se habilita SOLO EN UNA DE LAS ROM y por lo tanto hemos indicado que solo vamos a buscar en ese módulo. Lo demás es igual que la ROM aislada, una vez que tenemos seleccionada cuál debe funcionar, el resto del nombre de la dirección se busca en la ROM y como las OE del resto de ROMs están desactivadas no hay ningún problema con respecto a cortocircuitar las salidas con el BUS de la imagen puesto que solo hay corriente por las salidas de datos de la ROM en la que buscábamos y de este modo se devuelven los valores guardados en la dirección deseada.

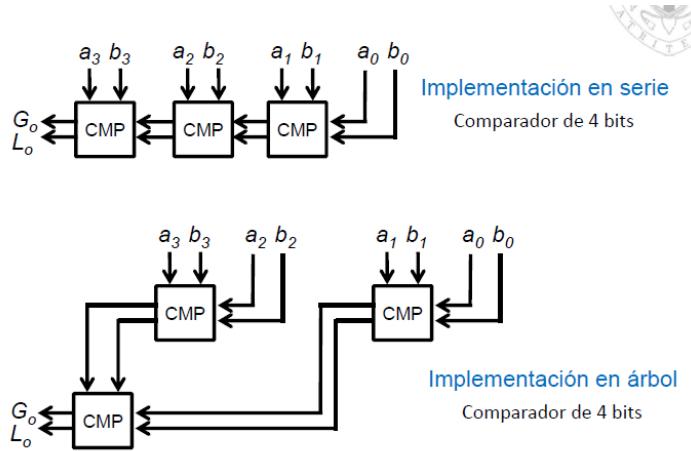
Comparador de magnitud

Se usa para evaluar los operadores lógicos $<$, $=$ y $>$. Para ello este módulo se compone de las siguientes partes:

- 2 entradas de n datos
- 2 entradas de acarreo
- 2 salidas de comparación



Los valores de salidas dependen de dos posibilidades, valdrán 1 automáticamente si un bit es menor que el otro o viceversa o también valdrán 1 en caso de que siendo iguales las entradas de acarreo reflejen que los de antes no lo eran. Viendo este funcionamiento en función de los valores anteriores, podemos ver de forma intuitiva que si se combinan de la misma forma que lo hacían los sumadores podemos componer comparadores de palabras de más bits.



Lo que podemos ver en el funcionamiento interno es que la comparación de dos números enteros positivos²⁰ se reduce a la comparación exclusiva de dos bits, comenzando por los menos significativos y yendo hacia los que más.

Aspectos tecnológicos

Para calcular el gasto de los módulos combinacionales en lo que a retardo, precio y área se refiere, procedemos de forma igual que en las redes de puertas, pero en este caso con la correspondiente tabla:

Módulo	Área (μm^2)	Retardo (ps)	Consumo estático (nW)	Consumo dinámico (nW/MHz)
	11.0592	223	84	8639
	23.0400	250	163	15169
	29.4912	191 (z_0) 189 (z_1) 132 (z_2) 127 (z_3)	23	543
	29.4912	205 (s) 226 (c)	159	5374 (s) 713 (c)

Fuente: Synopsys (SAED EDK 90 nm)

²⁰Nótese que dicha descripción es sólo válida para números positivos

ESPECIFICACIÓN DE SISTEMAS SECUENCIALES

ESPECIFICACIÓN DE ALTO NIVEL

La diferencia fundamental que presenta este tipo de sistemas con respecto a los combinacionales es que si para estos últimos una misma entrada tenía un única posible salida, para los secuenciales una misma entrada puede tener distintas salidas en función de cual sea la secuencia de entradas anteriores. Es decir, si para la entrada 1, la salida vale 0, en los sistemas secuenciales la salida puede pasar a valer 1 pasado un cierto tiempo cuando en los sistemas combinacionales la salida se mantiene invariante.

Veamos un ejemplo:

Supongamos que tenemos una máquina que indica si el número de 1 que ha recibido a lo largo del tiempo es par o impar.

$$\begin{array}{l} A = \begin{matrix} 0 & 0 & 1 & 0 & 1 \end{matrix} \\ B = \begin{matrix} 1 & 0 & 1 & 1 & 0 \end{matrix} \\ C = \begin{matrix} 0 & 1 & 1 & 0 & 1 \end{matrix} \\ D = \begin{matrix} 1 & 1 & 1 & 0 & 1 \end{matrix} \end{array}$$

Observamos claramente que para ambas secuencias de entradas A y D tenemos que la salida deberá indicar el mismo valor, al igual que ocurre con las secuencias B y C entre ellas. De este modo, vemos que el sistema en el caso de la secuencia A o D solo puede variar de una forma concreta:

$$\begin{cases} x(t) = 0 \rightarrow z(t) = 1 \\ x(t) = 1 \rightarrow z(t) = 0 \end{cases}$$

Es decir, decimos que si la secuencia que tenemos es de un número de 1 PAR, entonces si recibe otro 1 cambiará a una de tipo PAR y la salida valdrá 0 y si recibe un 0 seguirá siendo PAR y la salida seguirá siendo 1.

Es evidente que cualquier secuencia de entrada se puede agrupar en dos clases distintas, una secuencia con un número par de 1 o con un número impar. De esta dicotomía surge el concepto denominado **ESTADO**, que no es más que el conjunto de las secuencias que causan el mismo efecto y que determina la situación del sistema en función de la secuencia de entradas previas. De este modo, conociendo el estado en el que me encuentro soy capaz de determinar en función de mi entrada, cual será mi salida y mi estado futuros.

Una vez visto el concepto de sistema secuencial y las peculiaridades que presenta es notable resaltar que hay dos posibles formas de afrontar la especificación de estos sistemas. Podemos hacer que **la salida dependa directamente del estado en el que me encuentro** o que **la salida dependa del estado y entrada anteriores**. De estos dos enfoques nacen las **máquinas de Mealy** y **de Moore**.

Máquinas de Mealy y de Moore

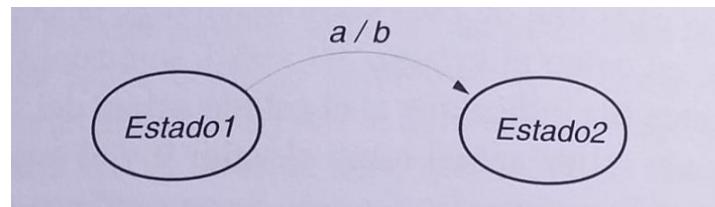
Es notable destacar antes de entrar en la descripción de ambos modelos que cualquier sistema secuencial se puede especificar de cualquiera de las dos formas y que la existencia de ambas es para la optimización de los diseños creados, cuya evaluación se deja a cuenta del diseñador.

Máquinas Mealy

Las máquinas de Mealy se decantan por la opción 2, es decir, en este tipo de sistemas secuenciales la salida es una función de la entrada y estado actuales, con lo cual siempre la salida refleja el valor en función de la entrada actual y **NO va con retraso**.

$$\begin{cases} z(t) = H(x(t), s(t)) & \text{donde } H \text{ es la función que define la salida} \\ s(t+1) = G(s(t), x(t)) & \text{donde } G \text{ es la función que define el estado} \end{cases}$$

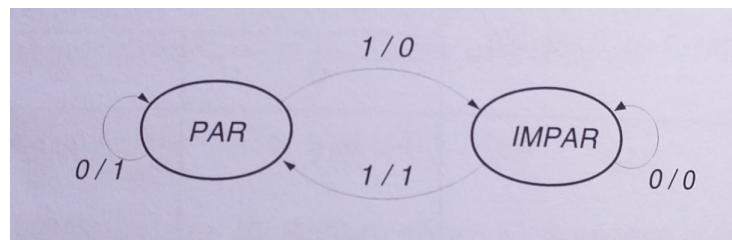
Para la descripción de estas máquinas es útil utilizar los conocidos como **diagramas de estados**:



En este tipo de diagramas se indican los cambios con la siguiente notación:

- Cada estado se escribe dentro de un círculo
- Se trazan tantas flechas como posibles entradas haya
- Cada flecha se dirige al siguiente estado según la entrada recibida
- En cada flecha se escribe de numerador la entrada que provoca el cambio y de denominador la salida que refleja

Ej.:



Máquinas de Moore

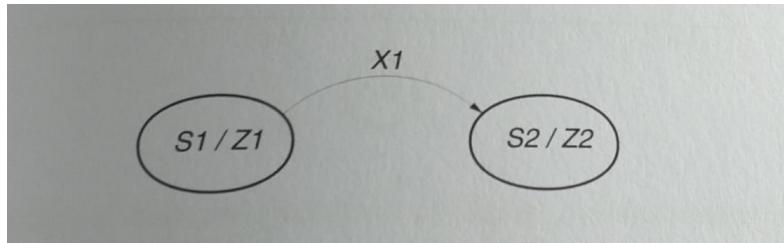
Las máquinas de Moore se decantan por la opción 1; la salida depende exclusivamente del estado actual. Con lo cual, las funciones que definen el estado y la salida en un ciclo de reloj t nos quedan como:

$$\begin{cases} z(t) = H(s(t)) & \text{donde } H \text{ es la función que define la salida} \\ s(t+1) = G(s(t), x(t)) & \text{donde } G \text{ es la función que define el estado} \end{cases}$$

Con lo cual, podemos ver que la salida que vemos en un instante t es consecuencia de la entrada del instante $t - 1$, por lo que nuestro sistema da salidas con un retraso de un ciclo de reloj²¹ con respecto a las máquinas de Mealy.

En este caso la representación mediante diagramas de cambio de estado también es útil, eso sí, cambiando debidamente la notación correspondiente:

- Se escribe la salida como denominador del estado dentro del círculo, pues depende del mismo.
- Las flechas siguen el mismo funcionamiento pero ahora solo se indica la entrada que provoca el cambio de estado.



Simplificación de estados

En general, puede ocurrir que al definir el comportamiento de un sistema secuencial, tengamos en cuenta más estados de los que realmente son necesarios, lo que encarece el circuito y contribuye al retardo de las señales. Aunque existen procedimientos sofisticados de computación para poder hacer esto, una forma más sencilla pero prácticamente igual de efectiva es tener en cuenta la siguiente norma, sean s_1 y s_2 dos estados del sistema:

$$\begin{aligned} s_1 \equiv s_2 &\Leftrightarrow H(s_1, x) = H(s_2, x) \\ s_1 \equiv s_2 &\Leftrightarrow G(s_1, x) = G(s_2, x) \end{aligned}$$

Es decir, dos estados **son equivalentes** si poseen la misma salida para la misma combinación de entrada y confluyen al mismo siguiente estado para la misma combinación de entrada. Si se dan estas dos condiciones, entonces podemos escribir dicho estado como el mismo, simplificando la síntesis de nuestro sistema secuencial.

Especificación binaria

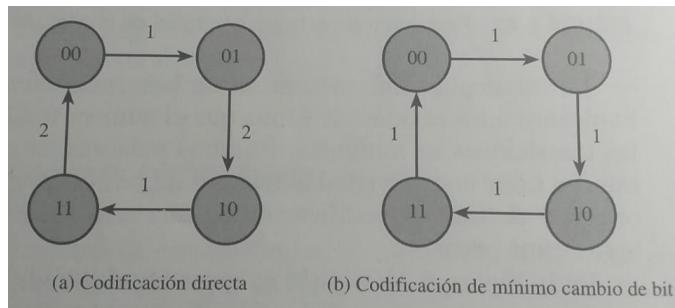
Para la codificación binaria de estos sistemas no hay más que escribir las tablas de verdad que hacíamos en temas anteriores para cada función de conmutación y tras haberlo definido todo, dar una codificación binaria a cada valor posible. El cálculo del resto de cosas, como la simplificación de las FC, los retardos, etc. es análogo a lo anterior y si cupiese alguna duda se recomienda mirar capítulos anteriores, el ejemplo del tema posterior de implementación o el propio libro de Fundamentos de Computadores que se posea.

Codificación eficiente

Como se verá posteriormente, una codificación u otra de las entradas, salidas, estados... repercutirá en el número de puertas, biestables, entradas... y, por lo tanto, en el diseño, su gasto y su retardo. Por ello, es primordial tener en cuenta algún método que permita tener una implementación óptima en cuanto a la codificación se refiere, para ello, basta con seguir estas reglas:

²¹Lo cual es muy significativo al combinar estas máquinas con otras que sean sincronas

- Hacer el diagrama de estados correspondiente y asignar a cada arco de cambio de estado el número de bits que difieren en la codificación conferida a los estados que une.
 - Codificar los estados de manera que la suma de los números asignados a cada estado sea la mínima posible

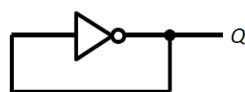


IMPLEMENTACIÓN DE SISTEMAS SECUENCIALES

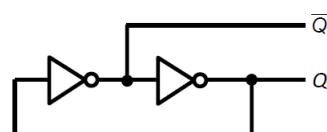
Para poder entender la implementación de este tipo de sistemas es necesario comprender las nociones de **biestable** y **registro de estados**.

Biestable

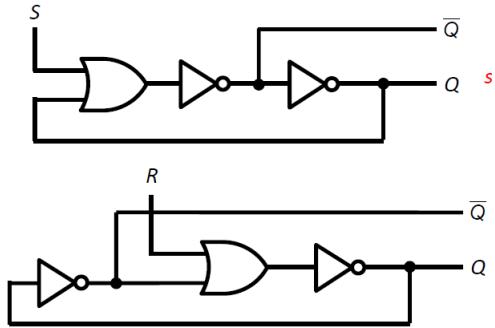
Un biestable es un dispositivo capaz de mantener inalterable un bit a lo largo del tiempo. Sabiendo esto es intuitivo el entender la evolución del mismo hasta el biestable final que mostraremos:



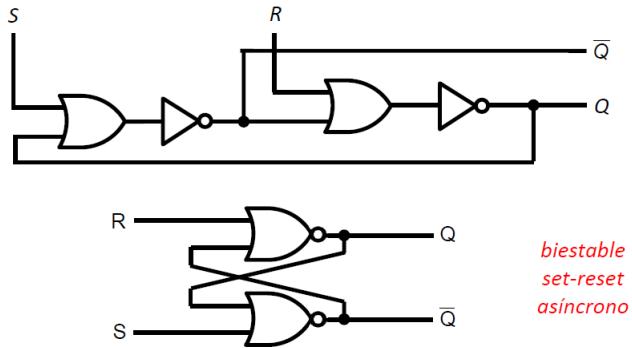
Como vemos, realimentando un inversor podemos mantener siempre un bit de información, el único problema es que este valor almacenado oscila entre 0 y 1 alternativamente, para ello podemos modificar el diseño de la siguiente manera:



De este modo, hemos conseguido que el circuito almacene siempre el mismo valor, pero como no tiene ninguna entrada, no podemos determinar cuál es ese valor almacenado, surgiendo el siguiente diseño:

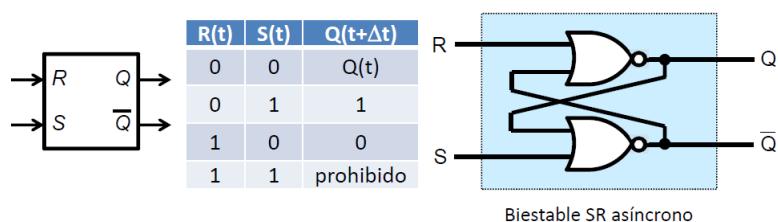


Con estos dos diseños, observamos que cada uno solo puede almacenar un 0 (el de entrada R) o un 1 (el de entrada S), por lo que combinando ambos ya podemos elegir que bit almacenar:

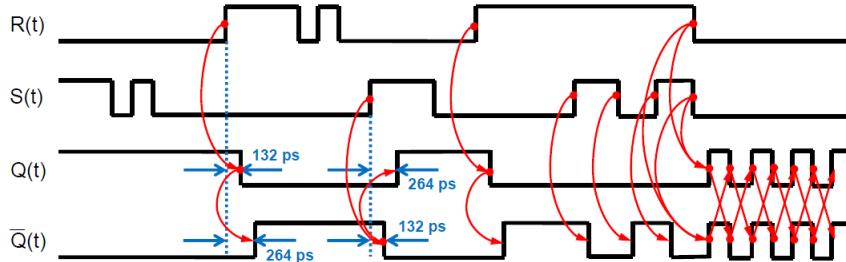
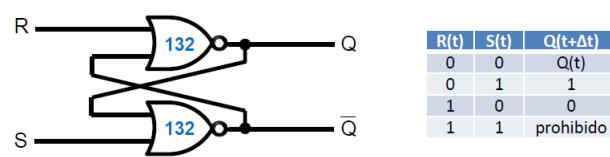


Biestable RS asíncrono

Con todo lo visto en la evolución previa llegamos precisamente al **biestable RS asíncrono**. Este dispositivo es capaz de almacenar la información de un bit de manera asíncrona.



Gracias al siguiente cronograma, podemos observar las siguientes características:



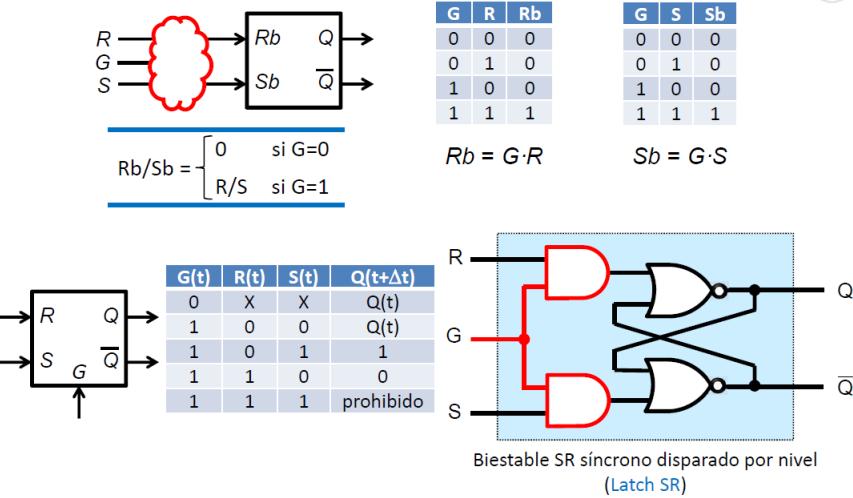
- La salida del biestable por convenio será denominada Q , siendo \bar{Q} la salida cuyo valor es el complementario de Q .
- Cuando la entrada R vale 1, el bit de salida Q vale 0, por ello se la denomina **Reset**.
- Cuando la entrada S vale 1, el bit de salida Q vale 1, por ello se la llama **Set**.
- Cuando ambas entradas valen 0, entonces la salida mantiene constante el valor que tuviera en el instante previo.
- Observamos que para el valor de ambas entradas a 1, se toma la configuración de la entrada que hubiese estado a 1 en el momento previo a esa subida a 1 de ambas, por lo que si se viene de tener ambas a 0 puede generar problemas, ya que la salida sería el resultado de la señal más rápida de ambas puestas NOR, con lo cual se le denomina **configuración prohibida**

Es notable destacar que utilizando los conocimientos adquiridos en temas anteriores, se puede transformar este biestable implementado por puertas NOR a otros implementados por puertas NAND, donde los estados *prohibido* y *conservación del bit* intercambian sus configuraciones pero el funcionamiento lógico es el mismo.

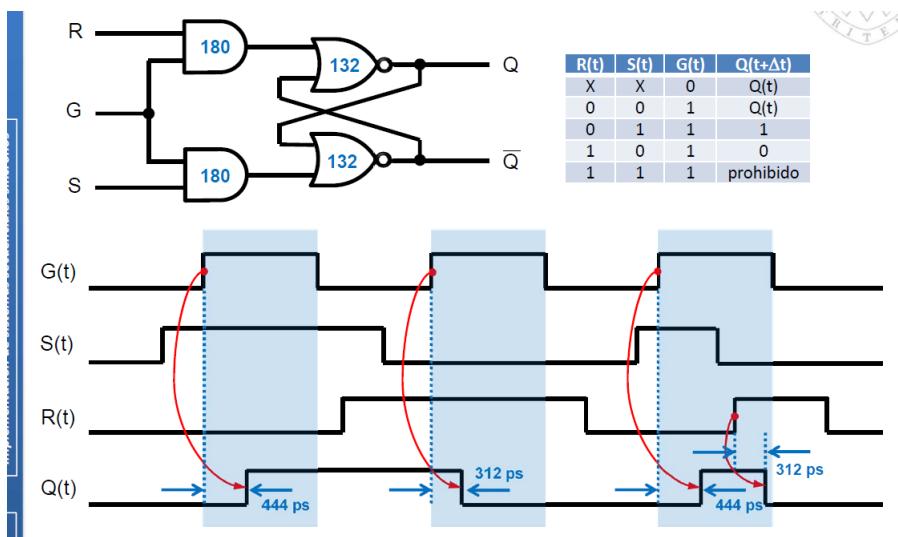
Biestable RS por nivel

Si queremos transformar ese biestable asíncrono en uno síncrono, el primer paso puede ser crear una señal que haga de enable y que solo permita el funcionamiento del módulo cuando se encuentra activa, esto es lo que fundamenta el uso del **biestable activo por nivel**.

Biestable SR síncrono (por nivel)



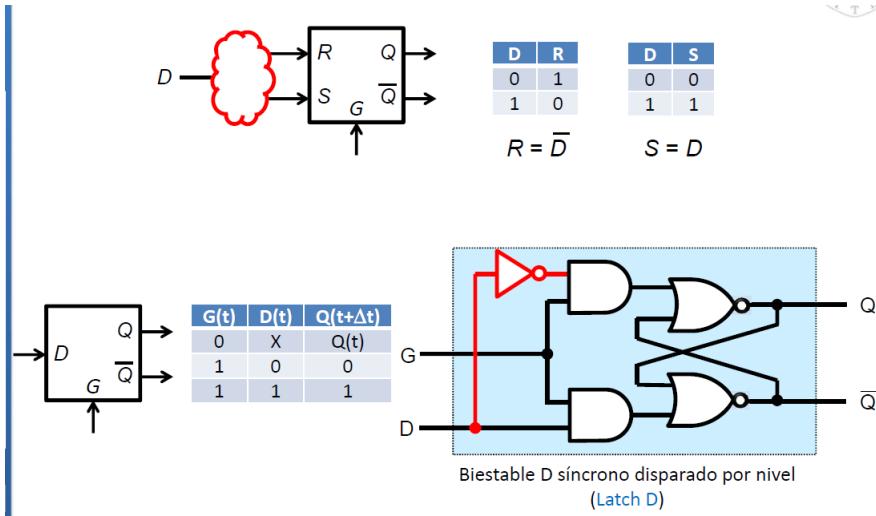
Entre sus características diferenciadoras del biestable anterior podemos observar que:



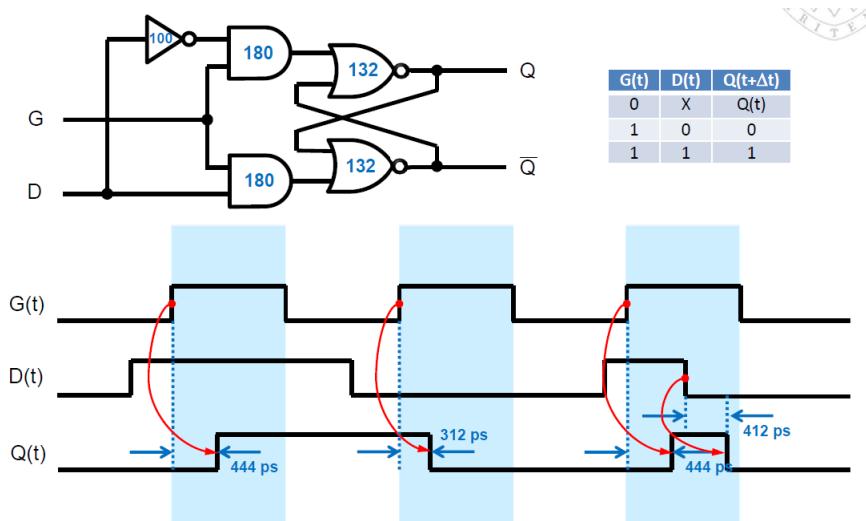
- La salida Q solo puede cambiar cuando la entrada G está activa.
- A pesar de que discretiza en cierta medida los momentos de cambio, su principal problema es que con la entrada G activa, cualquier cambio en las entradas produce un cambio en la salida, es decir, sigue siendo asíncrono en los momentos de actividad de G lo que suele ser fuente de errores.

Biestables D síncronos

Los biestables RS son útiles para elegir almacenar un 0 o un 1 durante un cierto ciclo de reloj, pero para poder almacenar el bit transmitido por otra señal al biestable, es necesario implementar un nuevo diseño que no hace más que distribuir la señal a R o a S para que la información guardada sea un 0 o un 1.



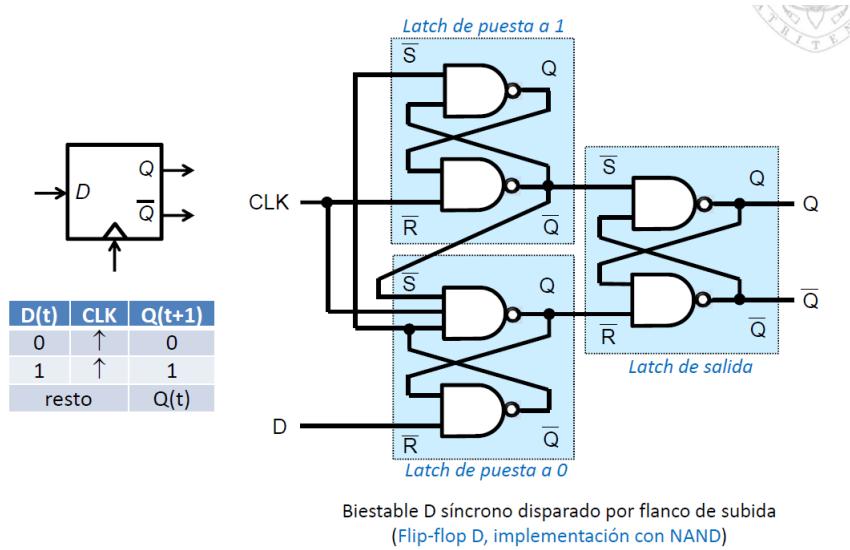
Este dispositivo, a pesar de poder guardar la información de una señal durante un ciclo de reloj, presenta inconvenientes similares a los biestables descritos anteriormente:



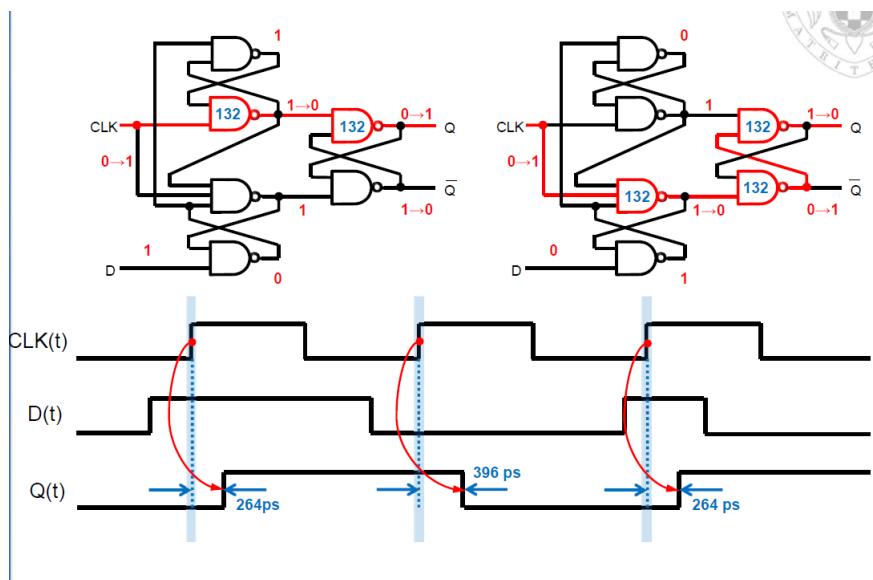
A pesar de que discretiza en cierta medida los momentos de cambio, su principal problema es que con la entrada G activa, cualquier cambio en las entradas produce un cambio en la salida, es decir, sigue siendo asíncrono en los momentos de actividad de G lo que suele ser fuente de errores.

Biestables D por flanco

Para solucionar este problema de discretización de los cambios surgen los **biestables D síncronos disparados por flanco**. Este tipo de dispositivos están preparados para “leer” su entrada únicamente cuando la señal de reloj se sube de 0 a 1 (flanco de subida) o se baja de 1 a 0 (flanco de bajada), pero nunca cuando la señal de reloj está estable en 0 o 1.



Como vemos²², este último ejemplo es el biestable que se quería expresar al principio del capítulo y que aglutina el conjunto de ventajas y utilidades de los anteriores, pero que prescinde de los inconvenientes que se han ido explicando a lo largo del desarrollo. Vemos aquí el cronograma final en el que se observa que se ha conseguido una discretización de los cambios de salida **casi completa**, puesto que idealmente es en el flanco cuando se produce el cambio pero todos los circuitos a nivel físico poseen siempre un retardo: el objetivo es que sea el mínimo posible y se ha de tener en cuenta el mismo para que la implementación no admita posibles errores en su funcionamiento.



Registros

Podemos definir, al menos por el momento, un **registro** como una asociación de biestables con una señal de reloj común; de modo que en vez de guardar un único bit se puede almacenar un vector de n bits, siendo n el número de biestables que componen el registro. Posteriormente, se presentarán otros tipos de registros con unas características especiales más orientados a la explicación de cómo es la arquitectura de un ordenador y para gestionar el control de rutas de datos.

²²Se conseguiría una implementación por flanco de bajada cambiando todas las puertas NAND por puertas NOR

Implementación canónica de sistemas secuenciales

Los sistemas secuenciales se fundamentan prácticamente en tres partes:

- **Módulo combinacional de la función de estado**
- **Módulo combinacional de la función de salida**
- **Registro de estado**

Es decir, la síntesis de un sistema secuencial se fundamenta en la lógica combinacional vista en temas anteriores, con la única excepción de que en este caso los biestables juegan un papel fundamental en la gestión de los datos de entrada de cada módulo combinacional comentado, por lo que **lo único que hay que tener en cuenta es como se conectan los biestables puesto que todo lo demás es como si se tratase de sistemas combinacionales.**

Síntesis con biestables D

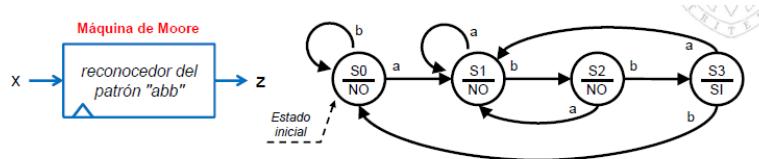
Es importante que los biestables del circuito, que es la única parte distinta, sigan las siguientes normas:

- Todos están conectados a una señal de reloj periódica
- Todos se disparan por flancos de la misma polaridad
- Toda las realimentaciones incluyen al menos un biestable

Entonces el funcionamiento de forma cronológica que debería seguir el circuito quedaría de la siguiente manera:

1. Los cálculos que realiza el sistema se realizan **ciclo a ciclo**.
2. Las fronteras del ciclo están marcadas por las transiciones de igual polaridad en el reloj común.
3. **Al comienzo del ciclo**, el sistema hace un cambio de estado mediante la **actualización simultánea de todos los biestables**.
4. El nuevo estado provoca transiciones en las entradas de los módulos combinacionales que a su vez provocarán transiciones en sus salidas.
5. El **cálculo** a realizar en el ciclo **finaliza** cuando **TODOS** los sistemas combinacionales han alcanzado su **régimen permanente**.
6. Los valores permanentes a la salida de los módulos combinacionales son utilizados para actualizar los biestables al comienzo del ciclo siguiente.

Por aquí el siguiente ejemplo del diseño completo de un circuito secuencial:



- Codificación domino: { a → 0, b → 1 }
- Codificación codominio: { NO → 0, SI → 1 }
- Codificación estados: { S0 → (00), S1 → (01), S2 → (10), S3 → (11) }

Función de transición de estados

x	q_1	q_0	q_1'	q_0'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

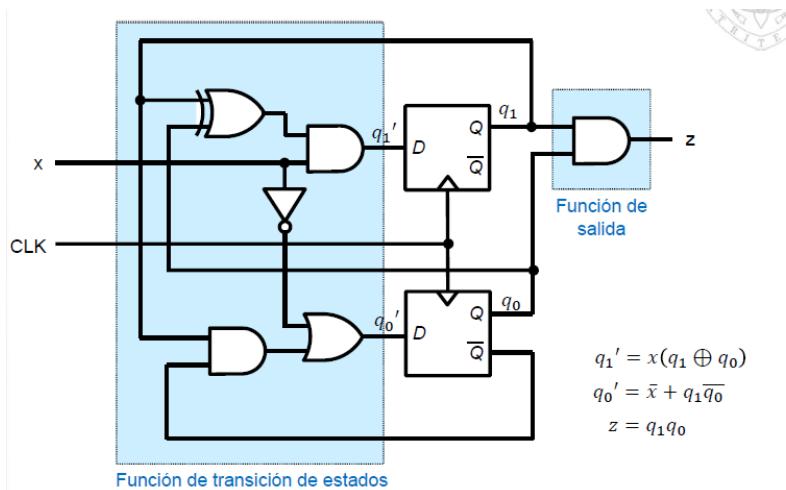
Función de salida

q_1	q_0	z
0	0	0
0	1	0
1	0	0
1	1	1

$$q_1' = x(q_1 \oplus q_0)$$

$$q_0' = \bar{x} + q_1\bar{q}_0$$

$$z = q_1q_0$$



$$q_1' = x(q_1 \oplus q_0)$$

$$q_0' = \bar{x} + q_1\bar{q}_0$$

$$z = q_1q_0$$

- Codificaciones distintas dan lugar a implementaciones diferentes de la misma máquina de estados.
 - Por ello es interesante elegir aquella codificación que reduzca al máximo el coste/retardo de los circuitos de transición y salida.
- Codificación domino: { a → 0, b → 1 }
- Codificación codominio: { NO → 0, SI → 1 }
- Codificación estados: { S0 → (01), S1 → (00), S2 → (10), S3 → (11) }

Función de transición de estados

x	q_1	q_0	q_1'	q_0'
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1

Función de salida

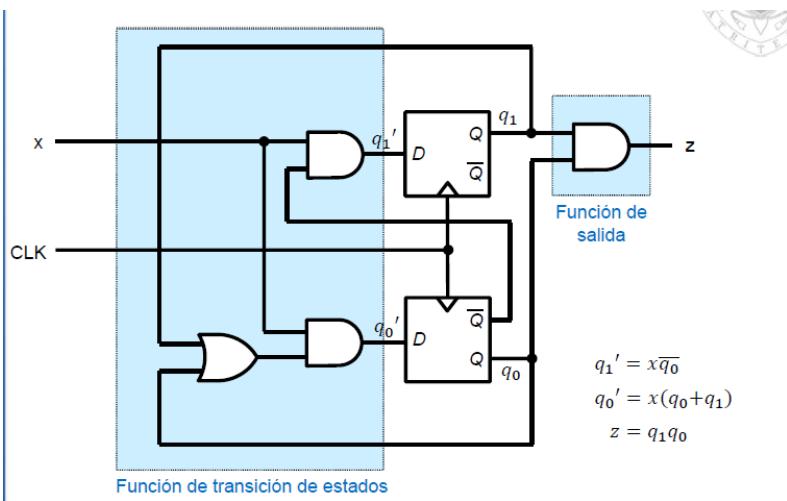
q_1	q_0	z
0	0	0
0	1	0
1	0	0
1	1	1

requiere 2 puertas menos

$$q_1' = x\bar{q}_0$$

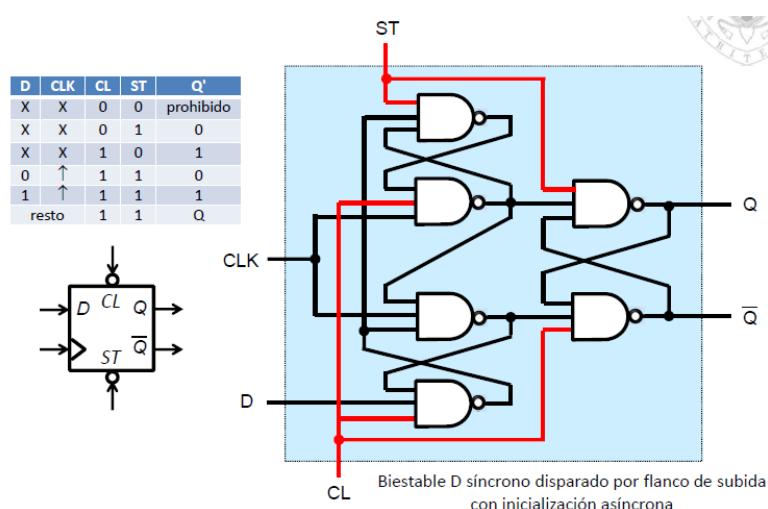
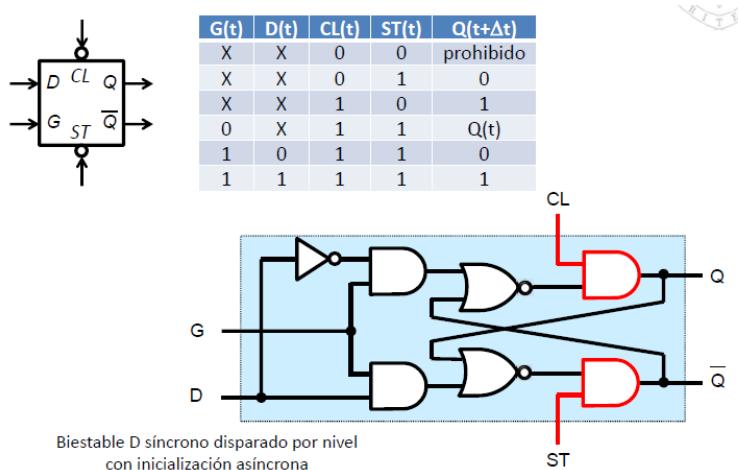
$$q_0' = x(q_0 + q_1)$$

$$z = q_1q_0$$



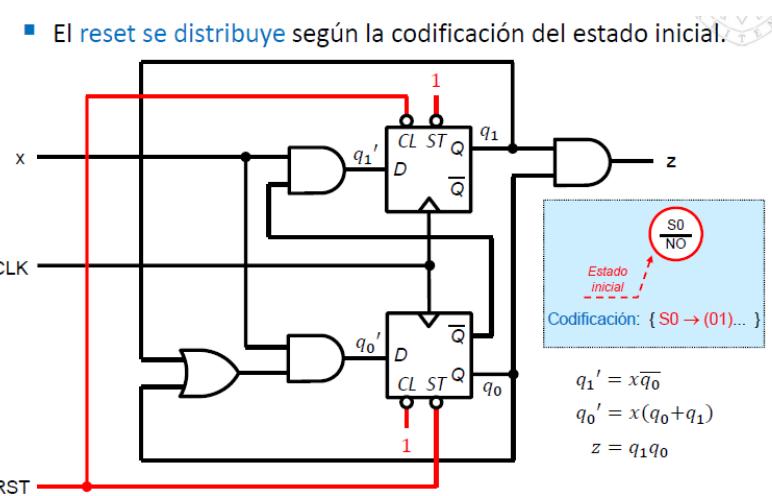
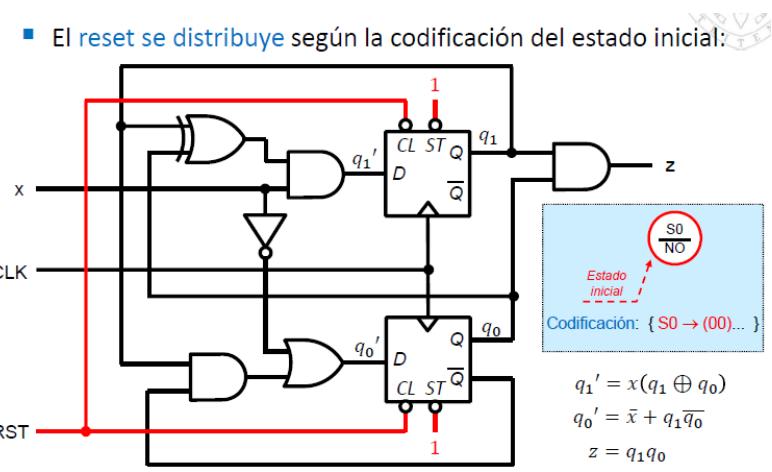
Inicialización de sistemas secuenciales

Hemos visto en el desarrollo de los biestables que poseen ciertos estados prohibidos porque pueden provocar un comportamiento impredecible en el circuito, de ello surge la pregunta de: ¿y si antes de recibir ninguna entrada me encuentro en un estado prohibido? Por este motivo y en primer lugar, en los biestables del circuito se incluyen dos entradas asíncronas nuevas: **preset** y **clear**:



Como vemos, cuando ambas entradas están a 1 el funcionamiento del biestable es el mismo que el descrito hasta ahora, pero cuando *Clear* baja a 0, entonces el biestable pasa a valer 0 y cuando lo hace *Preset* entonces pasa a valer 1, es decir, decimos que ambas entradas son *activas a baja*.

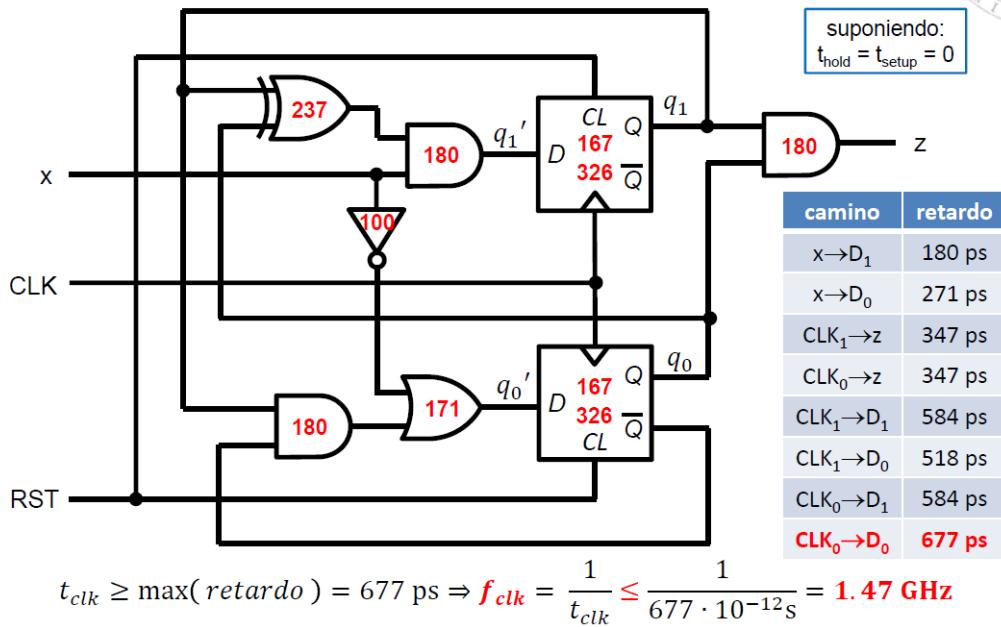
De este modo, en un circuito secuencial se ha de incluir una señal global conocida como **reset** que se encargue de inicializar los biestables al estado inicial definido en la codificación binaria de forma que esté debidamente conectado a las entradas *clear* o *preset* de cada biestable.



Aspectos tecnológicos



Cálculo de la frecuencia máx. de reloj (CMOS 90 nm)



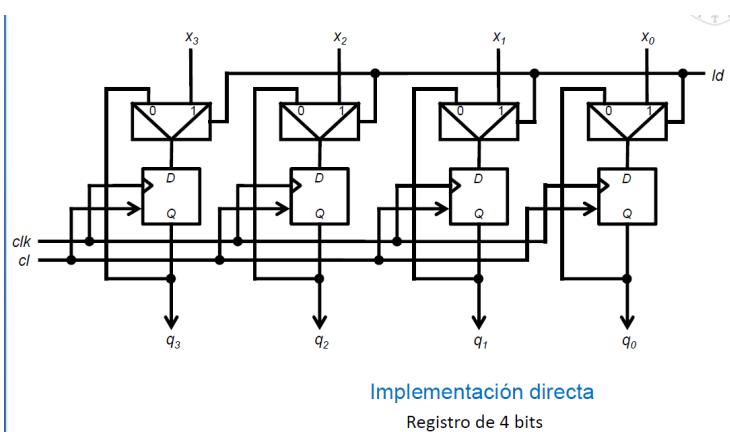
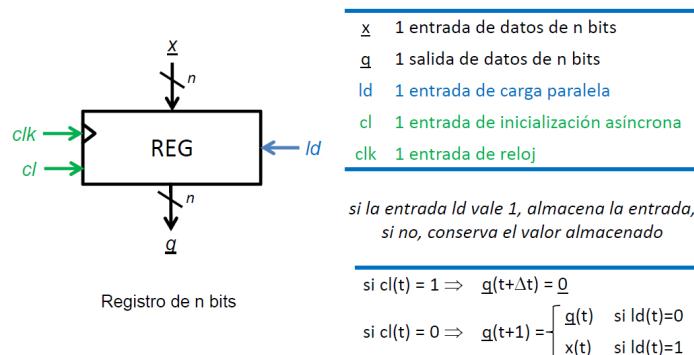
MÓDULOS SECUENCIALES BÁSICOS

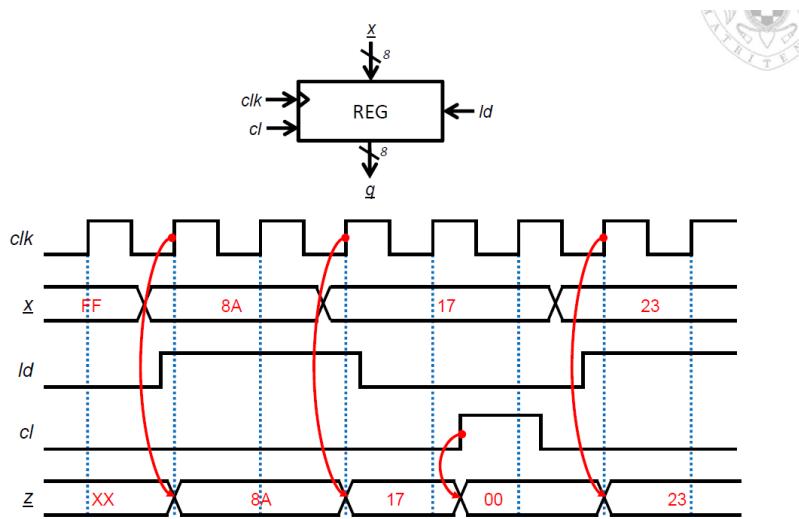
Del mismo modo que ocurría con los sistemas combinacionales, en los secuenciales existen una serie de circuitos que por su diseño y aplicaciones revisten gran importancia y que por ello se han encapsulado como módulos a parte para su uso como una única entidad.

Registro de carga paralela

Como ya comentamos anteriormente, los registros no son más que asociaciones de biestables para guardar más de un bit, en este tema se desarrollarán estos pero con funcionalidades distintas que permiten un aprovechamiento mayor de sus características como componentes de **memoria**.

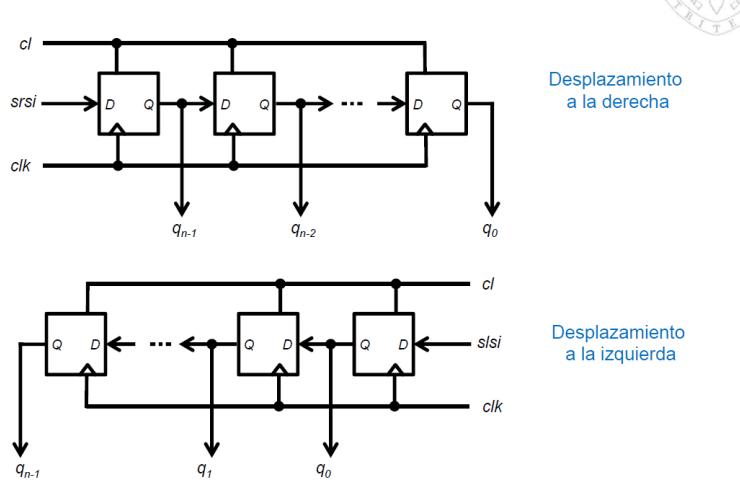
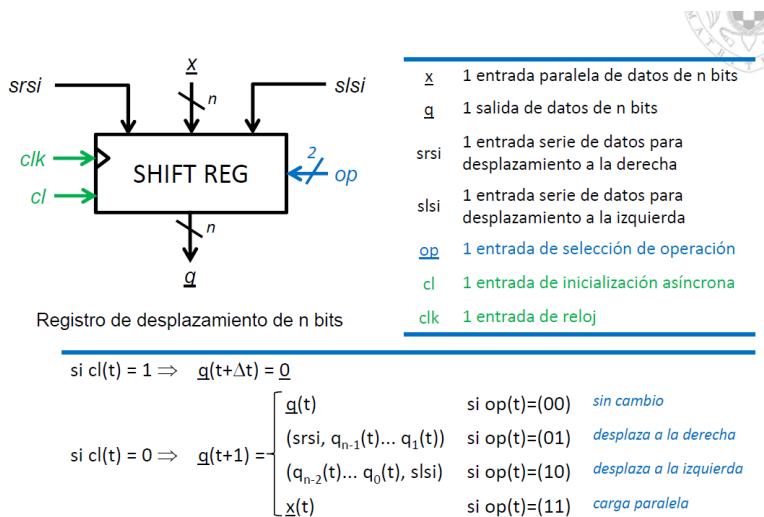
Los registros de carga paralela tienen, con respecto a los registros explicados, una entrada de *load* y otra de *clear* cuyo objetivo es poner el registro al estado inicial (clear) o permitir almacenar o no las entradas que le llegan (load).





Registro de desplazamiento

Este registro implementa las funcionalidades del anterior pero incluyendo la capacidad de introducir un bit por la derecha o por la izquierda al vector de bits que guarda en su interior, eso sí, desplazando los restantes una posición a la derecha y perdiendo la información del último.



En este caso las señales *srsi* y *slsi* seleccionan hacia que dirección se desplaza el bit y una nueva señal²³ *op* selecciona la operación a realizar puesto que puede seguir realizando la carga en paralelo del registro anterior.

Aplicaciones de diseño

Tiene dos aplicaciones principales: convertir series de datos a datos en paralelo e implementar un reconocedor de secuencias.

Datos de serie a paralelos:

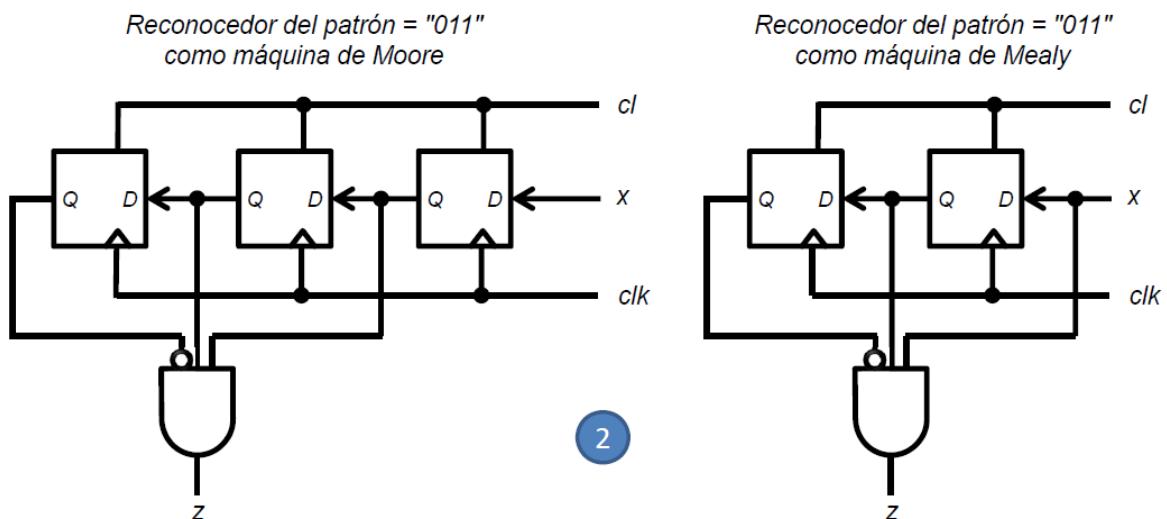
Para realizar esta conversión pensemos en el estado inicial con todo a ceros. Si conectamos una serie de datos a la entrada más a la derecha de las que están en paralelo y seguimos el siguiente bucle²⁴

- Introducimos la entrada (load).
- Desplazamos un bit “0” a la izquierda todo el registro

Entonces al realizarlo tantas veces como datos halla en esa serie o como biestables tenga el registro, tendremos la serie puesta en paralelo dentro del registro y este puede enviarla de esta forma a cualquier otro módulo.

Reconocedor de secuencias:

Una vez entendida la funcionalidad anterior, no es difícil darse cuenta de que la secuencia de datos que llegaban en serie ha quedado almacenada en el interior del registro, por lo que conectando a una puerta AND la salida de cada biestable del registro, si los biestables guardan la secuencia requerida, esta valdrá 1, si no valdrá 0. Por lo que podemos implementar dicha funcionalidad simplemente introduciendo las entradas del reconocedor por la entrada de desplazamiento.



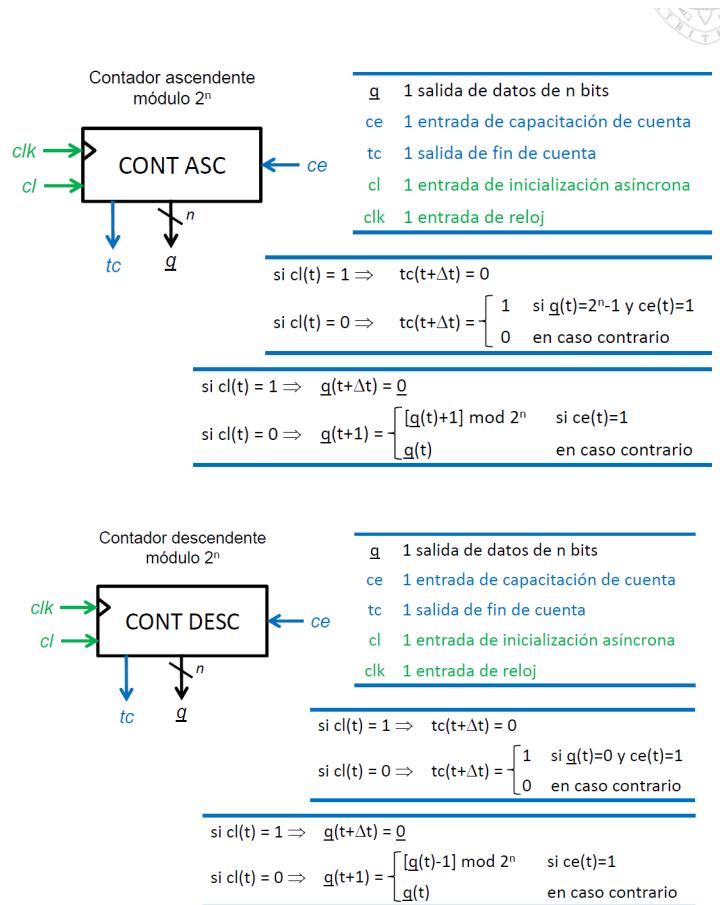
²³Nótese que load pasa a estar incluido en una de las codificaciones de esta señal

²⁴También es posible introducir directamente la serie por el desplazador de bits

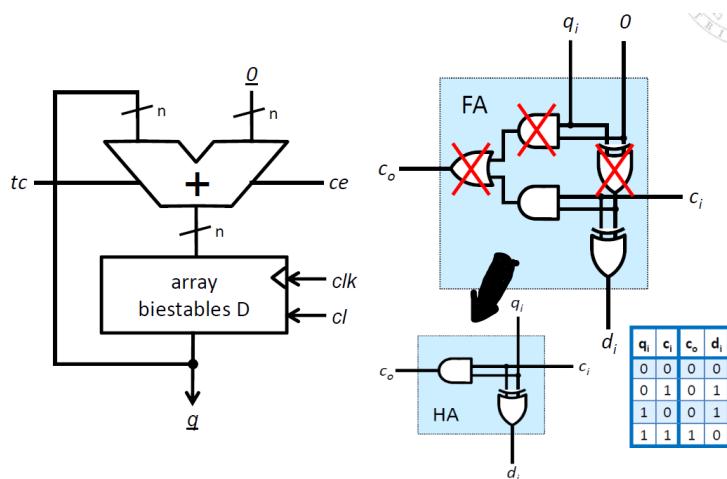
Contadores

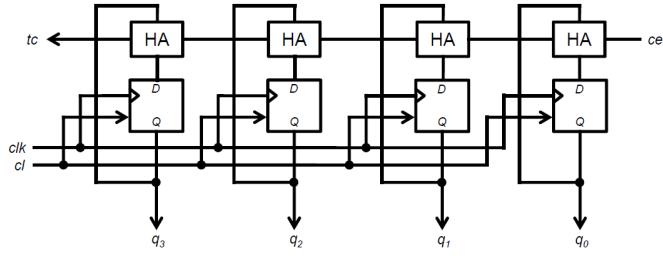
Realizan una cuenta progresiva módulo m , donde este módulo es $2^n - 1$. Es decir, cuentan hasta el valor máximo que le permite el número de bits que poseen y después vuelven a 0.

Como vemos, la señal de ce habilita el incrementar las salidas q en una unidad más, cuando estas llegan al valor $2^n - 1$, entonces al recibir una nueva instrucción de cuenta, las salidas se ponen a 0 todo y tc indica que se ha llegado al tamaño máximo de cuenta (el módulo) cuando vale 1. Del mismo modo y entendido el anterior, el contador descendente sigue un diseño análogo:

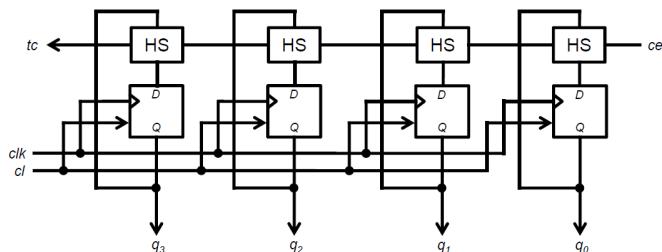
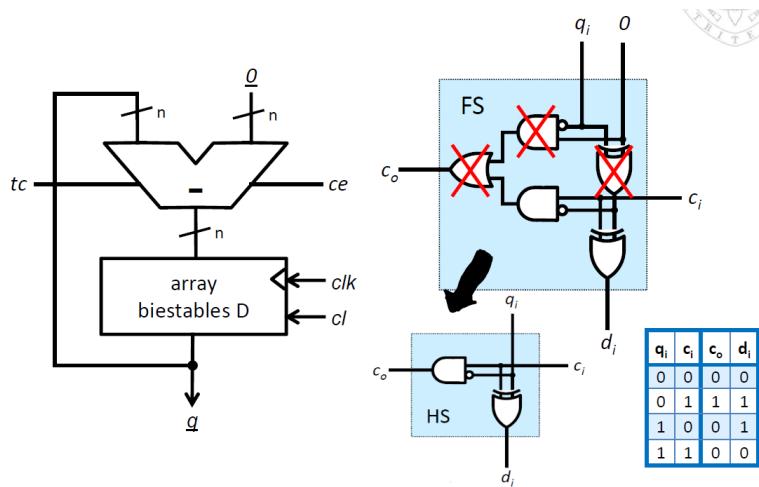


La implementación interna del contador, se hace parecida a la del sumador (o restador), pero cambiando los pequeños módulos que usábamos:



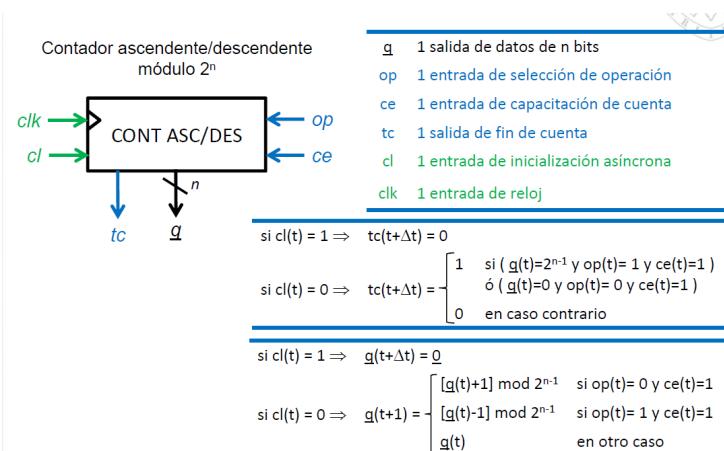


Implementación directa
Contador ascendente módulo 16

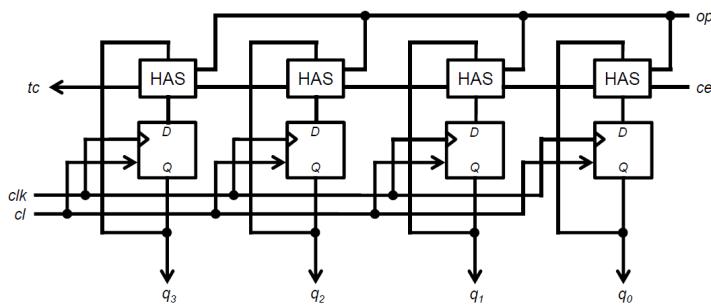
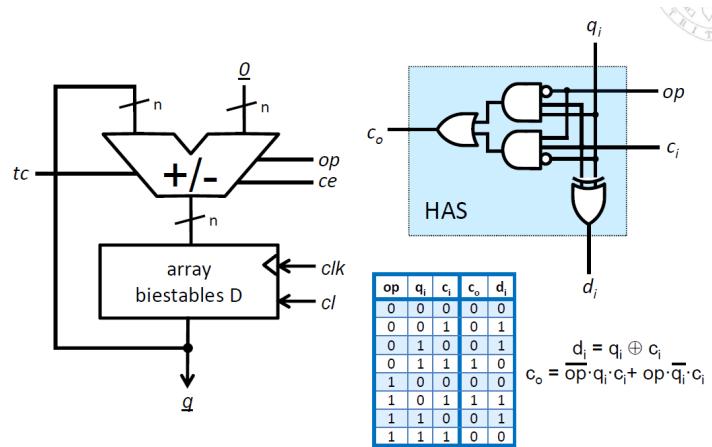


Implementación directa
Contador descendente módulo 16

Para poder implementar la funcionalidad de ambos en una única entidad, utilizamos el módulo del **contador ascendente/descendente**.

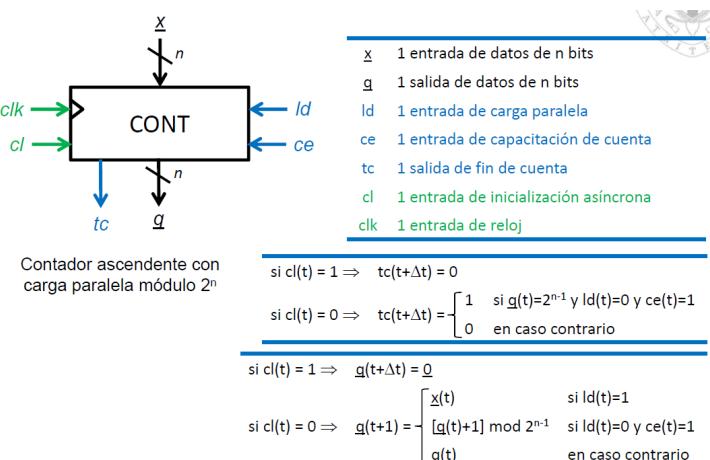


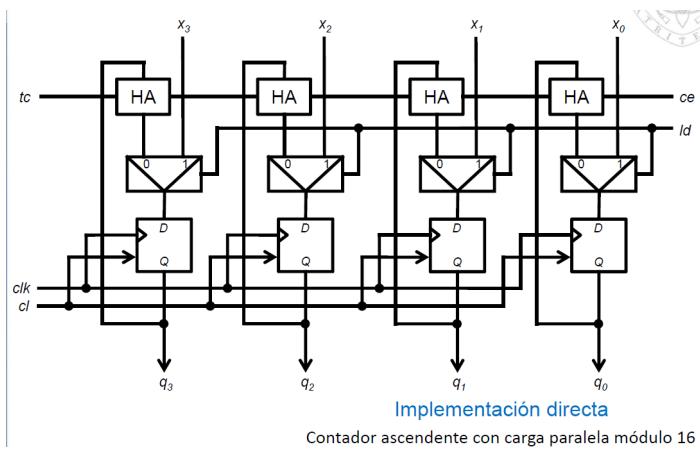
Como podemos ver ahora tenemos una nueva entrada, la entrada op , que se dedica a seleccionar el tipo de operación.



Implementación directa
Contador ascendente/descendente módulo 16

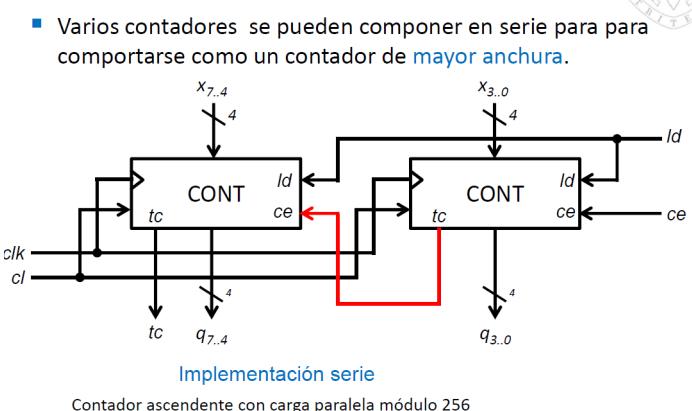
Existe un último tipo de contador que sirve para poder, además de hacer todo lo mencionado anteriormente, introducir como salidas una serie de entradas (al igual que si se tratara de un registro): el **contador de carga paralela**.





Aplicaciones de diseño

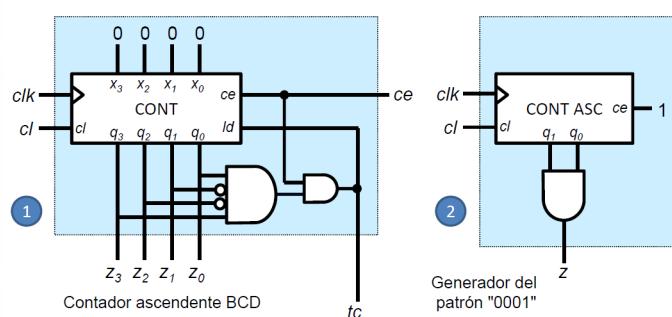
En primer lugar, varios contadores se pueden componer de forma paralela para actuar como uno de mayor anchura:



Además, un contador puede servir para **generar secuencias** que no tengan por qué seguir el orden de número del contador, por ejemplo:

En el primer contador vemos que a pesar de tener 16 posibles estados, como es un contador en BCD solo queremos que llegue hasta 9 y no podemos reducir el número de entradas porque el código BCD así lo exige. En este caso, se solventa indicando a la señal de *load* cuando debe cargar el estado 0, que es al recibir la codificación del 9 en BCD. La extrapolación de este diseño permite generar secuencias muy distintas al contador habitual generado por el número de entradas.

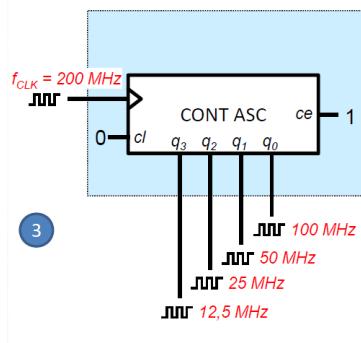
- Aplicaciones al diseño:**
 1. Generar secuencias (secuenciador).
 2. Generar patrones.



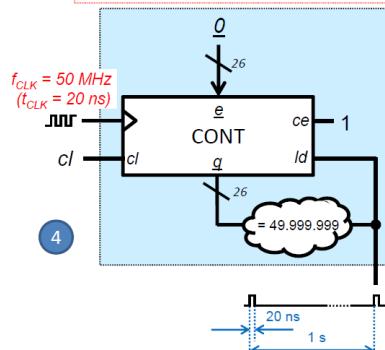
Por otro lado, es muy útil para **generar patrones**, es decir, igual que muchos módulos descrito ayudan a reconocer una secuencia de números, los contadores pueden ayudar a generarlas. Como vemos en el segundo, cuando se alcanza el número máximo del contador se devuelve 1 por lo que se genera todo el rato la secuencia 001. Si en vez de implementarlo así, definimos una función de commutación que devuelva 0 o 1 en función de la codificación de las salidas del contador, basta con crear un circuito combinacional conectado a esas salidas para que en cada codificación que recorra el contador, el circuito transforme esos valores en la salida deseada.

■ Aplicaciones al diseño:

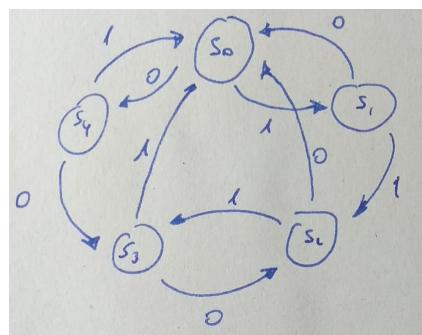
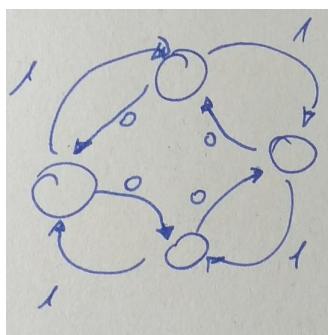
3. Dividir frecuencias.
4. Medir tiempo (temporizador).



$$\begin{aligned} (\text{num. ciclos}) &= \text{tiempo} / t_{clk} = \text{tiempo} \times f_{clk} \\ 1 \text{ s} &\equiv 1 \cdot 10^9 \text{ ns} \\ (1 \cdot 10^9 \text{ ns}) / (20 \text{ ns/ciclo}) &= 50.000.000 \text{ ciclos} \end{aligned}$$



Por último, con un contador somos capaces de **implementar cualquier circuito secuencial** porque si vemos el diseño que sigue un contador ascendente descendente, este tiene p estados donde p es el módulo del contador y pasa al siguiente o retrocede al recibir un 1 o retrocede al recibir un 0.



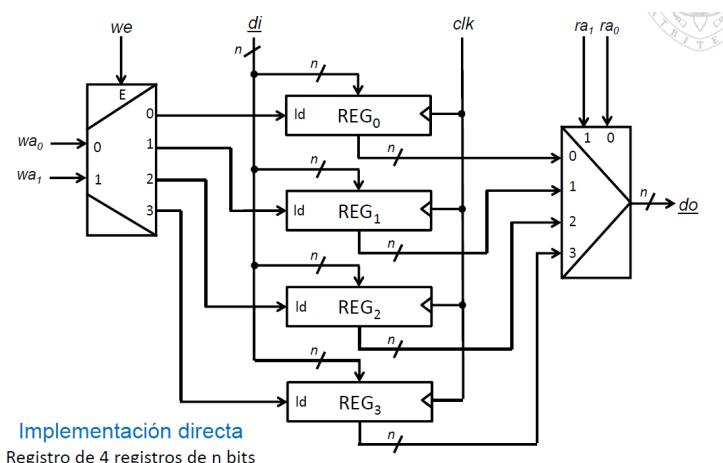
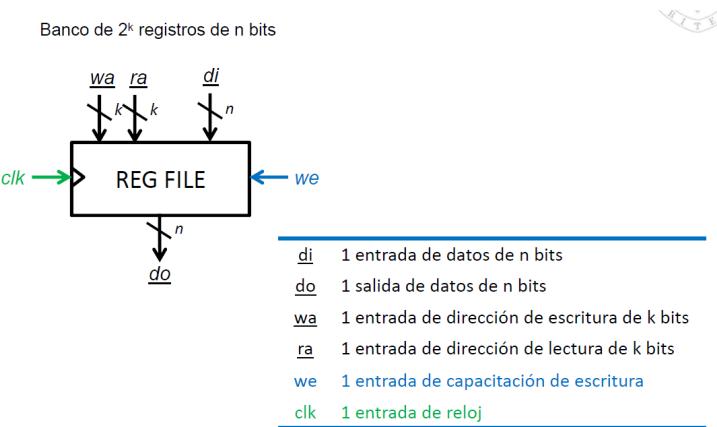
El problema viene cuando tenemos un circuito que en vez de seguir este esquema se salta estados, es decir, pasa de un estado n a $n + t$ donde $t \neq 1$, luego ¿cómo solventamos este problema?

Basta con indicar que cuando nos encontramos en uno de esos estados que no siguen el orden, en vez de contar al recibir una entrada que nos pasa a otro estado, decidimos cargar en la carga paralela la codificación del estado siguiente correspondiente, es decir, recibir esa entrada implica poner *load* a 1 y cargar las entradas de carga paralela con el estado siguiente correspondiente.

S_0	S_1	S_2	CE	L	x_0	x_1	x_2
0	0	0	0	0	—	—	—
0	0	0	1	0	—	—	—
0	0	1	0	0	—	—	—
0	0	1	1	0	—	—	—
0	1	0	0	1	0	0	0
0	1	0	1	0	—	—	—
0	1	1	0	0	—	—	—
0	1	1	1	1	0	0	0
1	0	0	0	0	—	—	—
1	0	0	1	0	—	—	—
1	0	1	0	—	—	—	—
1	0	1	1	—	—	—	—
1	1	0	0	—	—	—	—
1	1	0	1	—	—	—	—
1	1	1	0	—	—	—	—
1	1	1	1	—	—	—	—

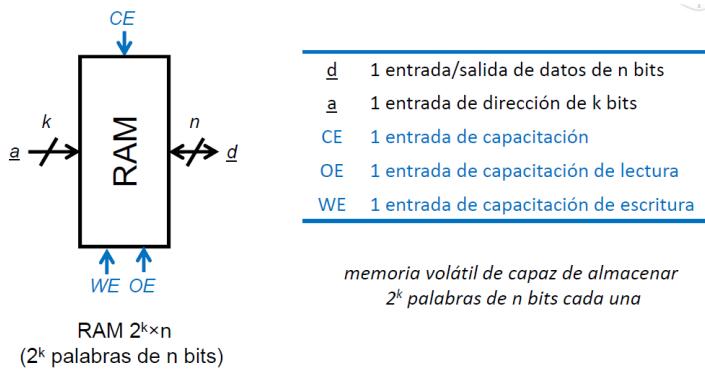
Banco de registros

Existe una construcción más potente que un registro y que es un modelo primitivo del funcionamiento real de las memorias, los **bancos de registros**, estos dispositivos son una asociación de registros sobre los que se tiene la capacidad de escribir (“write enable o we”) sobre un registro concreto (“write address o wa”) o de leer (“read address o ra”). Además poseen una entrada que es la que se copia en caso de escritura sobre el registro seleccionado y una salida que es la que refleja la información del registro elegido.

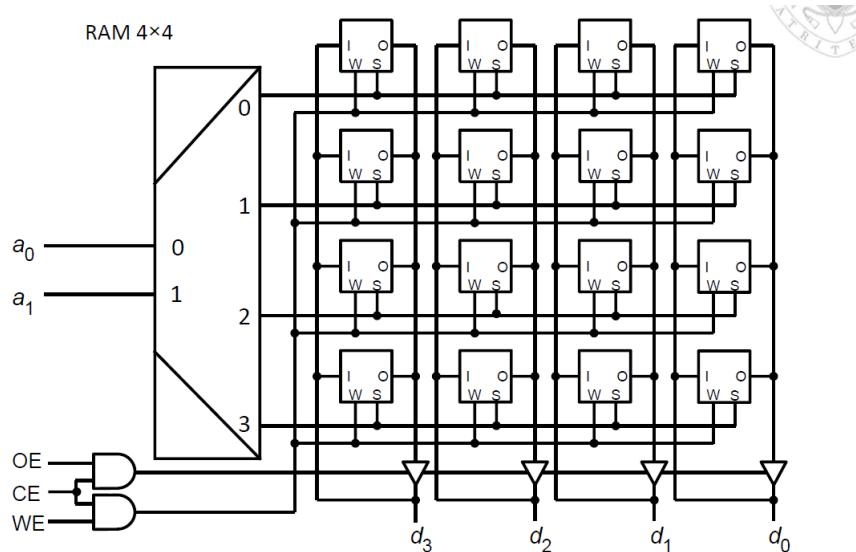
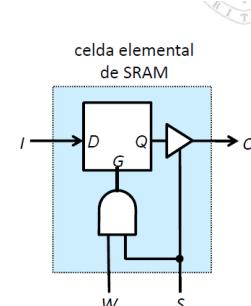


Memoria RAM

Sus siglas *random access memory* denotan la capacidad de escritura y lectura que la diferencia de una ROM. Se compone de una entrada de capacitación o *chip select (CE)*, una entrada de capacitación de lectura o *OE*, una entrada de capacitación de escritura o *write enable (WE)*, una entrada de n bits dirección de memoria para saber en qué dirección escribir/leer y, por último, una salida/entrada que muestra la información leída o introduce la información escrita a la dirección de memoria seleccionada.



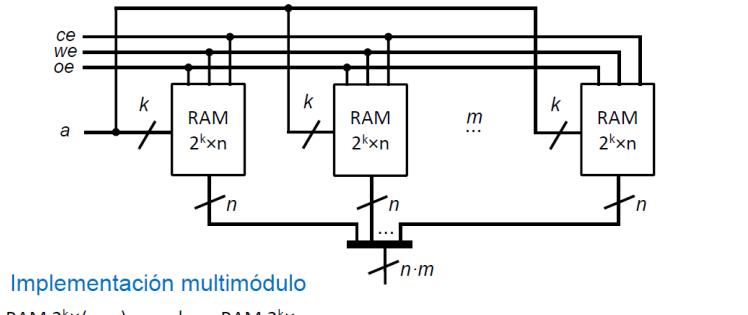
- **SRAM (Static RAM)**
 - Cada bit se almacena en un latch.
 - No requiere refresco.
- **DRAM (Dynamic RAM)**
 - Cada bit se almacena en un condensador.
 - Requiere refresco.
- **SDRAM (Synchronous Dynamic RAM)**
 - Cada bit se almacena en un condensador.
 - Requiere refresco.
 - El interfaz de lectura/escritura es síncrono.



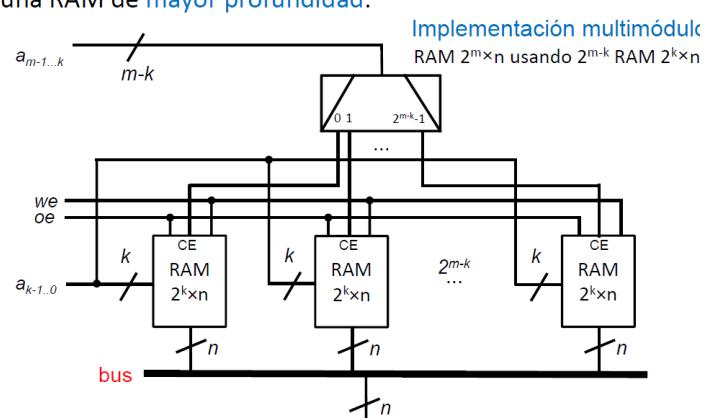
Aplicaciones de diseño

Del mismo modo que ocurría con las ROM, las RAM pueden asociarse de forma paralela para conseguir una mayor anchura de palabra y puede asociarse verticalmente para conseguir una mayor profundidad de palabra.

- Varias RAM se pueden componer para comportarse como una RAM de **mayor anchura de palabra**.



- Varias RAM se pueden componer para comportarse como una RAM de **mayor profundidad**.



DISEÑO DEL PROCESADOR

Ruta de datos y controladores

Todo lo visto hasta ahora es útil para, sobre una situación concreta, diseñar una máquina que resuelva de forma óptima dicho problema. La pregunta es: ¿se podrá hacer una máquina que valga para resolver múltiples problemas? Veamos por ejemplo el caso del algoritmo de multiplicación:

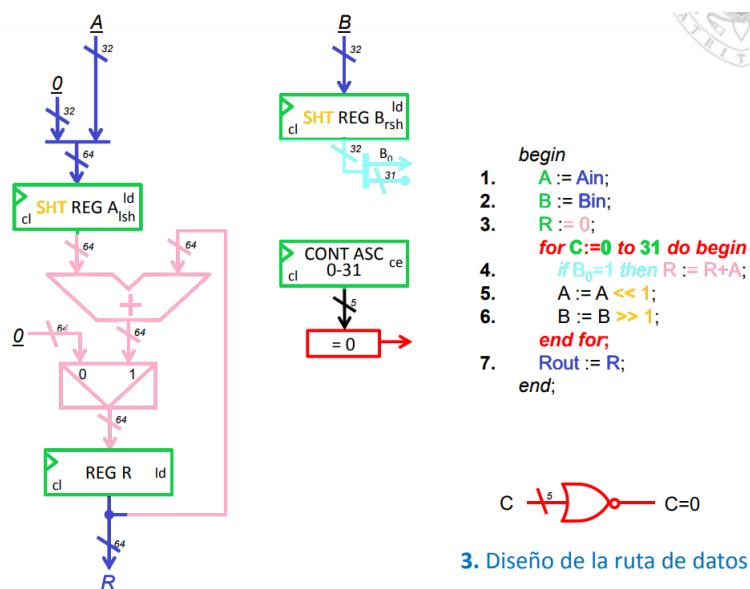
```

begin
1. A := Ain;
2. B := Bin;
3. R := 0;
for C:=0 to 2 do begin
4.   if B0=1 then R := R+A;
5.   A := A << 1;
6.   B := B >> 1;
end for;
7. Rout := R;
end;

```

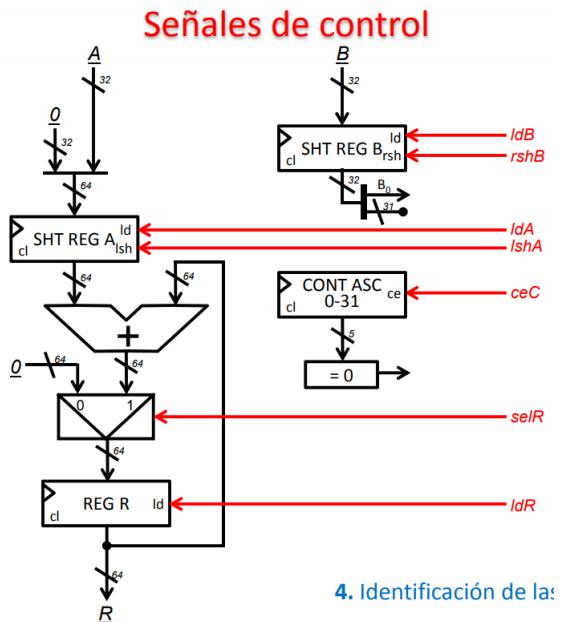
C	R ₅	R ₄	R ₃	R ₂	R ₁	R ₀	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	B ₂	B ₁	B ₀
1.	-	-	-	-	-	-	0	0	0	1	1	0	-	-	-
2.	-	-	-	-	-	-	0	0	0	1	1	0	1	0	1
3.	-	0	0	0	0	0	0	0	0	1	1	0	1	0	1
4.	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0
5.	0	0	0	0	1	1	0	0	0	1	1	0	0	1	0
6.	0	0	0	0	1	1	0	0	0	1	1	0	0	0	1
4.	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1
5.	1	0	0	0	1	1	0	0	1	1	0	0	0	1	0
6.	1	0	0	0	1	1	0	0	1	1	0	0	0	0	1
4.	2	0	1	1	1	1	0	0	1	1	0	0	0	0	1
5.	2	0	1	1	1	1	0	1	1	0	0	0	0	0	1
6.	2	0	1	1	1	1	0	1	1	0	0	0	0	0	0

En vez de realizar el diseño tal y como lo hemos hecho hasta ahora, podemos pensar en una estructura que me sirva para operar, guardar, modificar... los datos y crear un módulo que se dedique a distribuir los datos adecuadamente por la estructura.



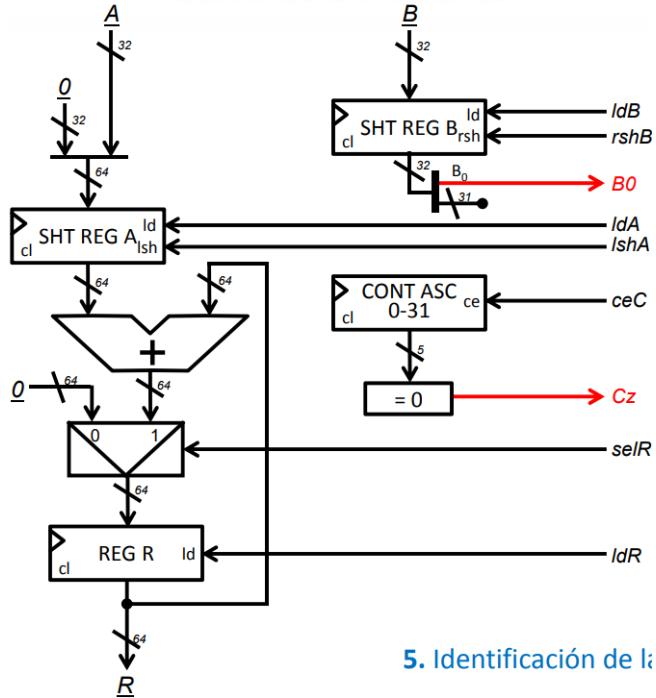
- Registros de desplazamiento para poder multiplicar los operandos por 2
 - Conexiones necesarias para que de algún sitio vengan los operandos pertinentes y otra extra para devolver el resultado de la multiplicación.
 - Módulo capaz de sumar ambos operandos que en este caso puede ser un sumador.
 - Multiplexor para poder inicializar en un primer momento el registro correspondiente a R y después poder seleccionar que los cambios que sufra sea por sumarle A.
 - Para comprobar ciertas condiciones sepáramos el bit 0 del operando B del que necesitamos conocer su valor.
 - Módulo que comprueba si el valor de C es 0 no para poder parar el bucle *for* que se está ejecutando.

Como hemos incluido muchos módulos que necesitan de una señal para activar o no sus funcionalidades, estas señales son las que serán el brazo ejecutor de la máquina que utilice esta ruta de datos para realizar las operaciones pertinentes, luego necesitamos que entren ciertas señales desde la máquina que controle la ruta.



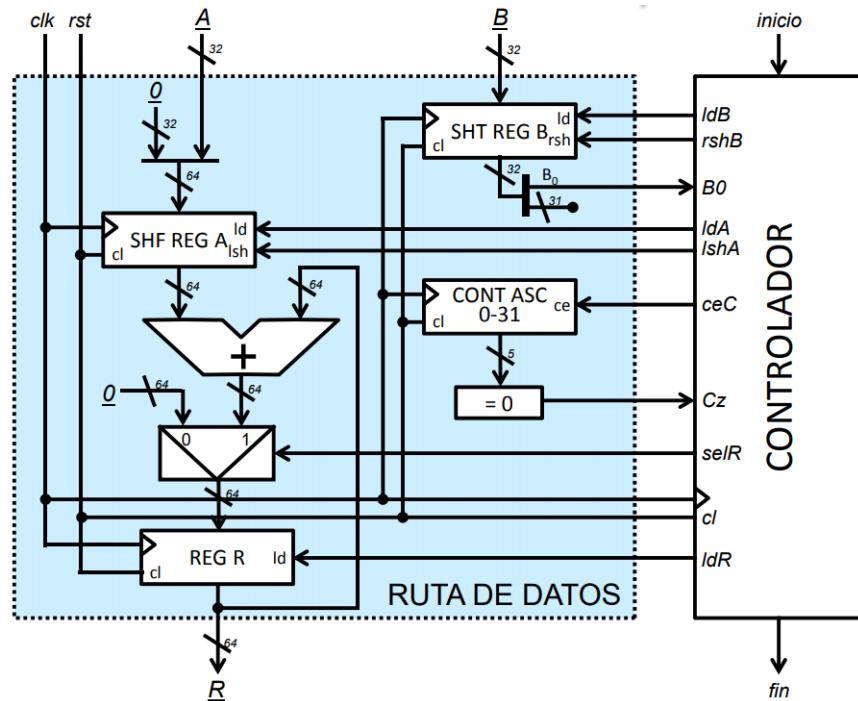
De modo análogo, hay información que solo conoce la ruta de datos que tiene que ser transmitida a la unidad de control para que en función de esos valores bifurque su toma de decisiones, por tanto constituirán las señales de estado que indican a la unidad de control hacia donde debe ir en los siguientes pasos.

Señales de estado



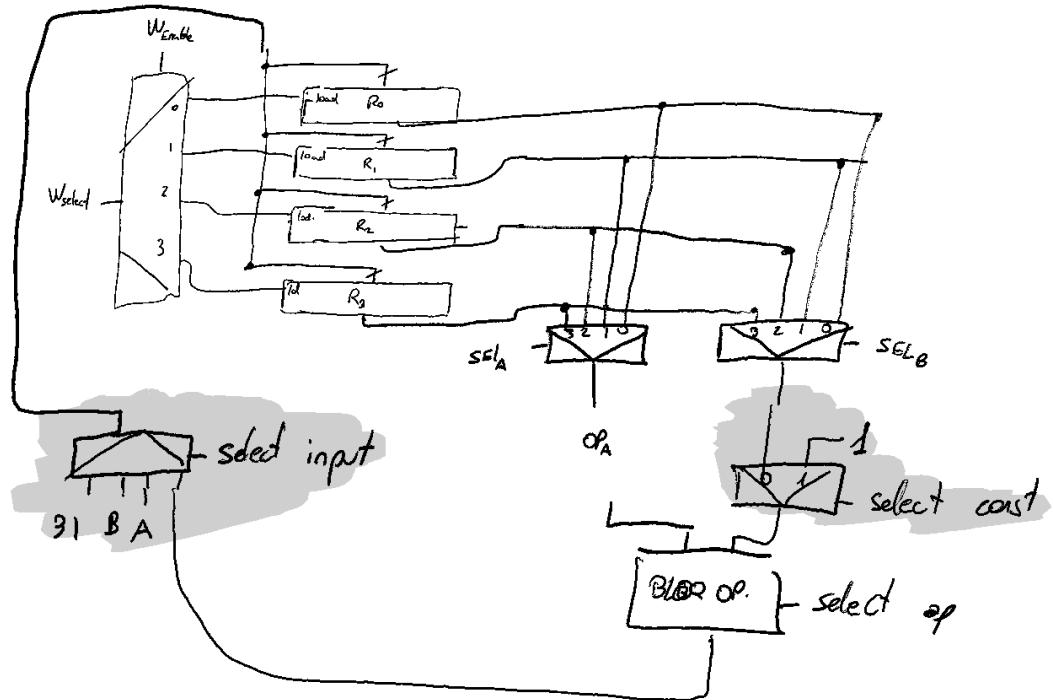
5. Identificación de la ruta de datos

Luego en conjunto, el sistema que nos queda al implementar la solución de este problema de la forma que lo hemos hecho es una estructura compuesta por dos módulos: una **ruta de datos** que se encarga de guardar, procesar y distribuir los datos del programa y una **unidad de control** que se encarga de dar órdenes y gestionar el flujo que debe seguir el programa y que la ruta de datos debe obedecer para operar con los datos.



Ruta de datos de propósito general

Tal y como lo hemos diseñado antes, ya tenemos una ruta de datos completamente funcional si queremos multiplicar números (habiendo diseñado correctamente la unidad de control que será una máquina de estados). La pregunta ahora es ¿cómo podemos modificar la misma para que sea posible usarla para cualquier operación aritmética?



Como vemos, hemos implementado, en vez de unos registros, un banco de registros conectado a dos multiplexores que son los encargados de elegir cuales se seleccionan para hacer la operación. Además, hemos incluido en el operando B un multiplexor pequeño para seleccionar entre el registro y una constante. Posteriormente hemos diseñado un bloque de operaciones (que habitualmente será una ALU) para elegir que operación se aplica a ambos operandos y, por último, hemos implementado un multiplexor que me permite: guardar los datos tras la operación en un registro, cargar datos (A y B) desde fuera de la estructura e introducir constantes en los registros.

De esta forma, hemos conseguido una ruta de datos de propósito aún más general que junto con una unidad de control conveniente serviría para ejecutar cualquier algoritmo con operaciones lógicas.

DISEÑO DE LA CPU

Un ordenador está compuesto principalmente por tres módulos: **cpu**, **memoria** y **periféricos de entrada y salida**. En este tema vamos a dedicarnos a estudiar el primero de ellos basándonos justamente en el conocimiento que tenemos de ensamblador del tema anterior y sobre una microarquitectura concreta.

A la hora de diseñar la CPU, que no es más que la unidad de control que maneja la ruta de datos, es importante tener en cuenta ciertos factores que cambiarán de un modo u otro el diseño final:

- **El repertorio de instrucciones:** que serán la base de nuestros programas. Cualquier computador debería tener al menos:
 - Instrucciones aritméticas

- Instrucciones de salto
- Instrucciones de escritura/lectura de memoria

Además, estas instrucciones van codificadas y como lo único que entiende el ordenador son las tiras de 0 y 1 que componen estas instrucciones, todo el funcionamiento del ordenador estará supeditado a la codificación que decidamos hacer sobre las mismas, modos de direccionamiento, número de operandos...

- **Los tipos de datos:** que determinarán que operaciones podemos hacer con ellos, como vamos a procesarlos...
- **El tamaño de palabra:** actualmente lo más habitual son los de 64 y 32 bits y determinan muchas cosas como el tamaño de almacenamiento entre otras.
- **El tipo de almacenamiento:**
 - En memoria: tiene más capacidad pero es más lenta
 - En banco de registros: tiene menos capacidad pero es muchísimo más rápido
- **El tiempo de ciclo:** que determinará la velocidad a la que se ejecutan los pasos de cada instrucción
- **El número de ciclos por instrucción:** que determinará el tiempo que tarda cada una en ejecutarse.

Todas estas decisiones de implementación son las que definen lo que se conoce como **arquitectura de un procesador**.

Arquitectura del procesador

Vamos a aprender con el modelo del procesador MIPS como es el proceso de diseño y construcción de un procesador real para un computador. Nosotros trabajaremos con una implementación simplificada del mismo en la cual:

- Todas las instrucciones son del mismo tamaño (32 bits).
- Solo 3 formatos de instrucciones.
- Solo hay 3 tipos de instrucciones: aritmético-lógicas, acceso a memoria y de salto condicional. De hecho, solo vamos a trabajar con estas:
 - LW Rt,desplaz(Rs)
 - SW Rt,desplaz(Rs)
 - ADD Rd,Rs,Rt
 - SUB Rd,Rs,Rt
 - AND Rd,Rs,Rt
 - OR Rd,Rs,Rt
 - BEQ Rs,Rt,desplaz
- Solo hay 3 modos de direccionamiento: directo a registro, indirecto a registro con desplazamiento e inmediato
- La temporización de las instrucciones será multiciclo

A pesar de que puede ser una simplificación demasiado exagerada, lo cierto es que esta en su conjunto no hace más que disminuir el rendimiento pues la mayoría de los programas necesitan básicamente de estas operaciones.

Formato de las instrucciones

Vamos definir ahora el código que van a tener las instrucciones concretas que se terminarán ejecutando como código máquina por nuestro ordenador. Como hemos dicho, habrá de tres tipos de formatos en general:

	31	26	21	16	11	6	0
Tipo R: aritmético-lógicas		op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	0
Tipo I: con memoria salto condicional	31	26	21	16			0
		op	rs	rt		inmediato	
	6 bits	5 bits	5 bits			16 bits	
Tipo J: salto incondicional	31	26					0
		op				dirección	
	6 bits					26 bits	

El significado de los campos es:

- **op:** identificador de instrucción
- **rs, rt, rd:** identificadores de los registros fuentes y destino
- **shamt:** cantidad a desplazar (en operaciones de desplazamiento)
- **funct:** selecciona la operación aritmética a realizar
- **inmediato:** es el valor del desplazamiento en direccionamiento a registro-base
- **dirección:** dirección destino del salto

Instrucciones aritmético lógicas:

Son del tipo R, tienen 3 registros, 2 operandos y 1 destino de la operación realizada. El direccionamiento será **únicamente directo a registro**.

31	26	21	16	11	6	0
000000	Rs	Rt	Rd	No usamos	funct	
OP=6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

- Campo OP (Código de Operación) = 000000
- Campo de los operandos:
 - Rs: Número del registro donde se encuentra el primer operando fuente
 - Rt: Número del registro donde se encuentra el segundo operando fuente
 - Rd: Número del registro donde se encuentra el operando destino
- Campo funct:
 - add: *funct* = 32 (100000)
 - sub: *funct* = 34 (100010)
 - and: *funct* = 36 (100100)
 - or: *funct* = 37 (100101)

Instrucciones de acceso a memoria:

Son del tipo I, tienen 2 registros; el de carga o descarga y el de direccionamiento. Solo hay dos modos de direccionamiento **directo a registro e indirecto a registro con desplazamiento**.



- Campo OP:
 - $OP = 35$ (100011): Load
 - $OP = 43$ (101011): Store
- Campo de los operandos:
 - Rs: Número del registro que se necesita para calcular la dirección de memoria
 - Rt:
 - Número del registro donde se encuentra el operando destino (inst Load)
 - Número del registro donde se encuentra el operando fuente (inst store)
- Desplazamiento: Contiene el valor que hay que sumarle al contenido del registro Rs para calcular la dirección de memoria.

Instrucciones de salto condicional:

Son del tipo I, tienen dos registros; los que se comparan para la condición y la dirección a la cual hay que saltar. Solo hay dos modos de direccionamiento, el directo a registro para los operandos y el inmediato para el desplazamiento (que estará incluida en la propia instrucción).



- Campo OP = (000100)
- Campo de los operandos fuentes:
 - Rs: Número del registro donde se encuentra el primer operando fuente.
 - Rt: Número del registro donde se encuentra el segundo operando fuente.
 - Desplazamiento: Contiene el valor del desplazamiento , este se necesita para calcular la dirección de memoria donde está la instrucción a la que hay que saltar.

Diseño de la ruta de datos

Una vez vistas las instrucciones, hemos de determinar como se ejecutarán las misas y sobre que ruta de datos se ejecutarán. Para ello, vamos a dividir todas las instrucciones en 5 fases claras de 1 ciclo cada una:

- **Fetch** (Lectura de instrucción): se lee la misma desde memoria
- **Deco** (Decodificación): se procesa el código máquina de la misma
- **Ex** (Ejecución): se realiza la operación especificada
- **Mem** (Acceso a memoria): para ciertas instrucciones
- **WB** (Almacenamiento de resultado): para guardar el resultado de la ejecución

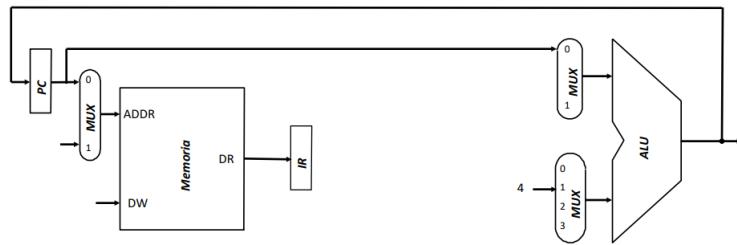
Fetch

Esta fase de lectura es común a todas las instrucciones puesto que siempre comienza una instrucción con su lectura desde la memoria. En esta etapa, se carga la instrucción desde memoria en un registro (**IR**) y se suma al contador de programa (**PC**) 4 para pasar a la siguiente instrucción, es decir, debemos realizar:

- $IR \leftarrow Instr$
- $PC \leftarrow PC + 4$

Luego necesitamos los siguientes componentes²⁵ hardware:

- Hardware necesario:**
- Contador de programa (PC)
 - Memoria
 - Registro de instrucción (IR)
 - ALU (la misma que usaremos para realizar op. aritmético-lógicas)
- IR <- Memoria(PC)
 PC <- PC + 4



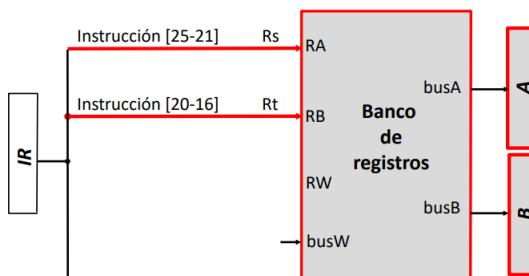
Decodification

Esta fase, de nuevo, es común a todas las instrucciones puesto que todas se valen de registros para realizar sus acciones. En esta etapa, se extraen de la instrucción los registros que tienen que ser seleccionados y se saca el valor de los mismos del banco de registros a dos registros auxiliares intermedios (**A** y **B**), es decir, debemos realizar:

- $A \leftarrow BR(Rs)$
- $B \leftarrow BR(Rt)$

Luego los componentes hardware necesarios para esta etapa van a ser:

- Hardware necesario:**
- Banco de registros **BR**
 - Líneas de selección de los registros **Rs** y **Rt** extraídas de los campos de **IR**
 - 2 registros de operandos, **A** y **B**



²⁵Los multiplexores o elementos adicionales no mencionados se incluyen por las posteriores implementaciones

Execution

Esta fase ahora es distinta en función de la instrucción puesto que es esencialmente la fase donde se producen los cambios que determinan cada una, luego es necesario ir viendo los componentes necesarios instrucción a instrucción.

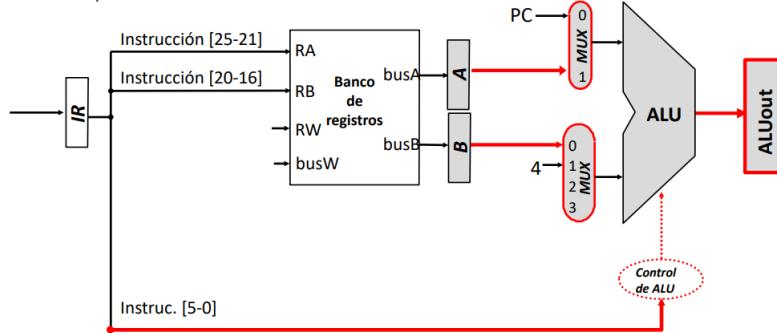
Aritmético-lógicas:

En este tipo de instrucciones (recordemos que ya tenemos los operandos en los registros A y B) lo que tenemos que hacer es realizar la operación aritmética asociada a la instrucción a ambos operandos, luego se va a realizar:

$$ALUout \leftarrow A \text{ funct } B$$

De modo que el hardware necesario se compone de los siguientes elementos:

- Conexiones desde A y B a las entradas de la ALU
- Registro **ALUout** para almacenar el resultado de la ALU
- Líneas **IR[5-0]** para selección de función de la ALU (Control ALU se describirá más adelante)



El registro *ALOut* se usa como registro intermedio donde conservar los datos cuando finaliza la fase, como los registros IR, A y B. El código de operación asociado se incluye dentro de la instrucción, luego es necesario sacar esos bits y llevarlo a una unidad que diseñaremos más tarde conocida como **control de la ALU**.

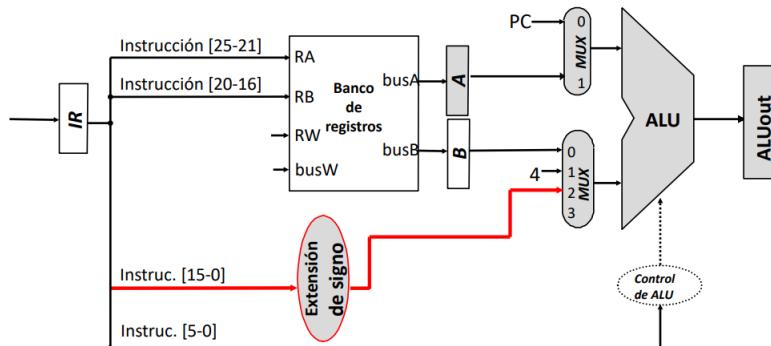
Acceso a Memoria:

Esta etapa, en esta instrucción, está constituida por el cálculo de la dirección de memoria efectiva sobre la que se quiere cargar o descargar cierto dato, es decir, vamos a realizar la siguiente operación:

$$ALUout \leftarrow A + extsign(IR[15 - 0])$$

Luego es necesario para la implementación el siguiente hardware:

- Extensor de signo para obtener 32 bits a partir de IR[15-0]
- Conexión del desplazamiento **sigext(IR[15-0])** a la entrada B de la ALU (nueva entrada del mux)



Salto:

La operación a realizar para estas instrucciones es sencilla, se comprueba que los dos registros especificados en la instrucción son iguales. Para ello simplemente se realiza $A - B$ y si el resultado es 0, entonces se activa una señal de control llamada 0 para indicárselo a la ruta de datos.

Memory Access

Esta fase de instrucción es solo necesaria para las instrucciones de *Load* que habíamos explicado, en ella lo que hacemos es cargar (con la dirección válida ya calculada) el dato requerido en un registro auxiliar y así preparar su posterior escritura en el banco de registros, es decir, ejecutamos la siguiente instrucción:

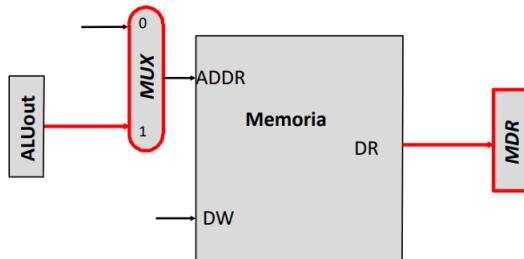
$$MDR \leftarrow Mem(ALUout)$$

Luego es necesario implementar el siguiente hardware a la ruta:

Nuevo hardware necesario:

- Memoria
- Conexión desde ALUout a **ADDR**
- Registro para almacenar el dato leído desde memoria (MDR)

$$MDR \leftarrow Mem(ALUout)$$



Write-Back

Esta es la última fase de la ejecución de una instrucción; la de guardado de los cambios producidos. Es común a todas las instrucciones y, por tanto, tenemos que distinguir por casos lo que se hace para poder hacer las implementaciones necesarias.

Aritmético-lógicas y Load:

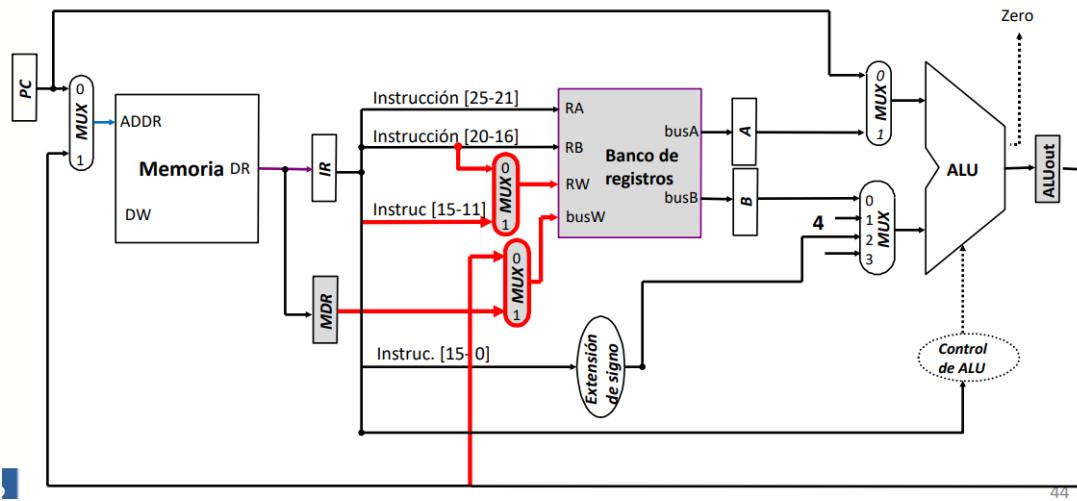
En este tipo de operaciones tenemos que guardar los datos en el banco de registros, para ello necesitamos especificar a qué registro debe ir el dato (**RW**) e introducir por el **bus W** el dato que vamos a guardar, es decir, vamos a implementar:

$$\blacksquare BR(Rd) \leftarrow ALUout$$

$$\blacksquare BR(Rs) \leftarrow MDR$$

En consecuencia, vamos a necesitar del siguiente hardware para poder implementar dichas operaciones:

- Dos formas de seleccionar el registro destino en **RW** del BR: Rd o Rt
- Conexiones desde ALUout y MDR a **busW** del Banco de registros



44

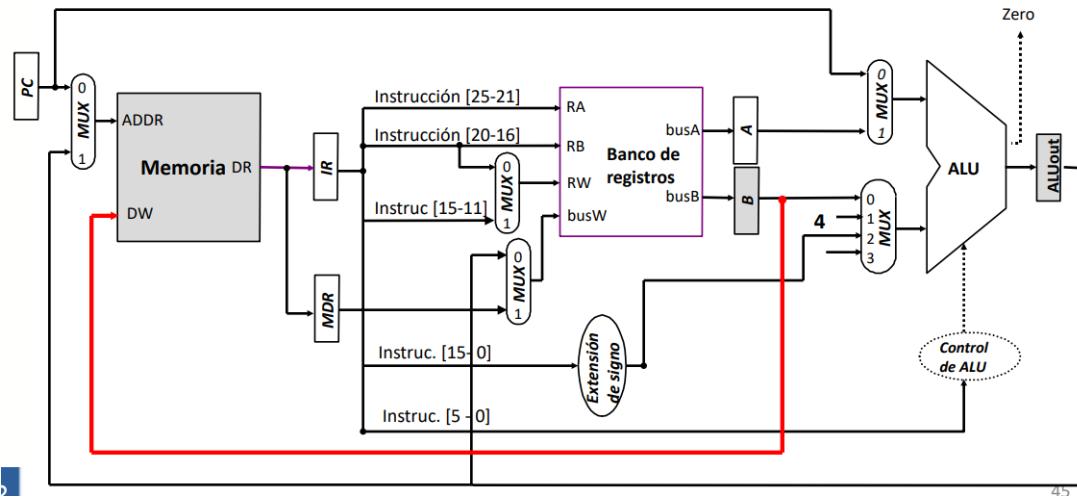
Store:

En este tipo de operaciones es necesario grabar un dato que teníamos en registro en una dirección de memoria. Ya que la dirección ha sido correctamente calculada en el paso previo, ahora basta con guardar este dato en memoria, es decir, que la operación que queremos hacer es:

$$\text{Mem}(ALUout) \leftarrow B$$

Luego para ello, es necesario implementar como hardware:

- Conexión de B con **DW** de la Memoria



45

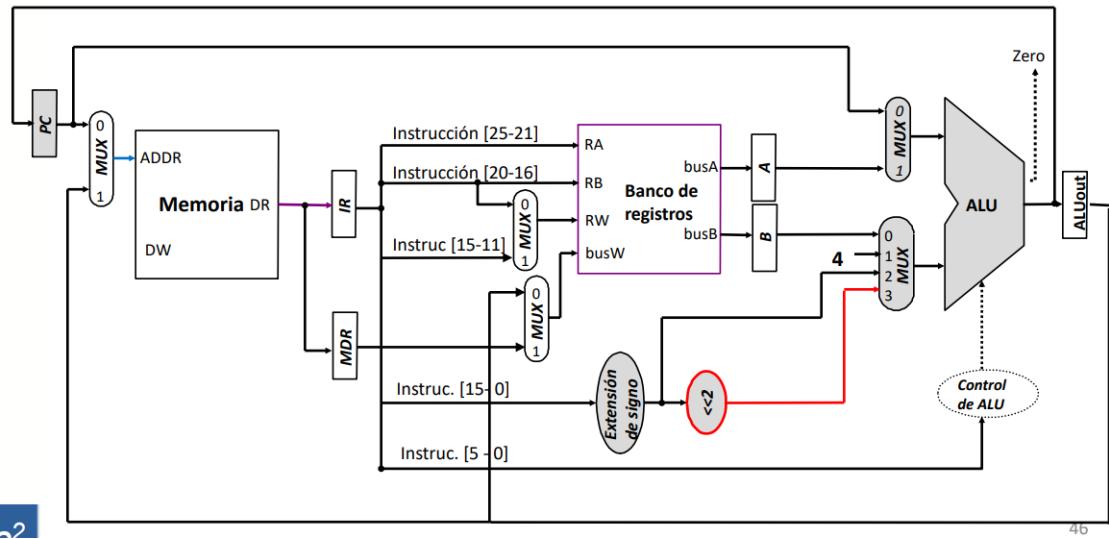
Salto condicional:

Por último, en estas instrucciones (si se ha verificado la condición de que los operandos sean iguales) es necesario hacer un salto en la ejecución del programa, es decir, necesitamos modificar el contador de programa. Como la dirección de salto está implícita en la propia operación, entonces tenemos que ejecutar la siguiente operación:

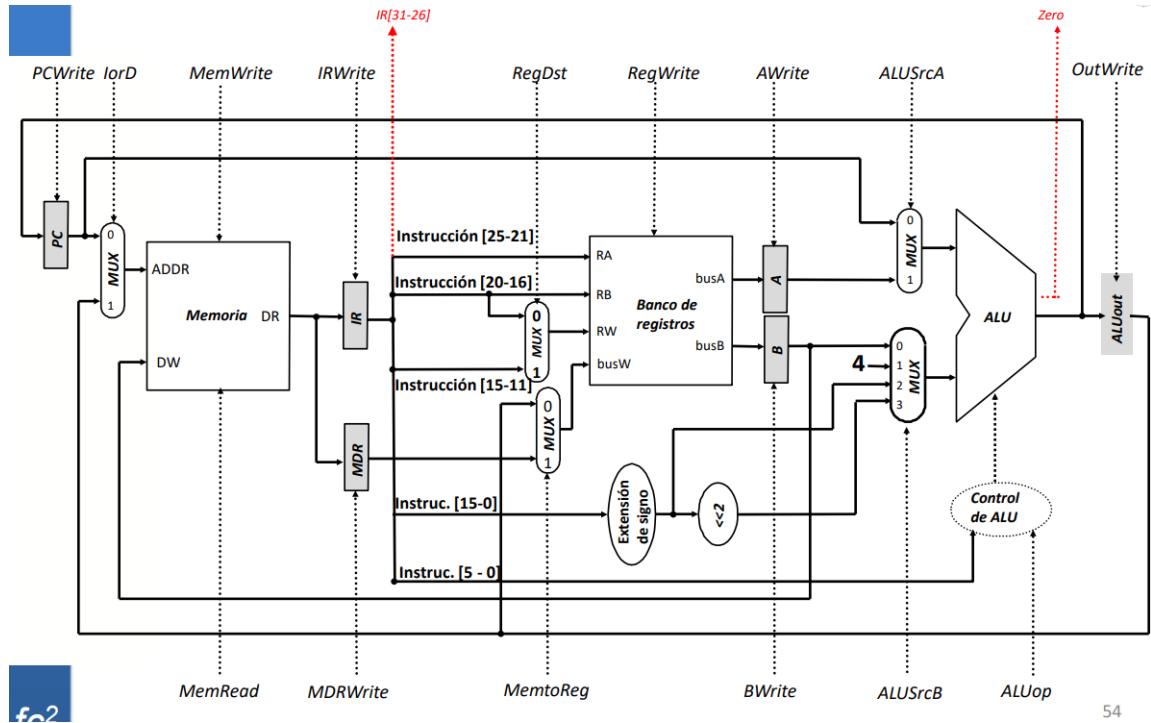
$$PC \leftarrow PC + 4 \cdot extsign(IR[15 - 0])$$

Luego es necesario implementar este elemento hardware:

- Desplazador a la izquierda para implementar la multiplicación por 4



Luego nuestra ruta de datos final, implementada con las condiciones e instrucciones seleccionadas quedaría de la siguiente forma:



Diseño del Controlador

Una vez diseñada la ruta de datos sobre la que vamos a trabajar, es evidente por la imagen anterior que hay ciertas señales de control que se dedican a dirigir el flujo de los datos dentro de esta ruta. Además, el orden de ejecución de las instrucciones es secuencial, luego parece lógico que sea necesario implementar una máquina de estados que controle dicha ruta para procesar los datos: **el controlador**.

Estados de ejecución

Esta máquina será diseñada como una máquina de Moore y, por tanto, tenemos que definir los distintos estados en función de lo que tenga que hacer la ruta de datos. Se van a indicar las señales relevantes en cada apartado, para todas las demás señales que no sean de registro estas son indiferentes, las que sean de registros y no estén entre las especificadas se ponen a 0.

Se recomienda observar la imagen anterior, en la que se especifica la ruta de datos completa, para seguir adecuadamente el desarrollo posterior de las señales y el funcionamiento de la ejecución de una instrucción.

Lectura de Instrucción (S_0):

En esta fase, que es común a todas las instrucciones del repertorio, lo que hacemos es leer de memoria la instrucción y almacenarla en el registro IR , luego es necesario generar:

$$IR \leftarrow Mem[PC]$$

$$PC = PC + 4$$

- $I_{orD} = 0$
- $MemRead = 1$
- $MemWrite = 0$
- $IRWrite = 1$
- $ALUSrcA = 0$
- $ALUSrcB = 1$
- $ALUop = \text{suma}$
- $PCWrite = 1$

Decodificación de Instrucción (S_1):

Esta fase también es común a todas las instrucciones, lo que debemos hacer en TODAS es por lo menos cargar el registro A . Para algunas instrucciones es necesario cargar también el registro B y como este no molesta (porque no se usa) para las instrucciones que solo usan el A hacemos que este estado sea general para todas las instrucciones cargando ambos (aunque para algunas el B solo sea basura).

$$A \leftarrow BR[Rs]$$

$$B \leftarrow BR[Rt]$$

- $AWrite = 1$
- $BWrite = 1$

Ejecución para Load (S_2)

Llegados a este punto, debemos distinguir entre instrucciones puesto que a pesar de que tenemos ejecuciones comunes en algunos casos (como este paso para el *Store* y para el *Load* como los siguientes pasos (que serán estados) son distintos, conforman estados distintos.

En esta fase es necesario calcular la dirección de memoria efectiva sobre la que cargaremos el dato, luego es necesario generar:

$$ALUout \leftarrow A + extsig(inmd)$$

- $ALUSrcA = 1$
- $ALUSrcB = 10_2$
- $ALUop = \text{suma}$
- $OutWrite = 1$

Memory Access para Load (S_3)

En esta fase es necesario cargar en MDR el dato especificado a través de la dirección de memoria para preparar su escritura en registro, es decir, vamos a ejecutar:

$$MDR \leftarrow Mem[ALUout]$$

- $I_{orD} = 1$
- $MemRead = 1$
- $MemWrite = 0$
- $MDRWrite = 1$

WriteBack para Load (S_4)

En esta última fase para Load, debemos escribir el dato que está en MDR que previamente hemos sacado en el registro especificado en la instrucción, es decir:

$$BR[Rt] \leftarrow MDR$$

- $RegDst = 0$
- $MemToReg = 1$
- $RegWrite = 1$

Memory Access para Store (S_5)

En esta fase es necesario cargar en MDR el dato especificado a través de la dirección de memoria para preparar su escritura en registro, es decir, vamos a ejecutar:

$$MDR \leftarrow Mem[ALUout]$$

A pesar de ser un estado completamente igual en cuanto a señales que el estado S_3 del acceso a memoria de Load, lo distinguimos por saltar posteriormente a un estado distinto de S_4 .

- $I_{orD} = 1$
- $MemRead = 1$
- $MemWrite = 0$
- $MDRWrite = 1$

WriteBack para Store (S_6)

En esta fase, escribimos en memoria (en la dirección correcta ya calculada anteriormente) el dato que queremos guardar del registro especificado, es decir:

$$Mem[ALUout] \leftarrow B$$

- $I_{orD} = 1$
- $MemWrite = 1$
- $MemRead = 0$

Ejecución para Aritmético-Lógicas (S_7)

En esta fase, realizamos la operación con los dos registros especificados (luego involucramos a la ALU) y guardamos el resultado en el registro de $ALUout$, es decir:

$$ALUout \leftarrow A \ op \ B$$

- $ALUSrcA = 1$
- $ALUSrcB = 0$
- $ALUop = \text{funct}$
- $OutWrite = 1$

WriteBack para Aritmético-Lógicas (S_8)

En esta última fase, realizamos la escritura del resultado (calculado previamente) sobre el registro especificado en la instrucción, es decir:

$$BR[Rd] \leftarrow ALUout$$

- $RegDst = 1$
- $MemToReg = 0$
- $RegWrite = 1$

Ejecución para Instrucciones de Salto (S_9)

En esta fase comparamos ambos registros especificados para ver si son iguales, es decir, calculamos su diferencia y, como novedad, **observamos si se activa la señal de zero para proseguir con la ejecución**, es decir:

$$A - B$$

- $ALUSrcA = 1$
- $ALUSrcB = 0$
- $ALUop = \text{resta}$

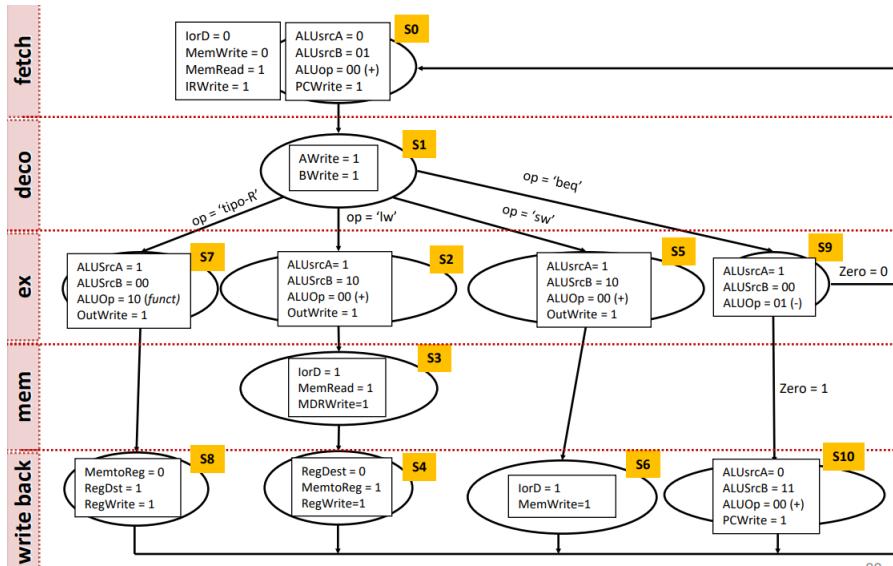
WriteBack para instrucciones de Salto (S_{10})

En este último estado, en caso de haber detectado la señal *zero*, tenemos que saltar a la dirección especificada, es decir, tenemos que modificar PC y sobreescibir en él la dirección a la que queremos saltar, luego:

$$PC \leftarrow PC + (\text{extsign}(inmd) \ll 2)$$

- $PCWrite = 1$
- $ALUSrcA = 0$
- $ALUSrcB = 11_2$
- $ALUop = \text{suma}$

Es decir, definidas todas las señales en cada estado concreto, podemos dibujar el diagrama de estados que dirige el flujo de ejecución del programa:



Implementación del controlador

De este modo, las distintas fases quedan definidas por completo y, en consecuencia, podemos implementar el controlador como una máquina de lógica secuencial en la que cada estado corresponde a cada una de las fases de ejecución de una instrucción:

- Posee 11 entradas:
 - Entradas de estado actual (4 bits).
 - Entrada de detección de 0.
 - Entradas [26-31] de OP de la instrucción.
- Tiene 21 salidas:
 - En la ruta de datos hay 17 especificadas
 - Las 4 de estados siguientes

Su tabla de verdad queda como:

Tabla de verdad del controlador

Estado actual	op	Zero	Estado siguiente	IRWrite	PCWrite	AWrite	BWrite	ALUsrcA	ALUsrcB	ALUOp	OutWrite	MemWrite	MemRead	IorD	MDRWrite	MemtoReg	RegDest	RegWrite
S0	XXXXXX	X	0001	1	1			0	01	00 (add)		0	1	0			0	
	100011 (lw)	X	0010									0	0				0	
	101011 (sw)	X	0101	0	0	1	1										0	
	000000 (tipo-R)	X	0111															
	000100 (beq)	X	1001															
S1	XXXXXX	X	0011	0	0			1	10	00 (add)	1	0	0				0	
	XXXXXX	X	0100	0	0								0	1	1	1	0	
	XXXXXX	X	0000	0	0								0	0		1	0	
S2	XXXXXX	X	0011	0	0												0	
	XXXXXX	X	0100	0	0												0	
	XXXXXX	X	0000	0	0												1	
S3	XXXXXX	X	0011	0	0												0	
	XXXXXX	X	0100	0	0												0	
	XXXXXX	X	0000	0	0												1	
S4	XXXXXX	X	0000	0	0												0	
	XXXXXX	X	0110	0	0												0	
	XXXXXX	X	0000	0	0												1	
S5	XXXXXX	X	1000	0	0												0	
	XXXXXX	X	0110	0	0												0	
	XXXXXX	X	0000	0	0												1	
S6	XXXXXX	X	1010	0	0												0	
	XXXXXX	X	0100	0	0												0	
	XXXXXX	X	0000	0	0												1	
S7	XXXXXX	X	1000	0	0												0	
	XXXXXX	X	0100	0	0												0	
	XXXXXX	X	0000	0	0												1	
S8	XXXXXX	X	1000	0	0												0	
	XXXXXX	X	0100	0	0												0	
	XXXXXX	X	0000	0	0												1	
S9	XXXXXX	0	0000	0	0												0	
	XXXXXX	1	1010	0	0												0	
	XXXXXX	X	0000	0	1												0	
S10	XXXXXX	X	1010	0	1												0	
	XXXXXX	X	0000	0	1												0	
	XXXXXX	X	0000	0	1												1	

Diseño del control de la ALU

En la ruta de datos que hemos especificado, hemos dejado indicado, incluso en algunas de las señales de control, todo lo relativo a como indicar a la ALU que operaciones realizar. En su caso, hemos dejado un módulo que hemos denominado **control de la ALU** que se encargará de indicarle a la misma que operaciones debe realizar en cada momento.

Tal y como hemos definido las señales de control y los estados en los apartados anteriores, las órdenes que van a la ALU se rigen por las siguientes tablas:

Estado	Operación
S_0	+
S_1	NULL
S_2	+
S_3	NULL
S_4	NULL
S_5	+
S_6	NULL
S_7	FUNCT
S_8	NULL
S_9	-
S_{10}	+

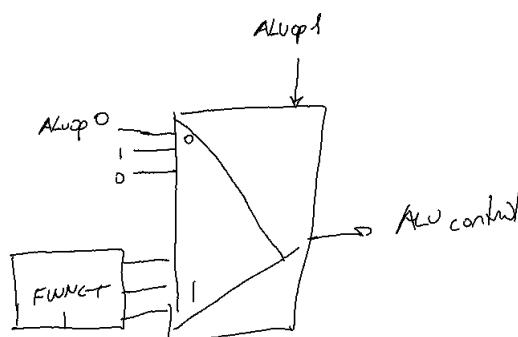
Código	Operación
00	add
01	sub
10	funct

ALUop	FUNCT	ALUctr
10	100100 (and)	000 (and)
10	100101 (or)	001 (and)
10	100000 (and)	010 (add)
00	xxxxxx	010 (add)
10	100010 (sub)	110 (sub)
01	xxxxxx	110 (sub)

Es decir, hemos de indicar al control cuando se trata de suma, diferencia o una operación FUNCT. Posteriormente, ha de generarse desde el control de la ALU la señal de control que realmente indicará que hacer a la misma, por lo que podemos distinguir dos casos:

- Se trata de una suma o una diferencia no FUNCT
- Se trata de una operación FUNCT

Tal y como se ha especificado en las tablas de arriba queda explicado este apartado, puesto que la implementación de este tipo de sistemas no concierne a este tema, sin embargo, se deja el siguiente diseño a modo de ejemplo:



SISTEMAS DE MEMORIA

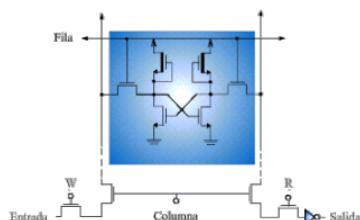
Como hemos visto en el capítulo anterior, el procesador precisa de un módulo conocido como memoria del que recoge tanto los datos como las instrucciones que se van a ejecutar. Parece lógico, por tanto, entrar a describir en detalle cuál es el funcionamiento y como programar un módulo de este tipo.

MEMORIA PRINCIPAL

Es la memoria que hemos descrito en el apartado anterior y sirve para que el procesador encuentre los datos e instrucciones del programa. En general podemos encontrar dos tipos:

- Memoria no volátil:
 - ROM
 - PROM, EPROM
 - EEPROM,FLASH
 - Disco magnético, Disco sólido
- Memoria Volátil:
 - SRAM
 - DRAM

Habitualmente, la memoria encargada de esta tarea en un ordenador común es una memoria RAM que puede estar implementada en ambas modalidades descritas en la memoria volátil:



Celda SRAM

Ventajas

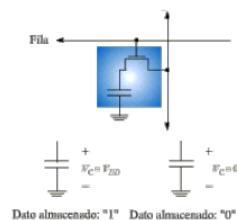
Tiempo de acceso y de ciclo reducido

Desventajas

Disipan mucha energía

Baja densidad de integración

Coste elevado



Celda DRAM

Ventajas

Bajo consumo de energía

Alta densidad de integración

Coste reducido

Desventajas

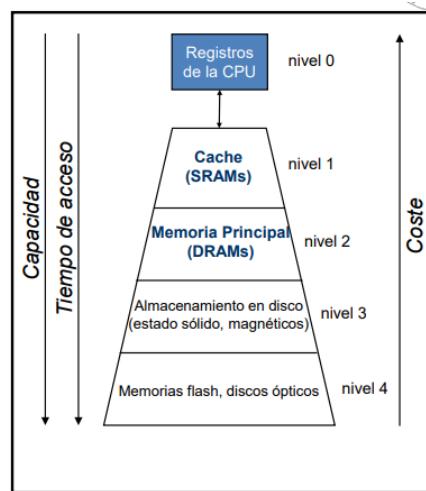
Tiempo de ciclo elevado

Necesidad de refresco

Para ver el detalle el diseño y la implementación de las memorias RAM, es recomendable ir al apartado que se le otorga en el tema de Módulos Secuenciales.

Jerarquía de memoria

Sin embargo, la memoria principal no es la única utilizada puesto que los tiempos de acceso son muy largos y su consumo es elevado, en su lugar se tienen varios niveles de memoria organizados jerárquicamente:



Esta organización se hace con el objetivo de conseguir una memoria rápida, de bajo consumo y coste reducido. Se persigue hacer creer al procesador que tiene más memoria y que esta es más rápida de lo que es en realidad, es decir, que está trabajando con la capacidad del nivel inferior, pero con la velocidad del nivel superior al que se encuentra.

Para maximizar la gestión de este sistema de memoria es necesario basarse en dos principios:

- **Localidad espacial:** los datos cercanos a uno usado recientemente son muy susceptibles de ser usado más adelante, por ejemplo: arrays, ejecución de instrucciones, matrices...
- **Localidad temporal:** un dato recientemente utilizado suele ser requerido de nuevo en un plazo corto de tiempo.

Con lo cual, un buen sistema de memoria mantiene los datos recientes cerca del procesador para que tarden menos tiempo en ser leídos y mueve también los datos cercanos a uno pedido para no estar consultando de forma reiterada la memoria principal.

GESTIÓN DE MEMORIA CACHÉ

Ya hemos visto que acceder a memoria principal es costoso y además lento, por tanto, para asegurar un buen funcionamiento del computador es necesario implementar una solución más rápida que facilite dicha tarea; la **memoria caché**.

Funcionamiento de la memoria caché

La ventaja principal que tiene esta memoria intermedia entre la principal y la CPU es que explota al máximo los principios indicados en la jerarquía de memoria, por tanto, cuando la CPU pida un dato a memoria en primer lugar se mirará a la memoria caché y:

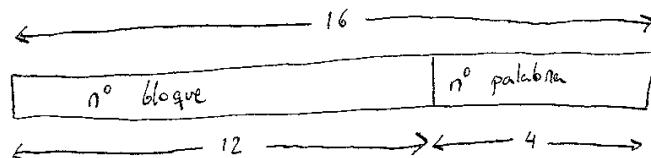
- Si el dato está en ella, se produce un acierto → se leerá directamente desde ahí (que es infinitamente más rápido)

- Si el dato no está, se produce un fallo → se cargará no solo ese dato en la memoria caché sino los datos cercanos al mismo.

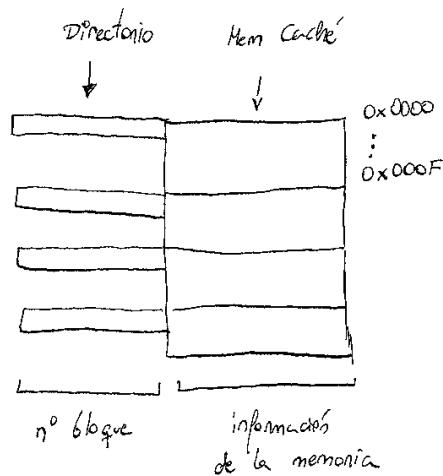
La unidad de gestión que se copia al copiar todos los datos cercanos a uno que ha sido pedido es el **bloque**. La memoria caché se divide en varias **líneas de caché o marcos** en los que irá colocado un bloque concreto.

Una vez entendido como está compuesta es necesario entender como se gestiona dicha memoria, para ello vamos ver un ejemplo real.

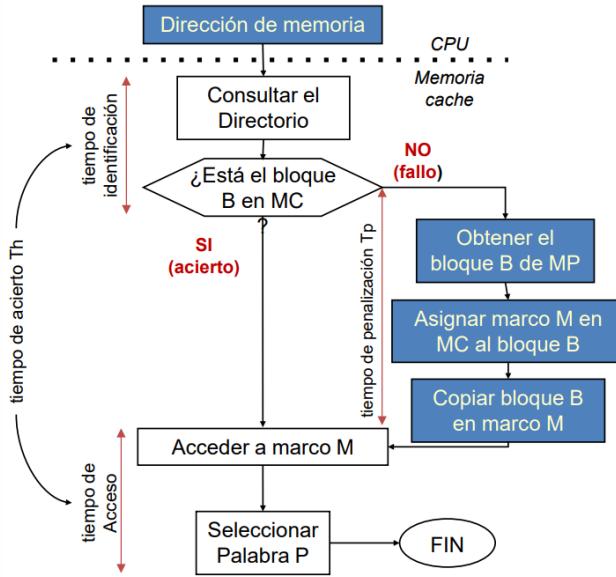
Supongamos que tenemos direcciones de 16 bits, es decir, tenemos una memoria principal de 2^{16} bytes. Es necesario elegir (que es una cuestión de diseño, no canónica) el tamaño de los bloques, es decir, la cantidad de datos que se van a copiar junto con el pedido; en este caso los haremos de 16 bytes. De este modo, podemos dividir la dirección de memoria en estas partes:



Los primeros 12 bits me indican en qué bloque me encuentro puesto que al haber 2^{16} bytes en la memoria principal y ser cada bloque de 2^4 bytes al dividir me quedan 2^{12} posibles bloques que quedan especificados con esos 12 bits. Por otro lado, para indicar a qué palabra del bloque me refiero se utilizan los 4 bits finales, puesto que dentro de un bloque puedo elegir hasta 16 palabras distintas (porque cada bloque es de 16 bytes).



A la primera parte es a lo que llamaremos **etiqueta** y es lo que el sistema consulta para saber si un bloque está o no en la memoria caché. Para poder realizar dicha tarea se crean los **directorios** que no son más que un anexo anterior a la memoria que guarda las etiquetas de los bloques almacenados. Así que cuando se requiere un dato, se mira en los directorios para determinar si está o no, si no está se carga el bloque entero y si está entonces se mira dentro del bloque cuyo directorio dió acierto y se utilizan los bits de palabra para elegir la palabra deseada.



Políticas de emplazamiento, reemplazamiento y actualización

No es difícil ver que uno de los inconvenientes de la memoria caché es que hay muchas menos líneas de caché que bloques hay en la memoria principal, luego es necesario **reemplazar** estos bloques cuando todas las líneas están ocupadas y se quiere cargar un nuevo bloque.

Asimismo, si colocamos los bloques de forma aleatoria, incluyéndolos en cualquier línea que esté libre, entonces tendré que comprobar todos los directorios cuando quiera ver si un bloque está o no, lo que no es práctico. Para solventar este problema existen las **políticas de emplazamiento** que sirven para ordenar en cierto modo el cómo y donde se incluye cada bloque al ser copiado.

Por último, si un dato se modifica desde la CPU, si está en la memoria caché su bloque, entonces se modificará sólo en la caché y en la memoria principal estará el dato sin actualizar. Para determinar como se actualizan estos datos en la memoria principal para que halla una coherencia entre los datos de la caché y los de la principal se crean las **políticas de actualización**.

Políticas de emplazamiento

Fundamentalmente podemos distinguir 3 técnicas principales para decidir en que lugar y de que forma se copian los bloques en la memoria caché:

- **Directo** → a cada bloque se le asigna una única línea y solo se comprueba si está en esa porque solo se le espera ahí.
- **Asociativo** → se copia el bloque en la primera línea de caché libre que este disponible.
- **Asociativo por conjuntos** → trata de aunar ambas políticas anteriores para sacar partido de ambas.

Nosotros vamos a trabajar con **emplazamiento directo**. El funcionamiento es sencillo: si la memoria caché tiene 2^n líneas de caché se utilizan los n bits menos significativos de la etiqueta para determinar en que línea puede ir ese bloque, de esta forma se le asigna una única línea a cada bloque y cuando se pregunta por él solo se le espera en esa.

Para asegurar que las comprobaciones de los directorios se hacen sobre valores válidos (no los datos que pudiera haber de un uso anterior), se incluye un bit para determinar la validez de la comprobación: el **bit de válido**.

Políticas de actualización

Hemos visto como se cargan los datos desde memoria en la CPU a través de la memoria caché y como se gestiona esta, ¿pero cuál es el criterio para cuando la CPU escribe algo en memoria?

- **Escritura inmediata** → cada vez que se hace una escritura en memoria y ese dato está en la memoria caché se actualiza la memoria principal.
 - Ventaja: se mantiene la coherencia entre ambas memorias
 - Desventaja: se aumenta mucho el tráfico de datos entre memorias
- **Post-escritura** → la memoria principal solo se actualiza cuando el bloque de la memoria caché va a ser borrado y ha sido modificado por la CPU.
 - Ventaja: se disminuye mucho el tráfico y se reduce el tiempo de ejecución
 - Desventaja: si hay varios procesadores usando la misma memoria, no hay coherencia entre ellas.

Para la política de post-escritura es necesario incluir en el directorio un bit más que va a indicar si ese bloque se ha modificado por la CPU o no (para después en su caso tomar las medidas correspondientes): el **dirt bit**.