

Universidad de Murcia  
Facultad de Informática

---

---

TÍTULO DE GRADO EN  
INGENIERÍA INFORMÁTICA

# Estructura y Tecnología de Computadores

Tema 1: Sistemas Digitales - Circuitos Combinacionales

Apuntes

CURSO 2010 / 11

---

---

VERSIÓN 2.1

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores



## Índice general

1.1. Introducción . . . . .	1
1.1.1. Álgebra de Boole . . . . .	3
1.1.2. Simplificación de funciones mediante Mapas de Karnaugh . . . . .	5
1.2. Circuitos combinacionales comunes . . . . .	10
1.2.1. Puertas lógicas básicas . . . . .	10
1.2.2. Retardos . . . . .	12
1.2.3. Implementación de funciones lógicas con puertas NAND/NOR . . . . .	12
1.2.4. Bloques lógicos . . . . .	14
1.2.5. Codificadores y decodificadores . . . . .	15
1.2.6. Multiplexores . . . . .	17
1.2.7. Memorias ROM y arrays lógicos programables . . . . .	20
1.2.8. Unidad aritmético lógica . . . . .	22
<b>B1. Boletines de prácticas</b>	<b>24</b>
B1.1. Uso del simulador TkGate . . . . .	24
B1.1.1. Objetivos . . . . .	24
B1.1.2. Prerequisitos . . . . .	24
B1.1.3. Plan de trabajo . . . . .	24
B1.1.4. El simulador TkGate . . . . .	25
B1.1.5. Ejercicios . . . . .	25
B1.2. Unidad aritmético lógica . . . . .	26
B1.2.1. Objetivos . . . . .	26
B1.2.2. Prerequisitos . . . . .	26
B1.2.3. Plan de trabajo . . . . .	26
B1.2.4. Implementación de una ALU de 1 bit . . . . .	26
B1.2.5. Implementación de una ALU de 32 bit . . . . .	28
B1.2.6. Ejercicios . . . . .	34
<b>E1. Ejercicios</b>	<b>35</b>
E1.1. Sistemas Digitales: Circuitos Combinacionales . . . . .	35
E1.2. Solución a ejercicios seleccionados . . . . .	38

## 1.1. Introducción

Los ordenadores que se fabrican hoy en día son dispositivos digitales en donde la información se representa de forma discreta, frente a los dispositivos analógicos que manejan formas continuas de información. La electrónica digital opera solamente con dos niveles de tensión de interés: una tensión alta y una tensión baja. Los demás valores de tensión únicamente se presentan durante la transición entre los dos valores anteriores. Por este motivo, los ordenadores utilizan sistemas de numeración binarios, lo que nos permite hacer coincidir la abstracción en la que se basan los ordenadores con la electrónica digital. Según la familia lógica, los valores y relaciones entre los dos niveles de tensión difieren. Por tanto, en lugar de referenciar niveles de tensión, hablamos de señales que son (lógicamente) verdaderas ó 1; y señales que son (lógicamente) falsas ó 0.

La figura 1 muestra la implementación de una puerta NOT mediante el uso de una resistencia y un transistor NMOS. Cuando el voltaje de entrada es menor que un cierto valor  $V_T$  el transistor, que funciona a grandes

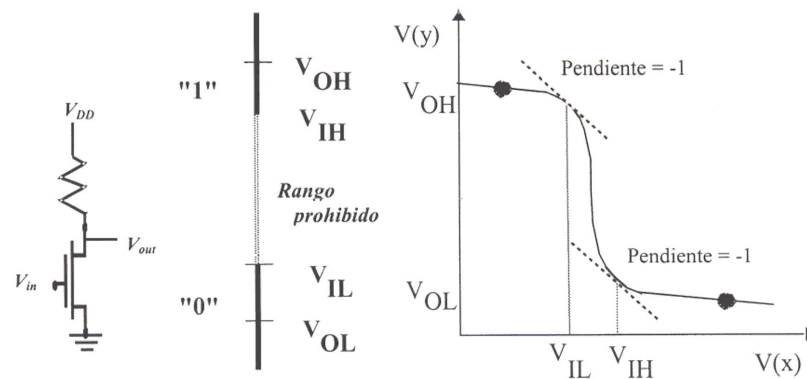


Figura 1: Implementación de una puerta NOT mediante una resistencia y un transistor.

rasgos como un interruptor, no conduce y el voltaje de salida  $V_{out}$  es igual a  $V_{DD}$  (la alimentación del circuito). Cuando el voltaje de entrada supera ese valor, el interruptor se cierra y la corriente circula por el transistor haciendo que el voltaje de salida  $V_{out}$  sea igual a 0.

Para establecer una correspondencia entre niveles de voltaje y valores lógicos usaremos el concepto de valor umbral. Los voltajes por encima de un cierto valor umbral representarán un valor lógico (normalmente el uno lógico), mientras que los valores por debajo de ese umbral representarán el otro valor (normalmente el cero lógico). Dado que en la realidad el voltaje en cualquier punto de un circuito electrónico puede variar levemente por diversos motivos, los valores cercanos al valor umbral son desaconsejables ya que, dependiendo de las circunstancias, pueden ser interpretados de diferente manera. Para evitar esta ambigüedad, se establece un rango prohibido, tal y como se muestra en la misma figura. Este rango viene determinado por el análisis de la llamada función característica del voltaje de salida de un inversor construido con la tecnología a utilizar. En este caso, valores por debajo de  $V_{IL}$  representan el valor 0, y voltajes por encima de  $V_{IH}$  representan el valor 1.

Aunque la implementación de cualquier función lógica mediante una resistencia de carga y un conjunto de transistores NMOS es factible, presenta dos problemas importantes que hace que este tipo de circuitos no se utilicen en la actualidad. El primero de ellos es el espacio en silicio que ocupa una resistencia, demasiado grande en comparación con el que ocupa un transistor. Esto provoca problemas en la escala de integración de puertas en el chip. El segundo y principal problema es el ligado al consumo de este tipo de circuitos: cuando la salida está a nivel lógico 0, el transistor se encuentra conduciendo y se produce un consumo de energía (y el correspondiente calentamiento) en la resistencia que impediría la fabricación de circuitos integrados con millones de puertas lógicas como los que necesitan los microprocesadores actuales.

La alternativa utilizada hoy en día es la sustitución de la resistencia por un transistor PMOS. Este tipo de transistor se comporta a la inversa del transistor NMOS: cuando se le aplica un voltaje alto al terminal de la puerta el transistor, que actúa como un interruptor, permanece abierto, cerrándose cuando a la entrada aparece un cero lógico. La figura 2 muestra la implementación de un inversor, así como la función característica de la salida.

La característica clave de este circuito es que los transistores operan de modo complementario; cuando uno conduce, el otro está en corte. Así, siempre existe un camino desde la salida hacia  $V_{DD}$  o hacia tierra. No existe camino entre la fuente y tierra, excepto durante un corto periodo donde los transistores están cambiando de estado. Esto significa que el circuito no disipa potencia cuando permanece en un estado estable, sólo cuando cambia de un estado lógico a otro. Por lo tanto, la disipación de potencia del circuito depende del número de cambios que se produce por segundo. Esta es la principal causa de que, con el aumento en la frecuencia de operación de los procesadores, aumenten los problemas de consumo y calentamiento.

Finalizaremos esta introducción comentando que en la electrónica digital podemos diferenciar dos grandes familias de circuitos: *circuitos combinacionales* y *circuitos secuenciales*. Si la salida en un instante dado sólo

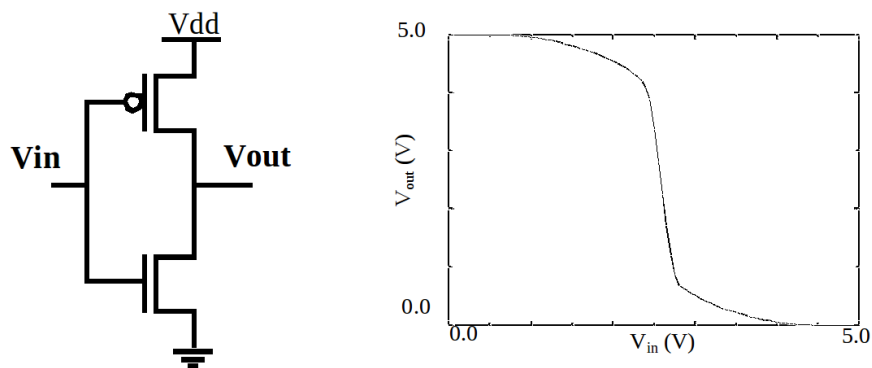


Figura 2: Inversor CMOS.

depende de las entradas en ese instante entonces nos encontramos ante un circuito combinacional. En el caso de que la salida dependa de entradas que se produjeron en instantes anteriores hablaremos de circuito secuencial. En este tema nos dedicaremos a ver algunos de los circuitos combinacionales más usuales, dejando para el tema siguiente el estudio de los circuitos secuenciales.

### 1.1.1. Álgebra de Boole

En 1854, George Boole publicó una obra titulada *Investigación de las leyes del pensamiento*, sobre las que se basan las teorías matemáticas de la lógica y la probabilidad. En esta publicación se formuló la idea de un álgebra de las operaciones lógicas, que se conoce hoy en día como Álgebra de Boole. El Álgebra de Boole es una forma muy adecuada para expresar y analizar las operaciones de los circuitos lógicos. Claude Shannon fue el primero en aplicar la obra de George Boole al análisis y diseño de circuitos.

Como ya visteis en la asignatura de Lógica, un álgebra booleana es un sistema Algebraico cerrado formado por un conjunto  $K$  de dos o más elementos (en nuestro caso, asumiremos que  $K = \{0, 1\}$ ); los dos operadores binarios AND (el resultado es 1 si los operandos son 1) y OR (el resultado es 1 si alguno de los operandos es 1), conocidos también como producto lógico ( $\cdot$ ) y suma lógica ( $+$ ), respectivamente; y el operador unario NOT (el resultado es la inversión o negación del operando, es decir, 0 si el operando es 1 y viceversa) denominado negación lógica ( $\bar{\phantom{x}}$ ).

Sobre dicha álgebra es posible definir funciones lógicas. Una función lógica es una función del tipo  $F : \{0, 1\} \times \{0, 1\} \times \dots \times \{0, 1\} \rightarrow \{0, 1\}$ , o lo que es lo mismo  $F : \{0, 1\}^n \rightarrow \{0, 1\}$ . La función  $F$  posee  $n$  entradas cada una de las cuales puede tomar un valor 0 ó 1. Dependiendo de estos valores la salida a su vez tomará también uno de los dos valores. Por tanto, tenemos  $2^n$  combinaciones de entrada para cada una de las cuales la función toma un valor.

Cualquier función lógica puede escribirse como una ecuación lógica, con una salida en la parte izquierda de la ecuación, y una fórmula, que consta de variables, complementos de variables y operadores, en la parte derecha. Para una función lógica se pueden encontrar varias ecuaciones lógicas diferentes equivalentes. De éstas, para facilitar el trabajo, se eligen determinadas formas de suma de productos y producto de sumas, denominadas formas canónicas, que se caracterizan por ser únicas y que se describen a continuación.

#### Minitérminos

Dada una función de  $n$  variables de entrada, si un término producto contiene cada una de las  $n$  variables, ya sea en forma complementada o no, es un minitérmino. Si la función se representa como una suma de minitérminos, se dice que la función tiene forma de **suma canónica de productos**, o que está en **forma normal conjuntiva**. Por ejemplo:

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

Para simplificar la escritura en forma de suma canónica de productos, se usa una notación especial. Se elige un orden fijo para las variables en los minterminos y a cada mintermino se le asocia un número cuya representación binaria tiene  $n$  bits que resultan de considerar como 0 las variables complementadas y como 1 las variables no complementadas. Así la función booleana anterior se representaría como:

$$F(A, B, C) = m_2 + m_3 + m_6 + m_7 = \sum m(2, 3, 6, 7)$$

### Maxitérminos

Dada una función de  $n$  variables de entrada, si un término suma contiene cada una de las  $n$  variables, ya sea en forma complementada o no, es un maxitérmino. Si la función se representa como un producto de maxitérminos, se dice que la función tiene forma de **producto canónico de sumas**, que está en **forma normal disyuntiva**. Por ejemplo:

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

Análogamente al caso de la forma normal conjuntiva, podemos simplificar la expresión de la función, indicando los maxitérminos que incluye. Sin embargo, en este caso cada variable complementada corresponde a un 1 y viceversa, al contrario que antes:

$$F(A, B, C) = M_0 \cdot M_1 \cdot M_4 \cdot M_5 = \prod M(0, 1, 4, 5)$$

**Ejercicio:** Dada la siguiente función, expresarla en forma de lista de minterminos:

$$F(A, B, Q, Z) = \bar{A} \cdot \bar{B} \cdot \bar{Q} \cdot \bar{Z} + \bar{A} \cdot \bar{B} \cdot \bar{Q} \cdot Z + \bar{A} \cdot B \cdot Q \cdot \bar{Z} + \bar{A} \cdot B \cdot Q \cdot Z$$

**Solución:**

$$F(A, B, Q, Z) = m_0 + m_1 + m_6 + m_7 = \sum m(0, 1, 6, 7)$$

Alternativamente, es posible emplear una tabla de verdad. En una tabla de verdad aparecen a la izquierda tantas columnas como entradas tenga la función y una columna para la salida. Cada fila representa uno de los posibles valores de entrada junto con su salida (la tabla de verdad de una función resulta de aplicar la ecuación lógica para todos y cada uno de los valores de entrada). Por tanto, completando todos los valores de la tabla, se tiene definida totalmente la función.

**Ejemplo:** La función lógica  $F$  tiene tres entradas  $A$ ,  $B$  y  $C$ , y una salida. La salida es verdadera si exactamente dos entradas son verdaderas. A continuación se muestra la ecuación lógica, y la tabla de verdad para esta función se muestra en la tabla 1.

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

Las funciones lógicas que operan sólo sobre 0's y 1's también son llamadas funciones de conmutación.

ENTRADAS			SALIDA
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 1: Tabla de verdad correspondiente a la función F.

### 1.1.2. Simplificación de funciones mediante Mapas de Karnaugh

La simplificación de las funciones lógicas es una meta importante. Su importancia radica en el hecho de que cuanto más sencilla sea la función, más fácil será construir el circuito equivalente. El objetivo de la simplificación es minimizar el costo de implementación de una función mediante componentes electrónicos, donde el costo depende del número y complejidad de los elementos necesarios para construirla.

Los mapas de Karnaugh constituyen un método sencillo y apropiado para la minimización de funciones lógicas, limitado en la práctica hasta cinco o seis variables. Como ya visteis en la asignatura de *Lógica*, un mapa de Karnaugh es una representación gráfica de una tabla de verdad y, por tanto, existe una asociación unívoca entre ambas. La tabla de verdad tiene una fila por cada minitérmino, mientras que el mapa de Karnaugh tiene una celda por cada minitérmino. De manera análoga, también existe una correspondencia unívoca entre las filas de la tabla de verdad y las celdas del mapa de Karnaugh si se utilizan los maxitérminos. En la figura 3 se muestran los mapas de Karnaugh correspondientes a 2, 3 y 4 variables de entrada. En dicha figura, cada celda aparece etiquetada con su respectivo número de minitérmino, o lo que es lo mismo, con el número de fila de la tabla de verdad con el que se correspondería. Las etiquetas de las filas y las columnas se refieren a los valores de las variables.

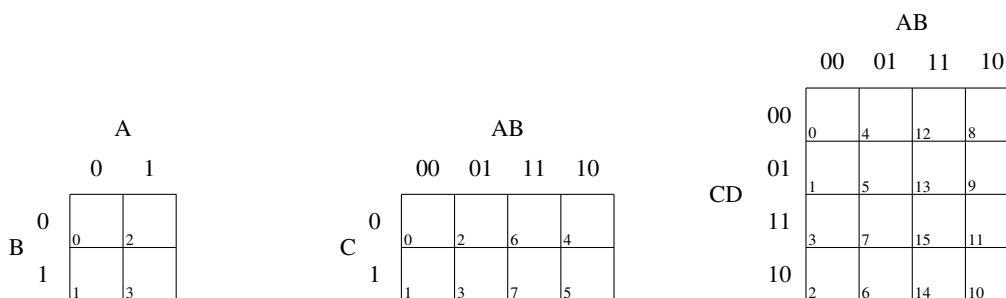


Figura 3: Mapas de Karnaugh para diferentes números de variables.

La minimización de funciones sobre el mapa de Karnaugh se aprovecha del hecho de que sobre el mapa, los términos adyacentes desde el punto de vista lógico, también son adyacentes físicamente. Definimos los minitérminos adyacentes desde el punto de vista lógico como dos minitérminos que difieren sólo en una variable. Por ejemplo,  $ABC\bar{D}$  y  $ABC\bar{D}$  son minitérminos de cuatro variables adyacentes lógicamente, ya que sólo difieren en la variable D.

Dos términos adyacentes pueden combinarse eliminando la variable en la que difieren, usando las propiedades distributiva y complementaria ( $A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot D = A \cdot B \cdot \bar{C} \cdot (D + \bar{D}) = A \cdot B \cdot \bar{C}$ ). En el mapa indicamos los términos que se pueden combinar trazando un anillo alrededor de ellos. Estos términos al combinarse nos darán una expresión más sencilla, o sea, con menos variables.

**Ejercicio:** Simplificar la función  $F(A, B, C, D) = \sum m(1, 2, 4, 6, 9)$  mediante su mapa de Karnaugh.

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

La expresión simplificada de la función queda:

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

En el paso 1 combinamos los minitérminos 1 y 9 que son adyacentes lógicamente. La variable que cambia es A que, por tanto, se elimina, quedando  $\bar{B} \cdot \bar{C} \cdot D$ . Del mismo modo procedemos en los pasos 2 y 3. En el paso 2 combinamos los minitérminos 4 y 6 que se diferencian en la variable C, y en el 3 los minitérminos 2 y 6 que difieren en la variable B.

### Criterios para la simplificación de funciones con mapas de Karnaugh

Hay cinco reglas que debemos recordar para poder simplificar funciones representadas sobre mapas de Karnaugh:

1. Cada cuadrado (minitérmino) sobre un mapa de Karnaugh de dos variables tiene dos cuadrados (minitérminos) adyacentes lógicamente; cada cuadrado sobre un mapa de Karnaugh de tres variables, tiene tres cuadrados adyacentes, etc. En general, cada cuadrado en un mapa de Karnaugh de  $n$  variables tiene  $n$  cuadrados adyacentes lógicamente, de modo que cada par de cuadrados adyacentes difiere precisamente en una variable.
2. Al combinar los cuadrados en un mapa de Karnaugh, agruparemos un número de minitérminos que sea potencia de dos. Al agrupar dos cuadrados eliminamos una variable, al agrupar cuatro cuadrados eliminamos dos variables, etc. En general, al agrupar  $2^n$  cuadrados eliminamos  $n$  variables.
3. Debemos agrupar tantos cuadrados como sea posible; cuanto mayor sea el grupo, habrá un número menor de literales en el término producto resultante.
4. Debemos formar el menor número posible de grupos que cubran todos los cuadrados (minitérminos) de la función. Un minitérmino está cubierto si está incluido al menos en un grupo. Si hay menos grupos, será menor el número de términos de la función minimizada. Podemos utilizar cada término tantas veces como sea necesario en los pasos 4 y 5; sin embargo, debemos usarlo al menos una vez. Tan pronto hayamos utilizado todos los minitérminos al menos una vez nos detenemos.
5. La combinación de cuadrados en el mapa se hará siempre empezando por los minitérminos que tienen menor número de cuadrados adyacentes (los más solitarios en el mapa). Los minitérminos con varios términos adyacentes ofrecen más combinaciones posibles y, por tanto, deben combinarse más adelante en el proceso de minimización.

### Terminología para la minimización de funciones lógicas

Los términos que vamos a exponer son útiles en los procedimientos de simplificación de funciones lógicas, pero no sólo con mapas de Karnaugh sino también con otras técnicas más generales de minimización.

- **Implicante:** Término producto que puede servir para cubrir los minitérminos de una función.
- **Implicante primo:** Implicante que no es parte de ningún otro implicante de la función. En el mapa de Karnaugh, un implicante primo equivale a un conjunto de cuadrados que no es subconjunto de algún conjunto con un mayor número de cuadrados.
- **Implicante primo esencial:** Implicante primo que cubre al menos un minitérmino que no está cubierto por algún otro implicante primo.
- **Cubierta:** Conjunto de implicantes primos tal que todos los minitérminos de la función están contenidos en al menos un implicante primo. La cubierta de una función debe incluir, como mínimo, todos los implicantes primos esenciales posibles.

**Ejercicio:** Identificar sobre la siguiente función mediante su mapa de Karnaugh, los implicantes, los implicantes primos, los implicantes primos esenciales y la cubierta:

$$F(A, B, C) = \sum m(0, 2, 3, 6, 7)$$

**Solución:**

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

- *Implicante* → 11 implicantes: (5 minitérminos, 5 grupos de dos minitérminos y 1 grupo de 4 minitérminos).
- *Implicante primo* → 2 implicantes primos,  $B$  y  $\overline{A} \cdot \overline{C}$ .
- *Implicante primo esencial* → 2 implicantes primos esenciales,  $B$  y  $\overline{A} \cdot \overline{C}$ .
- *Cubierta* →  $\{B, \overline{A} \cdot \overline{C}\}$ .

### Algoritmo de minimización mediante mapas de Karnaugh

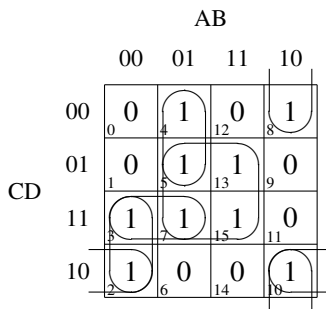
1. Identificar los implicantes primos. Para esto se busca obtener los grupos con mayor cantidad de unos adyacentes. Los grupos deben contener un número de unos que sea potencia de 2.
2. Identificar todos los implicantes primos esenciales.
3. La expresión mínima se obtiene seleccionando todos los implicantes primos esenciales y el menor número de implicantes primos para cubrir los minitérminos no incluidos en los implicantes primos esenciales.

**Ejercicio:** Utilizar el mapa de Karnaugh para simplificar la función:

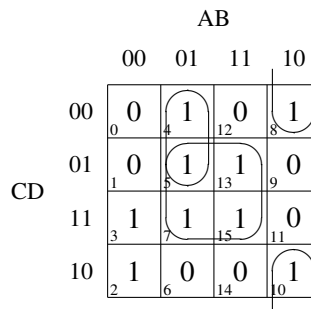
$$F(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$$

**Solución:**

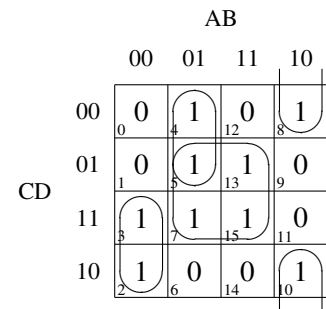
Implicantes primos:



Implicantes primos esenciales:



Cubierta:

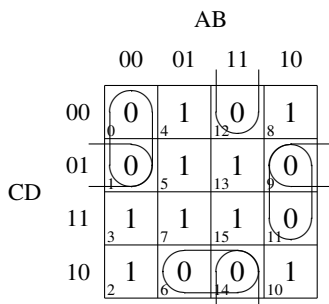


La función minimizada queda:

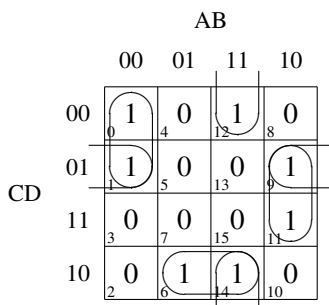
$$F(A, B, C, D) = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C} + B \cdot D + A \cdot \bar{B} \cdot \bar{D}$$

### Simplificación por ceros

A veces puede ser interesante agrupar los ceros en el Mapa de Karnaugh en lugar de los unos, ya se deba a que sea más sencilla dicha agrupación o porque nos interese obtener una expresión simplificada en forma de producto de sumas. Veamos cómo se realiza la simplificación en este caso utilizando la misma función que en el apartado anterior. El primer paso es dibujar el Mapa de Karnaugh marcando todos los implicantes primos:



El problema que nos encontramos es que no sabemos cómo se eliminan las variables que cambian en las agrupaciones, por lo que para obtener las reglas a seguir, escribiremos en este caso concreto (más adelante no hará falta) el Mapa de Karnaugh de la función  $\bar{F}$ :



Esta función sí sabemos simplificarla siguiendo los pasos habituales, con lo que expresión simplificada será:

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

Teniendo en cuenta que  $\overline{\overline{F}} = F$ , la expresión simplificada de  $F$  será por lo tanto (aplicando las leyes de De Morgan):

$$F = \overline{\overline{F}} = \overline{\overline{A} \cdot \overline{B} \cdot \overline{C} + B \cdot C \cdot \overline{D} + A \cdot B \cdot \overline{D} + A \cdot \overline{B} \cdot D} = (A+B+C) \cdot (\overline{B} + \overline{C} + D) \cdot (\overline{A} + \overline{B} + D) \cdot (\overline{A} + B + \overline{D})$$

Si observamos el resultado obtenido y vemos el primer Mapa de Karnaugh, observamos que la variable que se elimina sigue siendo la que cambia. De las variables que no cambian, éstas aparecerán negadas cuando en las casillas correspondientes aparezcan a 1 y sin negar cuando aparezcan a 0. Por último, la expresión obtenida es de producto de sumas, es decir, la expresión que representa una agrupación de ceros es una suma de variables negadas o no. Estas serán las reglas que utilizaremos a la hora de simplificar por ceros (sin necesidad ya de dibujar el Mapa de Karnaugh de la función  $\overline{F}$ ).

### Salidas no determinadas

Hasta ahora hemos tratado con funciones lógicas que, para todos las combinaciones posibles a la entrada ( $2^n$ , siendo  $n$  el número de variables de entrada) toman un valor binario determinado en la salida, ya sea 0 o 1. Sin embargo, existen situaciones en las cuales queremos diseñar un circuito en el cual no vamos a utilizar todas las condiciones de entrada, sino que algunas de ellas van a quedar sin utilizar. En ese caso, decimos que a aquellas combinaciones de entrada para las cuales no nos importa la salida (puesto que, en su operación normal, al circuito nunca le presentaremos esas entradas) les corresponde una salida no determinada. Se indicarán en la tabla de verdad y mapa de Karnaugh correspondientes colocando una X, en lugar de un 0 o un 1.

Un ejemplo de utilización se expone a continuación: Imaginar que queremos simplificar una función booleana de 4 variables de entrada: A, B, C, D, y una de salida: F(A, B, C, D), que valga uno cuando la ristra de 4 bits de entrada, interpretada en binario natural, esté entre 4 y 8 (ambos inclusive), valga cero para el resto de casos entre 0 y 9 (0, 1, 2, 3, 9) y que nos de igual la salida si está entre 10 y 16. La tabla de verdad correspondiente sería la indicada en tabla 2.

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

Las condiciones de “no importa” (X) pueden ser utilizadas en la simplificación de la función, con la ventaja de que, como nos da igual que la salida efectiva sea 0 o 1 en la función simplificada, podemos decidir cubrir las X o no en el mapa de Karnaugh correspondiente. La únicas obligaciones que tenemos son:

- Que se cubran todos los unos.
- Que no se cubra ningún cero.

Las X pueden utilizarse para hacer los grupos de unos más grandes (así el producto obtenido tendrá menos variables), o bien simplemente dejarse sin cubrir. En el ejemplo que nos ocupa, una solución sería

ENTRADAS				SALIDA
A	B	C	D	F
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	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Tabla 2: Tabla de verdad correspondiente a la función F con salidas indeterminadas.

$F(A, B, C, D) = B + A \cdot \overline{D}$ . Obsérvese como las X de los minterminos 12, 13, 14 y 15 se aprovechan para hacer un grupo de 8 unos, correspondiente al implicante primo B, y los términos 10, 12 y 14 se aprovechan para formar un grupo de 4 unos, correspondiente al implicante primo  $A \cdot \overline{D}$ . La X del mintermino 11, sin embargo, no se cubre en este caso, lo que significaría que la salida de la función simplificada obtenida valdría cero para esa combinación de entrada (la 1011). En general, las condiciones de no importa suelen resultar bastante útiles para obtener mayores simplificaciones.

## 1.2. Circuitos combinacionales comunes

En este apartado estudiaremos los circuitos combinacionales, que son aquellos en los que la salida en cada instante depende únicamente del valor de las entradas en ese instante, sin importar el comportamiento anterior del circuito. Como veremos posteriormente, la diferencia con el otro tipo básico de circuitos, llamados circuitos secuenciales, radica en que en estos últimos el valor de la salida no dependerá sólo del valor de las entradas en ese instante, sino también de la historia previa del circuito. (por ello se dice que ese otro tipo de circuitos poseen memoria).

Veamos a continuación algunos de los circuitos combinacionales más importantes, comenzando por las puertas lógicas.

### 1.2.1. Puertas lógicas básicas

Los bloques lógicos se construyen a partir de puertas lógicas que implementan las funciones lógicas básicas, y que se utilizan como elementos constructivos de partida para implementar la mayor parte de las funciones que desempeña la circuitería del ordenador. Por ejemplo, una puerta AND implementa el producto lógico y una puerta OR implementa la suma lógica. Como AND y OR son conmutativas y asociativas, una puerta AND o una puerta OR pueden tener múltiples entradas, siendo la salida igual al producto lógico o suma lógica de todas sus entradas. La negación lógica se implementa con una puerta NOT, llamada también inversor, que tiene una única entrada. La representación simbólica de estas tres puertas lógicas básicas puede verse en la siguiente figura:

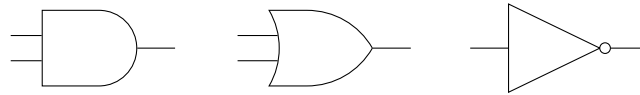


Figura 4: Símbolo de las puertas lógicas básicas (AND, OR, NOT)

En lugar de dibujar los inversores explícitamente, una práctica común es añadir burbujas a las entradas o salidas de una puerta para lograr que el valor lógico en la línea de entrada o en la línea de salida se invierta. Por ejemplo, la figura 5, muestra el diagrama lógico para la función  $F = \overline{(A + B)}$ , utilizando inversores explícitamente a la izquierda y utilizando entradas y salidas con burbujas a la derecha.

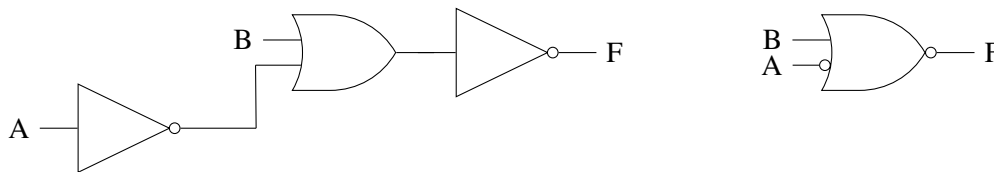


Figura 5: Implementación con puertas lógicas de  $F = \overline{(A + B)}$ , utilizando explícitamente inversores (izq.) y utilizando entradas y salidas con burbujas (der.).

Cualquier función lógica puede ser construida utilizando puertas AND, puertas OR e inversores. Por otro lado, es posible realizar todas las funciones lógicas utilizando un solo tipo de puertas, si esa puerta es inversora. Las dos puertas anteriores invertidas se denominan NAND y NOR. Las puertas NAND y NOR se denominan universales, ya que cualquier función lógica puede ser construida utilizando este tipo de puertas. Como ejemplo, veamos cómo puede expresarse la función  $F(A, B) = A + B$  usando sólo la operación NAND (AND negado):

$$A + B = (\text{por involución}) = \overline{\overline{(A + B)}} = (\text{por Ley de DeMorgan}) = \overline{(\overline{A} \cdot \overline{B})}$$

donde las respectivas negaciones de A y B pueden hacerse con una puerta NAND de sólo una entrada, y el posterior producto negado con una NAND de dos entradas. Análogamente, pueden encontrarse equivalencias para cada una de las puertas AND, OR y NOT sólo en función de NAND o NOR (se propone como sencillo ejercicio), con lo que quedaría demostrada su universalidad. En la figura 6 se detalla la tabla de verdad y el símbolo lógico de cada una de las puertas mencionadas.

Tan sólo comentar el caso de la operación de OR exclusivo (XOR), que se define de manera funcional como  $XOR(A, B) = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$ . Como vemos, la salida es 1, si y sólo si, sus entradas son diferentes. El OR exclusivo recibe su nombre debido a su relación con la puerta OR. Ambas difieren en la combinación de entradas  $A = 1$  y  $B = 1$ . El OR exclusivo da como salida 0, mientras que la puerta OR produce un 1, por lo que se le llama OR inclusivo. Esta operación cumple las propiedades conmutativa y asociativa. En general, para cualquier número de entradas, la salida de una puerta OR exclusivo es la suma módulo dos de sus entradas. Es decir, la salida es 1 si el número de entradas que valen 1 es impar, y 0 en caso contrario.

### Circuitos integrados

Las puertas lógicas no se fabrican ni se venden individualmente, sino en unidades llamadas circuitos integrados o chips. Un chip no es más que una pieza de silicio rectangular sobre la que se depositan algunas puertas lógicas. Las pastillas pueden dividirse en varios grupos, en base al número de puertas que contienen:

- SSI (circuitos integrados a escala pequeña): 1 a 10 puertas.
- MSI (circuitos integrados a escala media): 10 a 100 puertas.

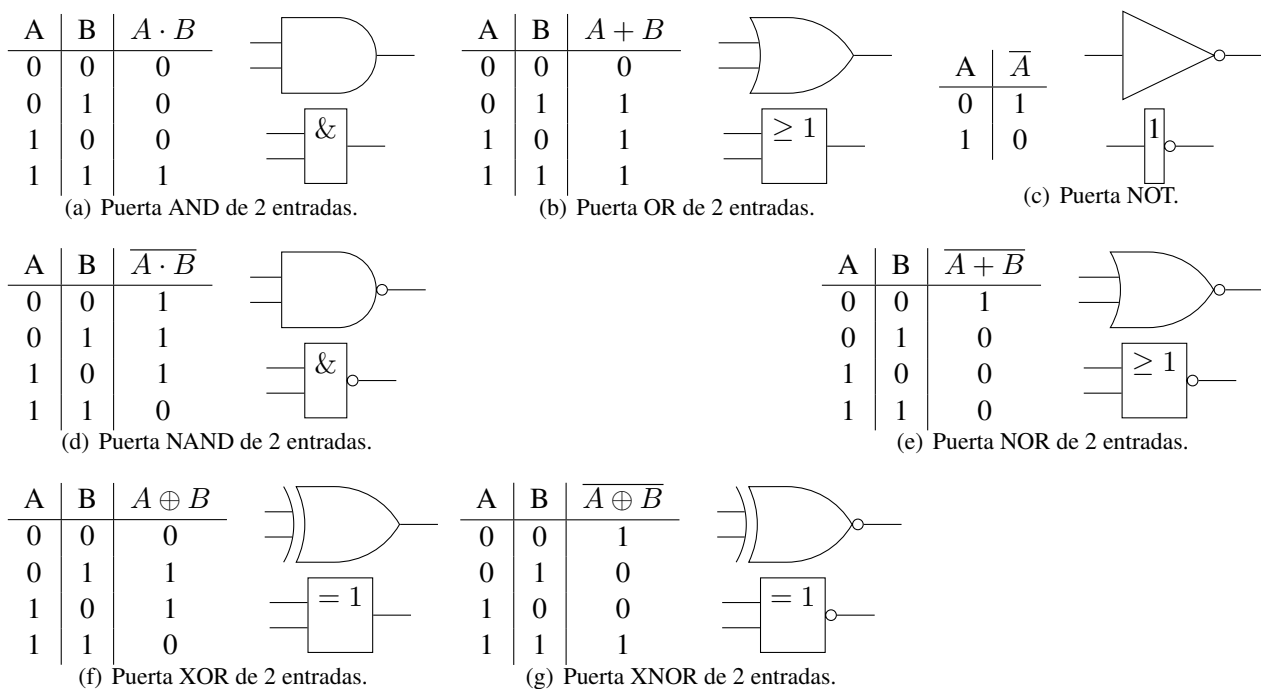


Figura 6: Símbolo y tabla de verdad de diferentes puertas lógicas.

- LSI (circuitos integrados a escala grande): 100 a 100.000 puertas.
- VLSI (circuitos integrados a escala muy grande): Más de 100.000 puertas.

donde SSI, MSI, LSI y VLSI son las iniciales, respectivamente, de *Short, Medium, Large* y *Very Large Scale of Integration*. En la figura 7 podemos ver diferentes pastillas SSI que implementan algunas de las puertas lógicas básicas que hemos estudiado hasta ahora.

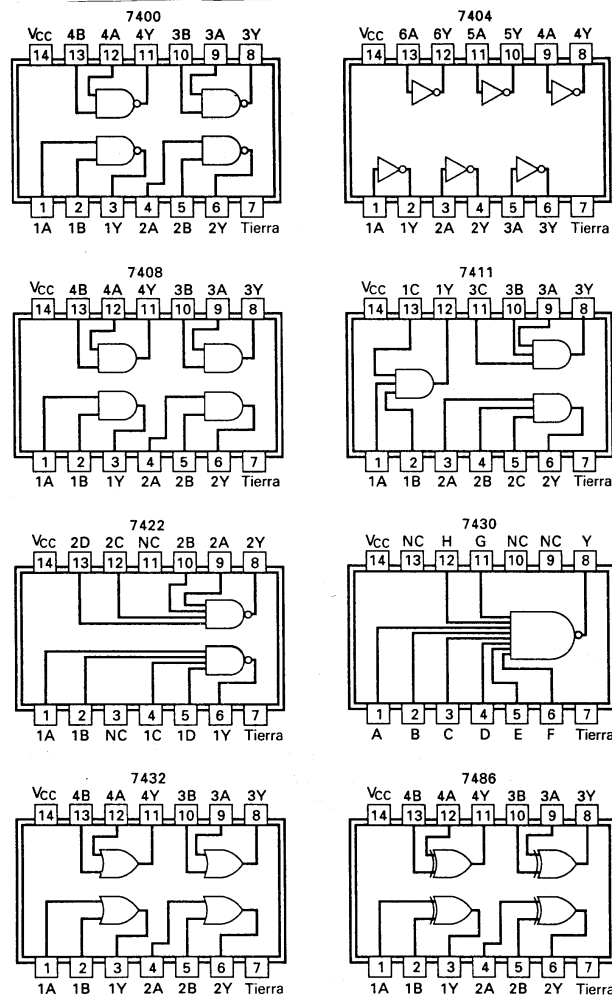
### 1.2.2. Retardos

Puesto que a la hora de implementar funciones con puertas, en última instancia éstas son dispositivos físicos reales, existen una serie de restricciones sobre el funcionamiento de cada puerta, de las cuales la más importante es el retardo temporal que introducen. Este retardo se puede definir como el tiempo que transcurre entre el instante en que un circuito tiene disponibles los valores de señal deseados a la entrada y el instante en que la señal de salida se estabiliza al valor deseado. Por ejemplo, si decimos que las puertas AND y OR tienen un retardo de 10 ns (nanosegundos), las NOT de 5 ns, y los retardos introducidos por las conexiones son despreciables, el circuito de la figura 8 tendrá un retardo total de 25 ns (puesto que la salida del NOT se obtendrá en 5 ns, la del AND superior en 5+10=15 y la del OR en 5+10+10=25 ns, mientras que el AND inferior funcionará en paralelo al superior y tardará sólo 10 ns en efectuar su operación).

### 1.2.3. Implementación de funciones lógicas con puertas NAND/NOR

Una de las características de las puertas NAND (NOR) es su carácter de puerta *universal*. Es decir, es posible implementar cualquier función lógica utilizando únicamente puertas NAND (NOR). La demostración es sencilla, basta con demostrar cómo se implementan las puertas AND, OR y NOT utilizando únicamente puertas NAND (NOR). La figura 9 muestra la implementación de las puertas AND, OR y NOT utilizando únicamente puertas NAND (la implementación con puertas NOR sería similar).

Una vez vista la universalidad de ambas puertas veamos cómo podemos implementar una función lógica utilizando únicamente puertas NAND (NOR). La situación más sencilla es la implementación con puertas



Algunas pastillas SSI. Disposiciones de patas extraídas de *The TTL Data Book for Design Engineers* (© D.R. por Texas Instruments Incorporated, 1976).

Figura 7: Algunas pastillas SSI.

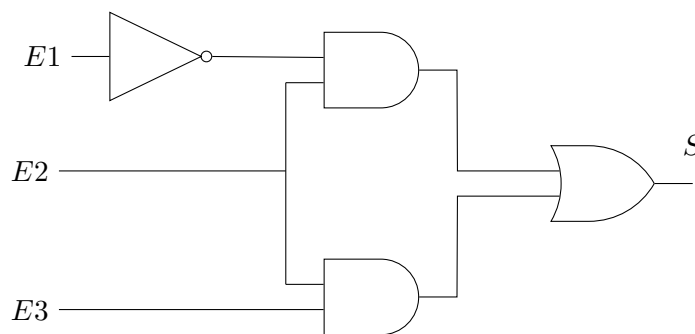


Figura 8: Ejemplo de retardo total de un circuito.

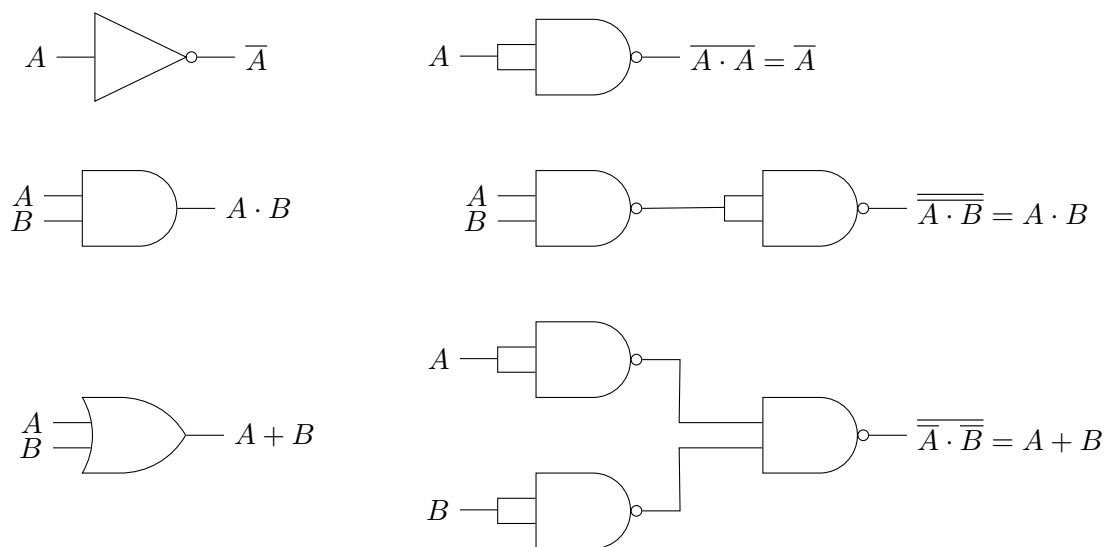


Figura 9: Equivalencia entre las puertas AND, OR y NOT y la puerta NAND.

NAND ya que una vez obtenida la función mínima expresada en forma de suma de productos (suma de minitérminos) sólo debemos negar la expresión dos veces y aplicar las leyes de De Morgan a la negación interior para obtener una expresión en donde sólo aparezcan puertas NAND. Así, por ejemplo, dada la función  $F(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$  cuya minimización ya vimos que era:

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

si a continuación negamos dos veces y aplicamos De Morgan obtenemos:

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

que puede implementarse directamente mediante puertas NAND.

En el caso de querer implementar la misma función utilizando puertas NOR sería conveniente obtener una expresión de la función en forma de producto de sumas (producto de maxitérminos), ya que, una vez que estemos en dicha situación, sólo necesitaremos repetir el proceso anterior (negar la expresión dos veces y aplicar De Morgan a la negación interior) para obtener de forma directa la expresión deseada. Por tanto, simplificaremos agrupando los ceros de la función para obtener una expresión en producto de maxitérminos. El resultado obtenido, como ya vimos, es:

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

Si a continuación negamos dos veces obtenemos y aplicamos De Morgan obtenemos:

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

que puede implementarse directamente mediante puertas NOR.

### 1.2.4. Bloques lógicos

Conforme construimos funciones lógicas cada vez más complejas (involucrando varias operaciones AND, OR, NOT, etc, en distintas etapas), se hace inviable representar gráficamente el diagrama de conexión completo con todas las puertas resultantes. Además, en general, el método de diseño de circuitos será jerárquico, es

decir, iremos construyendo sencillos circuitos capaces de realizar alguna tarea simple, para después, utilizando estos bloques elementales, construir circuitos cada vez más complejos. Si tenemos clara la función y estructura interna de cada bloque, no hace falta dibujar todas las puertas que componen el mismo, sino solamente un cuadro con el nombre del bloque, y sus entradas y salidas, como se muestra en la figura 10.

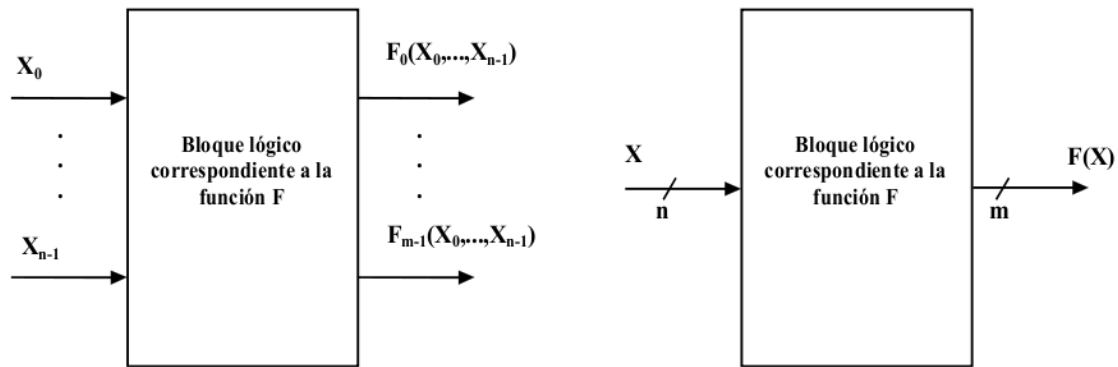


Figura 10: (Izquierda) Bloque lógico correspondiente a una función de  $n$  bits de entrada y  $m$  de salida. (Derecha) A veces, si las entradas y/o salidas están relacionadas entre sí, formando una secuencia de un determinado número de bits ( $n$  y  $m$  en la figura), emplearemos la abreviatura gráfica mostrada.

En los siguientes apartados, donde se construyen algunos circuitos básicos con diversas utilidades, y que serán empleados posteriormente en el diseño de un procesador, se verán numerosos ejemplos del uso de estos bloques para simplificar los diagramas obtenidos.

### 1.2.5. Codificadores y decodificadores

Un **codificador** es un circuito lógico combinacional con  $2^n$  líneas de entrada y  $n$  líneas de salida, tal y como se muestra en la figura 11 (en este ejemplo hay 8 entradas y 3 salidas). Cuando una de las entradas es activada y el resto se dejan a cero, en las  $n$  líneas de salida aparece el número de entrada activa codificado en binario. Por ejemplo, si se activa la entrada E3, la salida será  $(S2, S1, S0)=(0, 1, 1)$ , puesto que 011 en binario es 3. La generalización para construir un codificador con otro número de entradas es trivial, conectando cada entrada  $n$  a una puerta OR de salida si y sólo si el bit que ocupa la posición correspondiente a dicha puerta de salida está a uno al expresar  $n$  en binario.

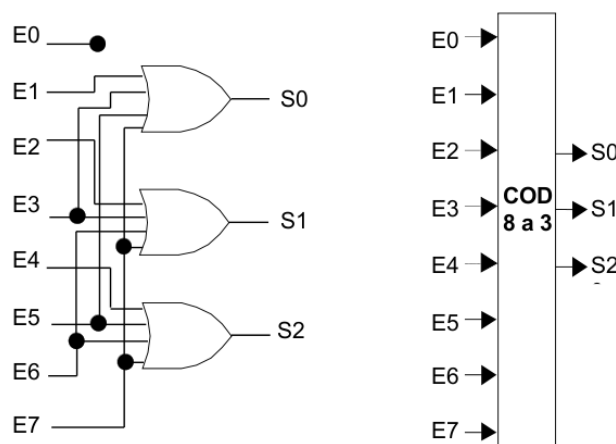


Figura 11: Implementación con puertas y diagrama lógico equivalente de un codificador de 8 a 3 bits.

La implementación anterior sólo permite que una de las entradas esté activa simultáneamente, como se

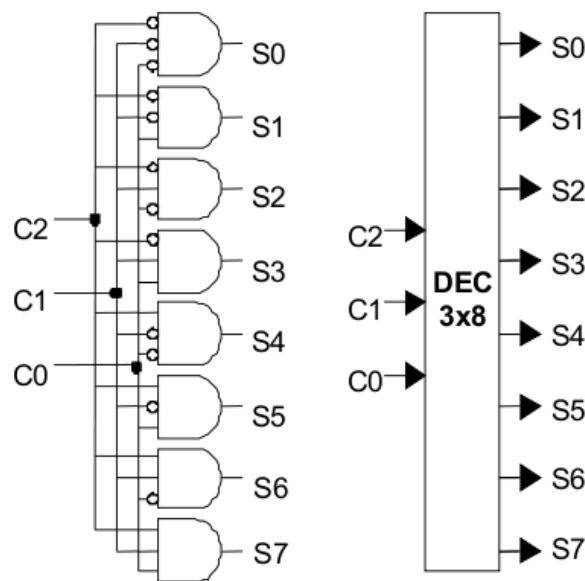


Figura 12: Diagrama funcional de un decodificador 3 a 8.

indica en la siguiente tabla de verdad:

E0	E1	E2	E3	E4	E5	E6	E7	S2	S1	S0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Inversamente, un **decodificador** es un circuito lógico combinacional con  $n$  líneas de entrada y  $2^n$  líneas de salida, tal y como se muestra en la figura 12.

En este caso, interesa que para cada posible valor de las líneas de entrada, una y sólo una de las señales de salida tenga el valor lógico 1. Por tanto, podemos considerar el decodificador  $n$  a  $2^n$  como un generador de minterminos, donde cada salida corresponde precisamente a un mintermino diferente. Si en la entrada, por ejemplo, colocamos la secuencia 110, entonces S6 valdrá 1, mientras que el resto de salidas valdrá 0. Al igual que ocurría con los codificadores, la extensión a mayor número de entradas es inmediata, mediante la simple adición de una puerta AND por cada salida  $n$ , a la cual se conectan todas y cada una de las entradas  $C_i$ , negadas si al escribir  $n$  en binario el bit  $i$  es 0, y sin negar en caso contrario.

Los decodificadores se usan para tareas tales como seleccionar una palabra de memoria, dada su dirección (lo veremos en el tema siguiente) o convertir códigos, por ejemplo, de binario a decimal. La siguiente tabla de verdad resume el funcionamiento del anterior decodificador:

C2	C1	C0	S0	S1	S2	S3	S4	S5	S6	S7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Tal y como se indicó anteriormente, un decodificador de  $n$  a  $2^n$  se puede ver como un generador de minterminos, donde cada salida corresponde precisamente a un mintermino diferente. Así, podemos utilizar un decodificador de  $n$  entradas junto a una puerta OR del tamaño adecuado para implementar cualquier función booleana de  $n$  variables. Por ejemplo, si quisiéramos implementar la función de cuatro variables:

$$F(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$$

sólo tendríamos que utilizar un decodificador de 4 a 16, conectando las 4 variables a las 4 entradas del decodificador y conectando a la puerta OR las salidas 2, 3, 4, 5, 7, 8, 10, 13 y 15 del multiplexor (ver figura 13).

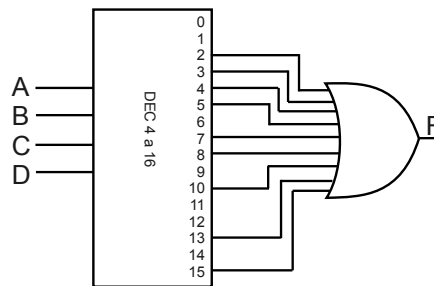


Figura 13: Implementación de una función booleana mediante el uso de un decodificador.

### 1.2.6. Multiplexores

Un multiplexor, también llamado selector de datos, es un dispositivo que selecciona una de varias líneas de entrada para que aparezca en una única línea de salida. La entrada de datos que se selecciona viene indicada por unas entradas especiales, llamadas de control. La figura 14 (izquierda) muestra el esquema del multiplexor más sencillo, de  $2 \times 1$  (dos entradas y una salida), con dos entradas de datos y una de control, que selecciona cuál de las dos entradas de datos queremos que se transfiera a la salida. Siguiendo la misma técnica que con los decodificadores, es fácil extender los multiplexores para mayor número de entradas. La figura 14 (derecha) muestra un multiplexor  $4 \times 1$ .

En general, en un multiplexor con  $n$  líneas de entrada y una única salida, se determina cuál de las líneas de entrada ( $D_{n-1}D_{n-2}D_{n-3} \dots D_1D_0$ ) se conecta a la única línea de salida (Y) mediante un código de selección ( $S_{k-1}S_{k-2}S_{k-3} \dots S_1S_0$ ) donde necesariamente  $n = 2^k$ . Así, por ejemplo, para un multiplexor con  $n = 4$  (y por tanto  $k = 2$ ), la ecuación lógica de la salida Y queda:

$$Y = (S1' \cdot S0') \cdot D0 + (S1' \cdot S0) \cdot D1 + (S1 \cdot S0') \cdot D2 + (S1 \cdot S0) \cdot D3$$

Además de en el número de entradas, otra extensión interesante de los multiplexores es en la anchura de los datos seleccionados. Así, en general hablaremos de un multiplexor de  $2^n \times 1$  de  $m$  bits de anchura, si es capaz de seleccionar entre  $2^n$  palabras de  $m$  bits cada una en función de una señal de control de  $n$  bits. En la figura 15 se muestra cómo crear un multiplexor  $2 \times 1$  de palabras de 32 bits, a partir de multiplexores

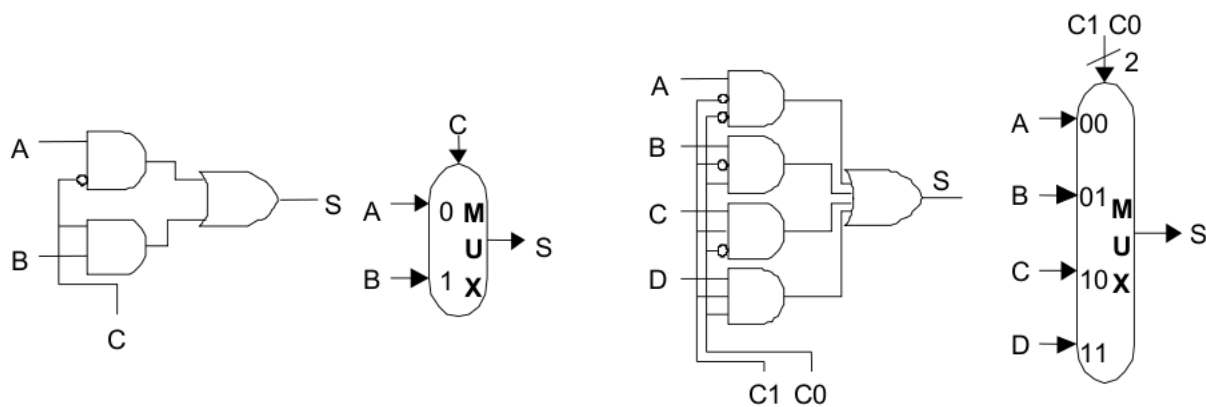


Figura 14: Implementaciones y bloques lógicos de dos tipos de multiplexores de un bit de ancho: (Izquierda) Multiplexor 2x1, con entradas de datos A y B y de control C. (Derecha) Multiplexor 4x1, con entradas de datos A, B, C y D y de control C1 y C0.

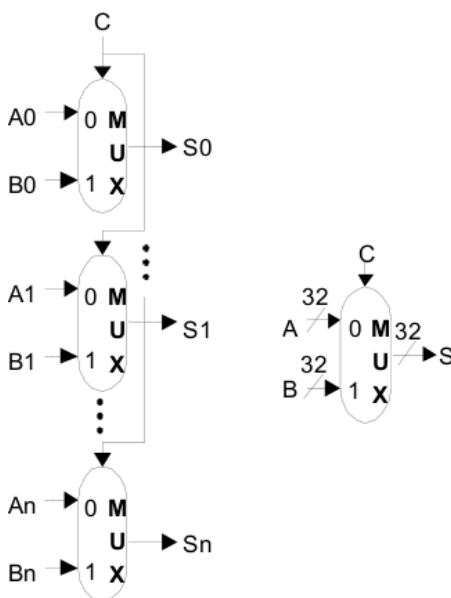


Figura 15: Implementación de un MUX 2x1 para palabras de 32 bits, y bloque lógico correspondiente. Obsérvese que es necesario sólo un bit de control para el multiplexor.

elementales de 1 bit. Usando esta misma técnica pueden conseguirse multiplexores con la anchura y el número de entradas deseadas.

Un multiplexor con  $n$  bits de control puede servir también para implementar una función cualquiera con  $n$  entradas, simplemente conectando las entradas de la función a las entradas de control del multiplexor, y las entradas de datos de éste a unos y/o ceros, según la tabla de verdad de la función. Así, la implementación de la función:

$$F(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$$

utilizando un multiplexor de 16 a 1 sería la mostrada en la figura 16.

Por otro lado, también es posible implementar funciones de más de  $n$  variables utilizando un multiplexor con  $n$  bits de control. En este caso elegimos  $n$  de las variables para ser conectadas a las  $n$  entradas de control del multiplexor. El resto de variables aparecerán en las entradas de datos en forma de función booleana. Veamos el ejemplo concreto en donde utilizamos un multiplexor con  $n$  bits de control para implementar una función de  $n + 1$  variables.

Para la función implementada en la figura 16, el multiplexor sería ahora de 8 a 1 (tres variables de control). Debemos seleccionar qué variables vamos a utilizar como control y cuál se quedará fuera. Supongamos que las variables de control elegidas son A, B y C, dejando fuera la variable D. Entonces cada entrada de datos del multiplexor depende del contenido de dos casillas adyacentes en el Mapa de Karnaugh. Así, la entrada de datos 0 debe mostrar los valores de las casillas 0 y 1, la entrada 1 los valores de las casillas 2 y 3, etc. En la figura 17(izq) se puede observar como quedan agrupadas las casillas en el Mapa de Karnaugh.

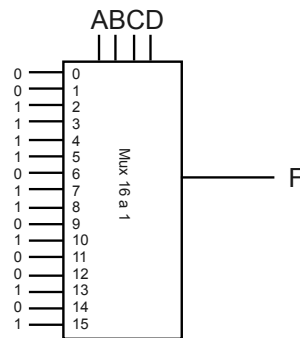


Figura 16: Implementación de una función booleana de 4 variables mediante el uso de un multiplexor 16 a 1.

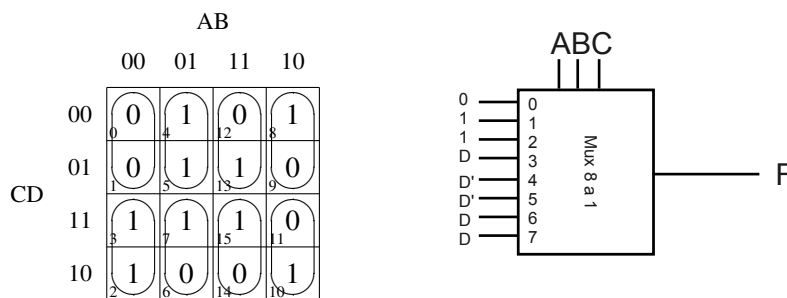


Figura 17: Implementación de una función booleana de 4 variables mediante el uso de un multiplexor 8 a 1 ( $D$  es la variable excluida).

Con cada entrada representando a dos casillas existen cuatro posibles opciones que determinan los valores que pondremos en cada una de las entradas del multiplexor. Si en ambas casillas encontramos un 0, la entrada correspondiente será 0. De igual manera si encontramos un 1, asignaremos un 1 a esa entrada. Por último, si las casillas tiene valores distintos procederemos de la siguiente manera. Observaremos el valor de la variable

excluida en esa agrupación de casillas. Si cuando dicha variable vale 0 la casilla contiene un 0 y cuando vale un 1 la casilla contiene un 1, conectaremos esa entrada del multiplexor con la variable excluida ( $D$  en nuestro caso). Si la situación es la inversa, asignaremos a dicha entrada la variable excluida negada (el valor  $\bar{D}$  en nuestro caso). En la figura 17(der) se puede observar la implementación final de la función  $F$ .

### 1.2.7. Memorias ROM y arrays lógicos programables

#### Memorias ROM

La ROM (*Read Only Memory*, memoria de sólo lectura) es una forma de lógica combinatorial estructurada que sirve para implementar un conjunto de funciones lógicas. Una ROM se llama memoria porque contiene un conjunto de posiciones que pueden ser leídas; sin embargo, el contenido de estas posiciones es fijo, habitualmente desde el instante en que se crea la ROM. Así, la salida del circuito ante una entrada determinada no depende de su evolución anterior (como ocurrirá en una memoria RAM), sino que depende sólo de la entrada (esto es, la dirección) que se le proporcione en un momento dado. Se trata, pues, de un circuito combinatorial.

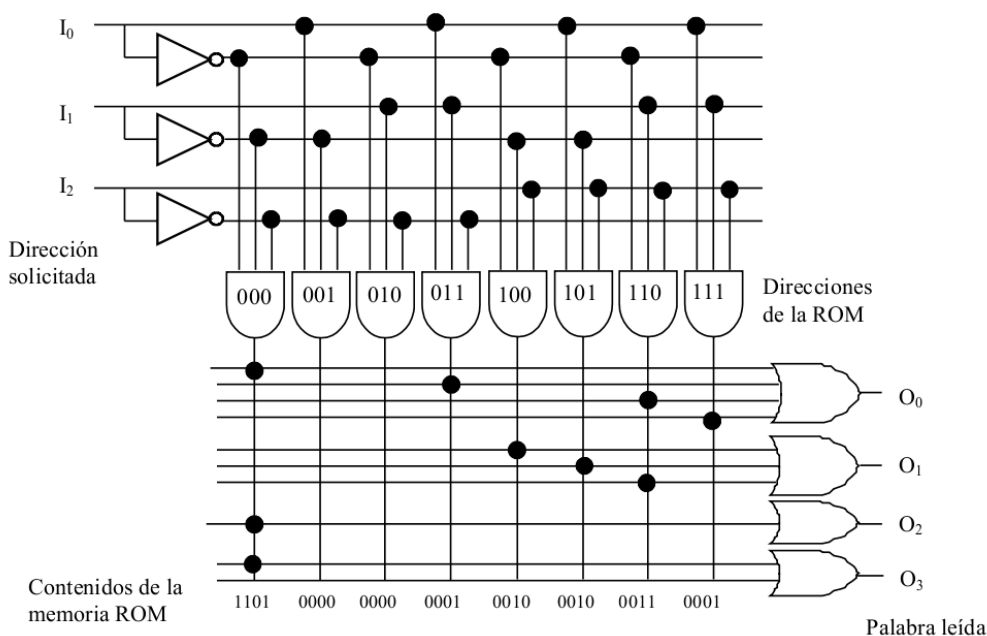


Figura 18: Esquema de una memoria ROM con 8 posiciones de 4 bits cada una (3 bits de dirección, y anchura de datos 4).

Una ROM tiene un conjunto de líneas de entrada de dirección y un conjunto de salidas. El número de elementos direccionables de la ROM determina el número de líneas de dirección: si la ROM contiene  $2^m$  elementos direccionables, o altura de la ROM, entonces hay  $m$  líneas de dirección, o sea,  $m$  líneas de entrada. El número de bits de cada elemento direccionable es igual al número de bits de salida y a veces se denomina anchura de la ROM. El número total de bits de la ROM es igual a la altura por la anchura. La altura y la anchura a veces se denominan forma de la ROM.

Una ROM puede implementar directamente una colección de funciones lógicas a partir de la tabla de verdad. Por ejemplo, si hay  $n$  funciones con  $m$  entradas, necesitamos una ROM con  $m$  líneas de dirección y  $2^m$  elementos, teniendo cada uno  $n$  bits de ancho. Las entradas de la parte de entrada de la tabla de verdad representan las direcciones de los elementos contenidos en la ROM, mientras que la parte correspondiente a las salidas de la tabla de verdad constituye el contenido de la ROM para cada dirección. Si se organiza la tabla de verdad para que la secuencia de entradas de la tabla de verdad sea la secuencia de números binarios (tal y

como hemos hecho en las tablas de verdad hasta ahora), entonces la parte de salida da también el contenido de la ROM en orden.

El esquema de implementación de una ROM consta de dos niveles de puertas, o planos, a los cuales se les llama plano AND y plano OR, por el tipo de puertas que se utilizan en cada nivel. En general, una ROM de  $m$  bits de entrada (dirección) y  $n$  bits de anchura (es decir, capaz de guardar  $2^m$  palabras de  $n$  bits cada una), tendrá un total de  $2^m$  puertas AND, cada una con  $m$  entradas, y  $n$  puertas OR, cada una con, como máximo,  $2^m$  entradas. El plano de conexión AND será siempre el mismo para todas las ROM con la misma capacidad, mientras que el OR dependerá de los contenidos concretos de la memoria de sólo lectura que estamos construyendo. La figura 18 muestra la construcción de una ROM de ejemplo  $2^3 \times 4$  (8 posiciones de 4 bits), que implementa la siguiente tabla de verdad:

I2	I1	I0	O3	O2	O1	O0
0	0	0	1	1	0	1
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	1
1	0	0	0	0	1	0
1	0	1	0	0	1	0
1	1	0	0	0	1	1
1	1	1	0	0	0	1

## PROM, EPROM, EEPROM

La memoria programable exclusiva para lectura (PROM) es similar a la memoria ROM pero con la particularidad de que el plano OR es programable. El plano AND, al igual que en la memoria ROM, genera los  $2^m$  posibles minitérminos para sus  $m$  entradas. El plano OR permite incluir cualquier combinación de los términos producto en cada término suma. Por tanto, es posible construir cualquier función que esté en forma de suma canónica de productos.

Para hacer que el plano OR sea programable, en cada cruce de líneas del plano OR hay un fusible metálico. Un fusible intacto se comporta como un circuito cerrado conectando la línea correspondiente a la puerta OR. Si se funde tal fusible, pasando una corriente alta a través de él, se impide tal contacto. De este modo es posible programar la memoria PROM. En la figura 19 se muestra un ejemplo de PROM aún sin programar (obsérvese el uso abreviado en el diagrama de las líneas de entrada a las puertas de los planos AND y OR).

Durante el desarrollo de un circuito lógico, la información por almacenar en cada PROM experimenta cambios frecuentes hasta que el diseño ha sido depurado. Por desgracia, las PROM, al igual que las ROM, no se pueden modificar una vez programadas. En este caso, se utilizan las memorias programables exclusivas para lectura borrables (EPROM). El plano OR de una EPROM se programa con un voltaje de programación especial que hace que en determinadas celdas (cruce de líneas) se almacene carga. La presencia o ausencia de carga determina si la línea se conecta o no a la puerta OR correspondiente. Esta carga se puede eliminar al irradiar el circuito con luz ultravioleta a través de una ventana de cuarzo que tiene el chip, lo cual devuelve el plano OR a su condición inicial, es decir, sin programar. Este ciclo de borrado y programación puede repetirse hasta que el contenido sea correcto, lo que permite usar una única EPROM durante todo el desarrollo.

Una memoria programable exclusiva para lectura borrable eléctricamente (EEPROM) es similar a una EPROM en el sentido de que el plano OR se puede programar repetidas veces mediante cargas eléctricas. Sin embargo, en una EEPROM el borrado se hace eléctricamente, aplicando un voltaje especial al circuito. Esto permite borrar y programar un circuito sin tener que aplicarle una luz especial. Por tanto, las EEPROM son atractivas en aplicaciones en las que es posible que haya que modificar la información almacenada a menudo. Muchos dispositivos EEPROM permiten un borrado selectivo del circuito. También existe un tipo de dispositivos EEPROM, llamados memorias flash, que permiten el borrado de todo el circuito, o por bloques, sin necesidad de ser extraídas de la placa en la que se encuentran. En resumen, las memorias EPROM y EEPROM son más flexibles que las PROM y, por tanto, más caras por bit.

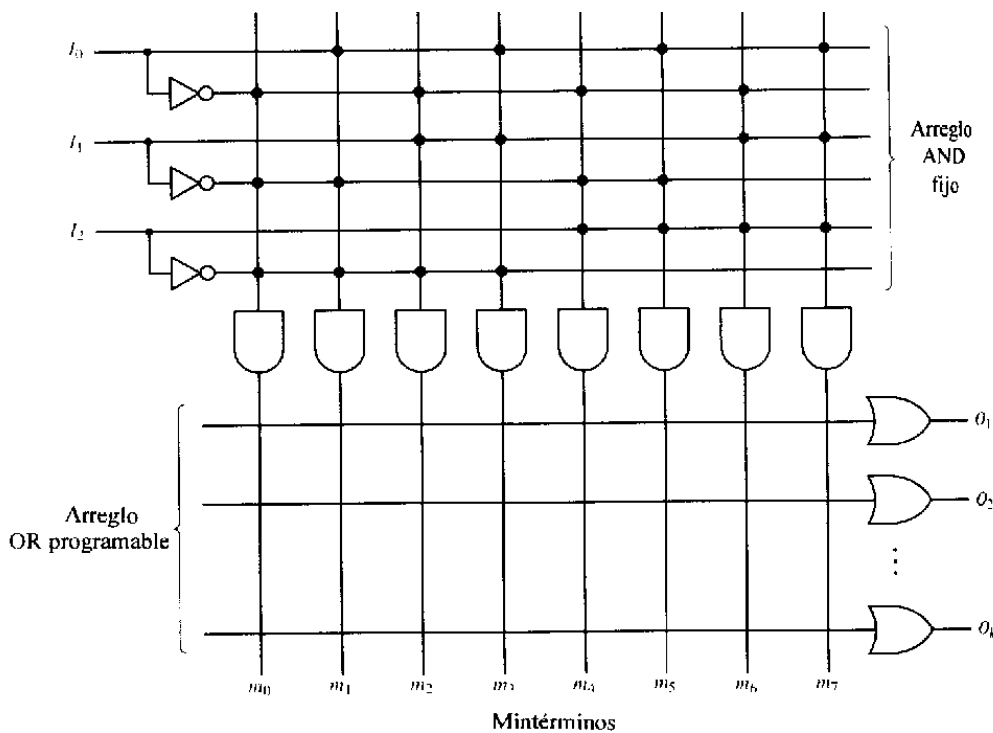


Figura 19: Esquema de una memoria programable exclusiva para lectura (PROM).

### PLA

Un array lógico programable (o PLA) es, de alguna manera, una generalización de una ROM. Al igual que aquella, tiene un conjunto de entradas con sus complementos correspondientes, a los que siguen dos etapas de lógica, una AND y una OR. En la primera etapa, cada puerta AND implementa un mintermino, mientras que en la segunda cada puerta OR forma una suma lógica de cualquier número de los minterminos realizados. Por tanto, un PLA también puede, como la ROM, implementar directamente la tabla de verdad de un conjunto de funciones lógicas con múltiples entradas y salidas.

La diferencia básica entre un PLA y una ROM radica en el plano AND. Mientras que, como vimos, en una ROM este plano es fijo e implementa todos los minterminos posibles ( $2^m$ , siendo  $m$  el número de entradas), en una PLA este plano es programable e implementa un número limitado de minterminos.

De esta forma, podemos ahorrar puertas AND si elegimos un PLA del tamaño adecuado, es decir, que tenga al menos tantas puertas AND como minterminos tenga la función que deseamos implementar. El tamaño de los PLAs se indica mediante una expresión de la forma  $I \times A \times O$ , donde  $I$  es el número de entradas,  $A$  el número de puertas AND empleadas, y  $O$  el número de salidas (que coincide con el de puertas OR).

Por ejemplo, para la función que implementamos en la ROM de la figura 18, se observaba que las filas correspondientes a los minterminos 001 y 010 tienen todas sus salidas nulas (0000). Así, en la implementación podemos ver que las correspondientes puertas AND no se utilizan. En este caso, podríamos implementar la función con un PLA de tamaño  $3 \times 6 \times 4$ , como se muestra en la figura 20.

En general, para funciones con más entradas, donde muchas de las combinaciones de entrada no se utilizan, es muy probable que existan muchos minterminos no utilizados.

### 1.2.8. Unidad aritmético lógica

Una ALU (de *Arithmetic Logic Unit*) es un circuito capaz de realizar operaciones aritméticas y lógicas sobre dos operandos de entrada, para producir como resultado el correspondiente operando de salida. La operación concreta a realizar vendrá dada por unos bits de control, con los que indicaremos a la ALU si

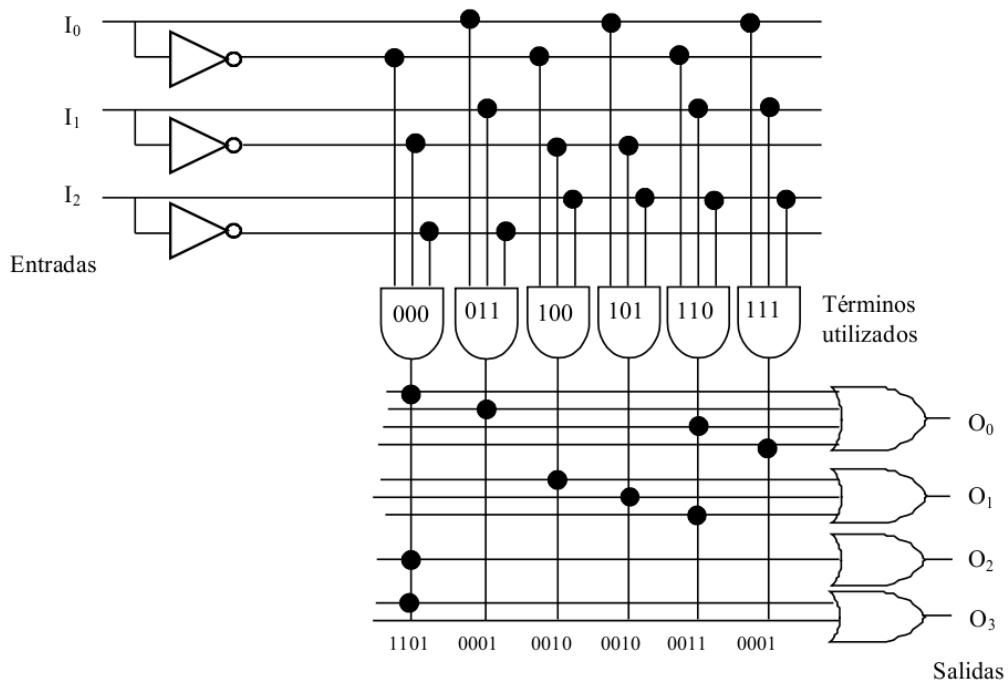


Figura 20: PLA equivalente a la ROM de la figura 18.

se desea realizar una suma, una resta, un AND lógico, etc. Hay que distinguir entre ALUs enteras y ALUs de punto flotante, dependiendo del tipo de datos con las que trabaje el circuito. En este curso estudiaremos únicamente el diseño de una sencilla ALU para trabajar con números enteros.

Cada uno de los operandos será una ristra de bits de una determinada longitud. En nuestro ejemplo trabajaremos con palabras de 32 bits, tamaño de palabra que, como veremos, utiliza el procesador MIPS que estudiaremos en temas posteriores. El bloque lógico de esta unidad sería el indicado en la figura 21.

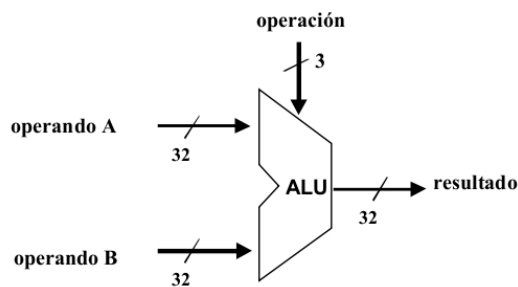


Figura 21: Bloque lógico de una ALU de 32 bits.

## Boletines de prácticas

### Normas sobre la entrega de boletines de prácticas

Será necesario seguir las siguientes reglas para entregar las prácticas:

- Las prácticas se entregarán mediante la opción de contenidos del alumno en SUMA.
- Se entregará un único archivo comprimido en formato *.tar.gz* o *.zip* que contendrá la memoria en formato PDF, los circuitos y programas que se hayan generado (código fuente) y cualquier otro fichero que se considere oportuno. El nombre del archivo será *prácticas-DNI-BOLETIN.FORMATO* (por ejemplo: *prácticas-12345678-B2.3.tar.gz*).
- La memoria incluirá, al menos, la siguiente información en un solo documento PDF:
  - Nombre y DNI del autor de la práctica.
  - Descripción de los ficheros y directorios contenidos en el archivo entregado.
  - Contestación a las preguntas planteadas en los boletines. La respuesta a cada pregunta debe ser independiente, y debe estar claramente identificada.
  - Explicación de las pruebas realizadas para comprobar la corrección de la práctica entregada e instrucciones para su reproducción. Cuando sea posible, se incluirán los ficheros utilizados en dichas pruebas.
  - Explicación del trabajo realizado y cualquier aclaración que el alumno considere pertinente.
  - Lista de bibliografía y otras fuentes de información consultadas.

No se corregirá ninguna práctica que no se ciña estrictamente a los formatos especificados anteriormente.

### B1.1. Uso del simulador TkGate

#### B1.1.1. Objetivos

Los objetivos de esta sesión son que el alumno se familiarice con el simulador TkGate y que sea capaz de crear y simular circuitos sencillos.

#### B1.1.2. Prerequisitos

No se establecen prerequisites para esta sesión de prácticas aunque sería deseable que el alumno hubiera leído los apuntes de teoría, en especial la sección 1.2.

#### B1.1.3. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de la sección B1.1.4.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

### B1.1.4. El simulador TkGate

TkGate es un programa para la creación y simulación de circuitos basado en el Lenguaje de Descripción de Hardware (HDL) Verilog. Puede ser descargado libremente de la dirección <http://www.tkgate.org>. TkGate incluye en su distribución una amplia documentación que está accesible a través del menú “Ayuda”. Dispone además de circuitos de ejemplo y de un tutorial al que dedicaremos esta sesión de prácticas.

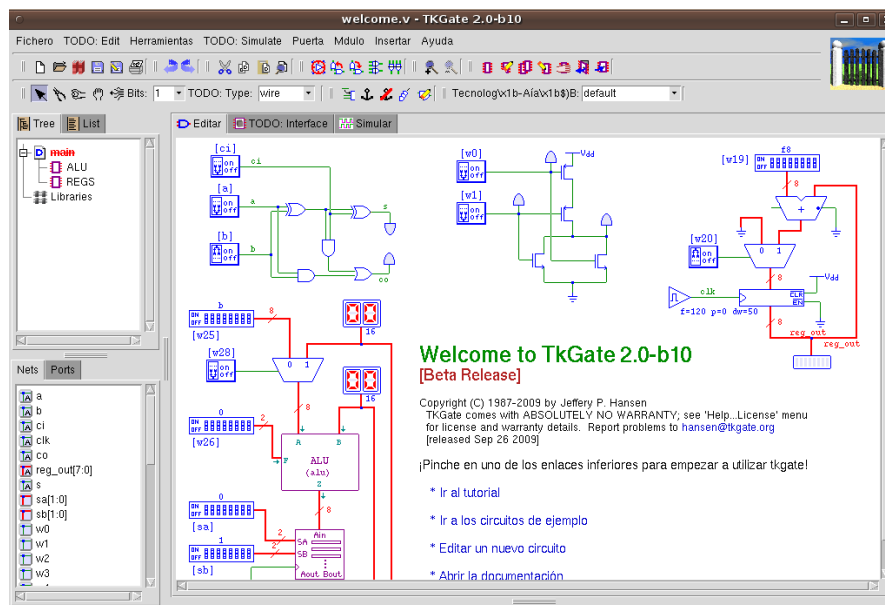


Figura B1.1: Aspecto de la pantalla principal de TkGate

Cuando se arranca el programa la ventana de TkGate tiene el aspecto que se puede ver en la figura B1.1. Como puede observarse en dicha figura, TkGate nos muestra cuando lo ejecutamos varias ventanas y paneles de información:

- Tenemos una área de trabajo principal dividida en tres pestañas: Editar, Interfaz y Simular. La primera pestaña nos permite la edición de un circuito mediante una interfaz gráfica y utilizando diversas herramientas. La pestaña de Interfaz nos permite trabajar con las interfaces de los módulos que están actualmente cargados. Por último, la pestaña de Simular permite la simulación de los circuitos, pudiéndose realizar simulaciones punto a punto, incluir Puntos de Ruptura (*BreakPoints*) y mostrar cronogramas como el que se puede observar en la figura B1.2.
- En la parte superior encontramos el menú principal de TkGate con las opciones típicas de manejo de ficheros, edición de circuitos, inserción de puertas, módulos, etc. Además del acceso directo a determinadas herramientas a través de botones.
- A la izquierda encontramos dos paneles de información que nos permiten visualizar los módulos cargados en un determinado momento (parte superior izquierda) y el nombre de los diferentes elementos que componen nuestro circuito (parte inferior izquierda).

### B1.1.5. Ejercicios

Pinche la opción “Ir al tutorial” en el menú principal del TkGate y siga las instrucciones del mismo hasta completarlo. En esta primera sesión de prácticas no es necesario introducir ninguna información en el portafolios pero se recomienda a los alumnos que practiquen con el simulador hasta familiarizarse con él de cara a las siguientes sesiones de prácticas.

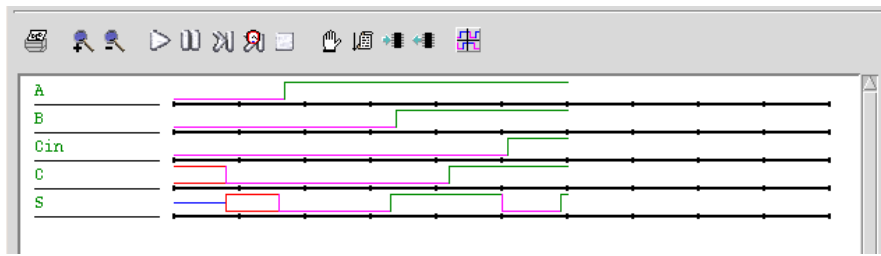


Figura B1.2: Aspecto de un cronograma en TkGate

## B1.2. Unidad aritmético lógica

### B1.2.1. Objetivos

En el segundo boletín de prácticas implementaremos uno de los circuitos esenciales dentro de un procesador: la **ALU** o unidad aritmético-lógica (*Arithmetic Logic Unit*).

El objetivo de la sesión es que el alumno implemente una ALU capaz de realizar operaciones sencillas: AND, OR, suma, resta y comparación con operandos de 32 bits y utilizando para ello el programa *TkGate* mediante un diseño modular. El resultado final será una ALU sencilla que utilizaremos en temas posteriores.

### B1.2.2. Prerequisitos

- Lectura de los apuntes de teoría, en especial la sección 1.2. En la sección 1.2.8 se explica el funcionamiento del módulo que se diseña en esta práctica.

### B1.2.3. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de las secciones B1.2.4 y B1.2.5.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

### B1.2.4. Implementación de una ALU de 1 bit

En lo que sigue diseñaremos una Unidad Aritmético Lógica de un bit utilizando para ello puertas lógicas elementales. Empezaremos implementando para ello las funciones lógicas AND y OR que se implementaría como se indica en la figura B1.3.

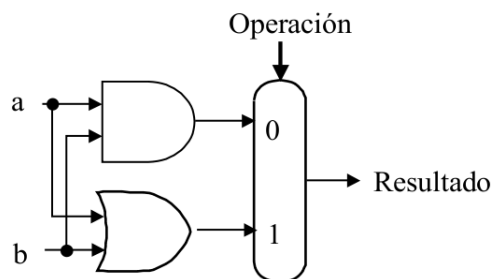


Figura B1.3: Unidad lógica de 1 bit con las operaciones lógicas AND y OR.

La línea más gruesa corresponde a una señal de control, la cual sirve para seleccionar en el multiplexor el resultado de la operación AND o OR, en función del valor de *Operación*. En este ejemplo, si *Operación* vale 0 se seleccionará la salida de la puerta AND, y si vale 1 la salida de la puerta OR.

Con lo anterior, nuestra ALU sólo realizaría operaciones lógicas. Vamos a aumentarla con la operación aritmética para la suma (como veremos, la misma circuitería servirá también para restar, usando el convenio de complemento a dos). El diagrama de bloques de un sumador de 1 bit tendría el aspecto indicado en la figura B1.4

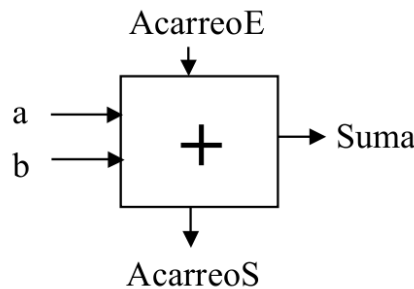


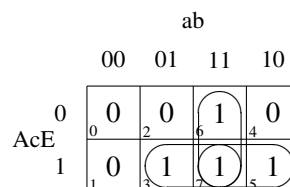
Figura B1.4: Representación de un sumador completo de 1 bit.

Este sumador, también conocido como *Full Adder* (Sumador Completo) tiene tres entradas, las dos correspondientes a los sumandos y una correspondiente al acarreo de salida del sumador anterior. Las dos salidas corresponden al resultado de la suma y al posible acarreo de salida que pudiera tener. Un sumador con sólo dos entradas y una salida se conocería como *Half Adder* (Semisumador). La relación entre las entradas y las salidas de un sumador total está especificado en la Tabla B1.1.

ENTRADAS			SALIDAS		FUNCIÓN
A	B	AcE	AcS	Suma	
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

Tabla B1.1: Tabla de verdad correspondiente al sumador completo.

La función de salida AcarreoS (AcS) puede simplificarse con un sencillo mapa de Karnaugh:



Y la función lógica, ya simplificada, quedaría pues así:

$$AcS = (b \cdot AcE) + (a \cdot AcE) + (a \cdot b)$$

En cuanto al bit de Suma, se activa cuando una y sólo una de las entradas valen 1 o cuando las tres valen 1, y su mapa de Karnaugh quedaría así:

		ab			
		00	01	11	10
0	AcE	0 <sub>0</sub>	1 <sub>2</sub>	0 <sub>6</sub>	1 <sub>4</sub>
1		1 <sub>1</sub>	0 <sub>3</sub>	1 <sub>7</sub>	0 <sub>5</sub>

Puesto que los grupos de unos no pueden hacerse más grandes, la ecuación booleana correspondiente sería:

$$Suma = (a' \cdot b' \cdot AcE) + (a' \cdot b \cdot AcE') + (a \cdot b' \cdot AcE') + (a \cdot b \cdot AcE)$$

El circuito lógico que implementa ambas funciones quedaría como se indica en la figura B1.5.

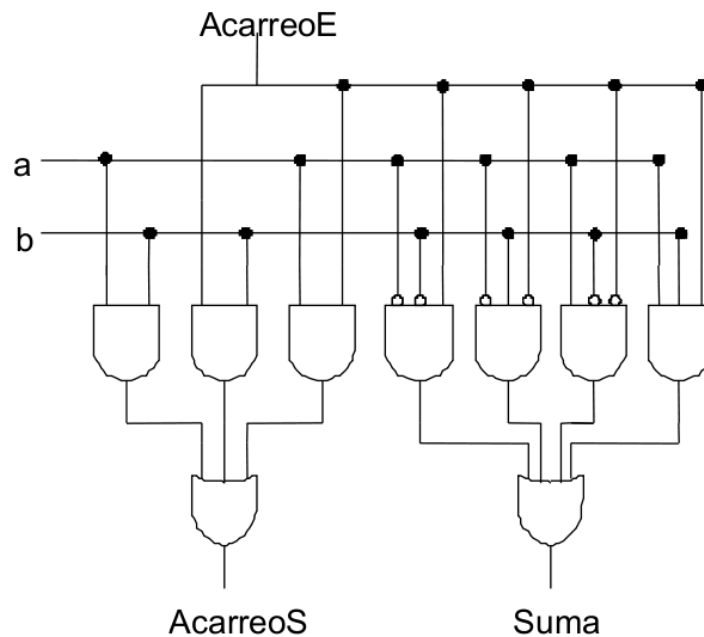


Figura B1.5: Sumador completo de 1 bit.

Otra posible solución para el bit de Suma sería el empleo de una simple puerta XOR de tres entradas (puesto que, como se observa de la tabla de verdad, la salida es uno cuando es impar el número de entradas iguales a uno):

$$Suma = (a \oplus b \oplus AcE)$$

Nos quedamos, sin embargo, con la primera implementación, puesto que no siempre es fácil la implementación física de una puerta XOR.

Una vez analizados cada uno de los componentes individuales de una ALU de 1 bit, sólo queda juntarlos en un sólo circuito, incorporando la lógica de control necesaria para seleccionar la operación lógica (OR o AND) o aritmética (SUMA) que se quiera utilizar. En este caso, la señal de entrada *Operación* necesita 2 bits para poder seleccionar una de las tres posibles operaciones, AND, OR o SUMA. En la figura B1.6 se muestra el resultado.

### B1.2.5. Implementación de una ALU de 32 bit

Construir una ALU sencilla (si bien un poco ineficiente) de 32 bits a partir de ALUs de 1 bit es tan sencillo como unir las cajas negras formadas por ALUs de 1 bit en cascada, de modo que el acarreo de salida

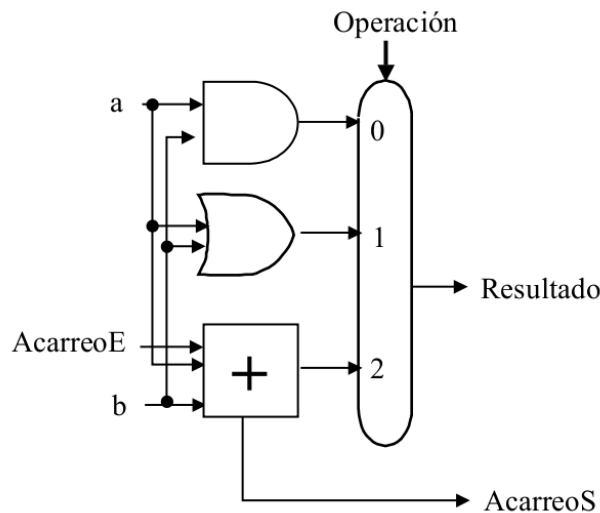


Figura B1.6: Unidad lógica de 1 bit con las operaciones AND, OR y Suma.

del sumador de cada una corresponda al acarreo de entrada de la siguiente. El sumador resultante se llama sumador con propagación del acarreo (*ripple carry adder*). La ineficiencia de este sumador radica en el hecho que para calcular el bit  $i$ -ésimo de la suma es necesario haber calculado los  $i - 1$  bits anteriores, es decir, el bit 1 se calcula después del 0, el 2 después del 1 y así sucesivamente hasta el bit 31. El retardo del circuito completo queda, pues, como el de cada sumador individual multiplicado por el total de bits a calcular. Existen modos de solucionar este problema, mediante los llamados circuitos con acarreo anticipado (*look ahead carry*). En estos circuitos, el acarreo de salida de cada sumador no tiene que esperar a atravesar tantos niveles de puertas como ocurre en el sumador con propagación. De todos modos, el estudio detallado de estos últimos queda fuera del alcance de este curso.

Una ALU de 32 bits formada a partir del encadenamiento de ALUs de 1 bit quedaría como se indica en la figura B1.7.

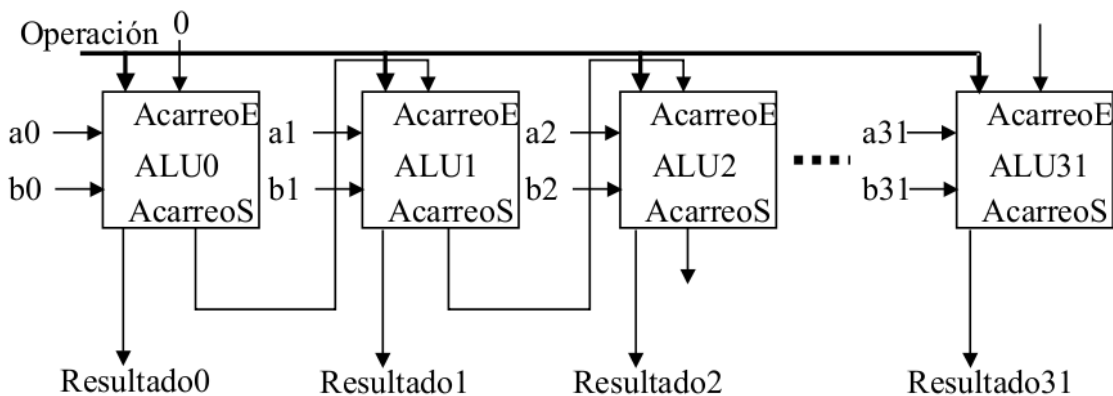


Figura B1.7: Unidad aritmético lógica de 32 bits con acarreo serie.

La resta se implementa utilizando la aritmética en complemento a dos y, por tanto, basta con sumar el complemento a dos del operando B para conseguir el resultado deseado. Como ya se vio en la asignatura de *Fundamentos de Computadores*, para obtener el complemento a dos basta con negar el número bit a bit y sumarle uno. Así, para invertir el segundo operando bit a bit se añade un multiplexor que seleccione entre  $b_i$  o  $\bar{b}_i$  en función de si la operación solicitada es la de sumar o la de restar. Para terminar de obtener el

complemento a dos, aún falta sumarle 1. Para ello, si utilizamos la ALU de 32 bits antes explicada, basta con poner a 1 el acarreo de entrada de la ALU0, es decir la de menos peso, que para el resto de operaciones siempre estará a 0. La ALU de un bit quedaría como se indica en la figura B1.8 (se muestra la ALU0, las demás tendrán como acarreo de entrada el acarreo de salida de la ALU anterior).

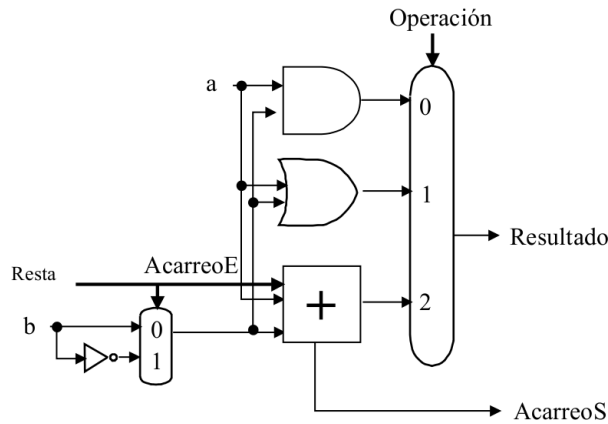


Figura B1.8: Unidad aritmético lógica de 1 bit, con posibilidad de restar (ALU0, bit de menos peso).

Nótese que si la entrada *Resta* vale 0, es decir, la operación no es una resta, el bit *b* no será invertido y el acarreo de entrada valdrá 0. Si el campo *Resta* vale 1, el bit *b* será invertido y el acarreo de entrada valdrá igual que *Resta*, es decir, 1.

Además de la suma y la resta, nos interesará implementar una operación para comparar la magnitud de los dos operandos, con el fin de saber cuál es mayor o menor. A esta operación la llamaremos *slt* (*set on less than*, activar si menor que), y devolverá un 1 si el primer operando fuente es menor que el segundo, y 0 en caso contrario (en realidad, puesto que la salida de la ALU es de 32 bits, devolverá 00...01 o 00...00, es decir, las constantes 0 y 1 en 32 bits). Para implementar la nueva operación hay que realizar unas pequeñas modificaciones en la ALU. Si se quiere saber si  $A < B$  basta con saber si la diferencia es negativa, es decir,  $A < B$  si  $A - B$  es menor que cero. Para ello, podemos realizar la resta de  $A - B$  y si el bit de signo, es decir, el bit más significativo de la ALU de 32 bits vale 1, significa que  $A$  es menor que  $B$ . La ALU31 quedaría como se indica en la figura B1.9.

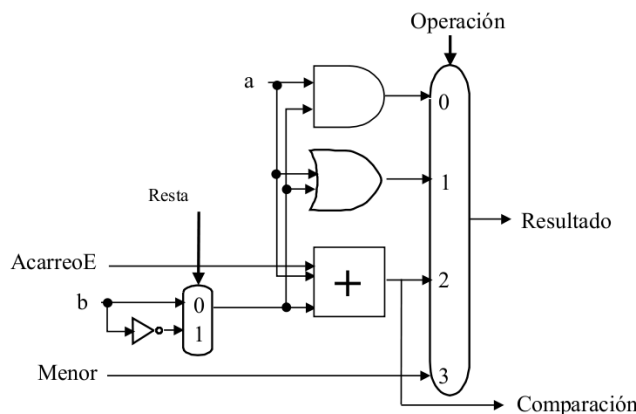


Figura B1.9: Unidad aritmético lógica de 1 bit, con posibilidad de restar y comparar.

La operación de comparación *slt* ha de devolver 00..01 en el bit menos significativo de la salida de 32 bits (Resultado0) si  $A=(a_{31}..a_0)$  es menor que  $B=(b_{31}..b_0)$ , y el resto de bits (Resultado31..Resultado1) han de quedar a 0. Pero como se ha explicado antes, para saber si  $A$  es menor que  $B$  se ha de comprobar el bit de

signo del resultado de la resta A-B. Por lo tanto, la ALU de 32 bits que realiza la comparación ha de conectar la salida del sumador de la ALU31 a la entrada *Menor* del multiplexor de la ALU0, dejando a 0 las entradas *Menor* del resto de las ALUs.

Finalmente, es interesante que nuestra ALU sea capaz de decir si sus dos operandos de entrada A y B son exactamente iguales. Para ello, basta con hacer la resta de los dos operandos y añadir una unidad de detección de 0 en el resultado: dos valores de 32 bits son iguales si su resta es igual a 00...00, y son diferentes en caso contrario. Una sencilla operación NOR sobre todos los bits del resultado será suficiente para esta tarea.

Antes de mostrar el esquema completo de la ALU de 32 bits, hay que tener en cuenta un posible problema que se puede presentar a la hora de realizar operaciones aritméticas. Este problema surge del hecho de que el conjunto de números enteros es infinito, mientras que no lo es la cantidad de bits destinados a representar dichos números dentro de un computador. Así, a la hora de construir la ALU, hay que tener en cuenta la posibilidad de que el resultado de una operación no se puede representar con los bits del hardware destinados a almacenar el resultado. Es decir, que pueda producirse un desbordamiento (*overflow*).

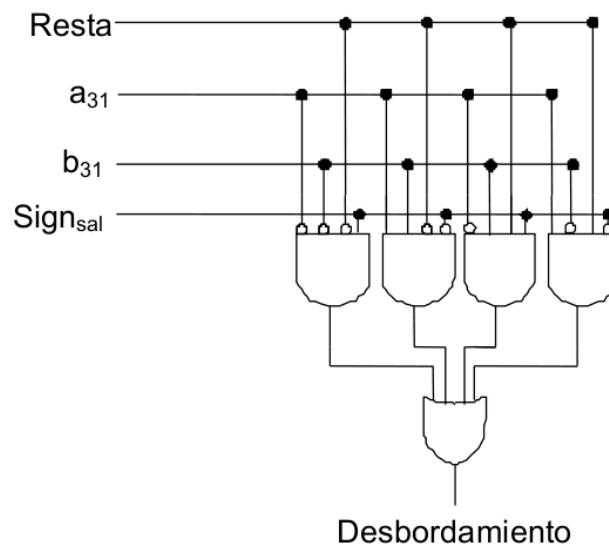


Figura B1.10: Unidad de detección de desbordamiento.

Por ejemplo, con 32 bits para representar números binarios, se sabe que se pueden representar  $2^{32}$  números diferentes. Suponiendo que operamos con aritmética entera positiva, estos números irían del 0 al  $2^{31} - 1$ . Si sumamos dos números y su resultado es mayor que  $2^{31} - 1$ , entonces se habrá producido un desbordamiento, puesto que el resultado no cabe en 32 bits. Más en general, en una operación aritmética sobre valores enteros se produce desbordamiento cuando el resultado no se puede representar con los bits destinados, en nuestro ejemplo 32 bits. En el caso de la suma, esto sucede cuando se suman dos números positivos y el resultado es negativo (en complemento a dos significa que el bit más significativo vale 1). En el caso de la resta sucede cuando se resta un número negativo de uno positivo y el resultado es negativo o cuando se resta uno positivo de uno negativo y el resultado es positivo.

La siguiente tabla muestra las condiciones que han de cumplirse para que se produzca desbordamiento:

Operación	Operando A	Operando B	Resultado
A+B	$\geq 0$	$\geq 0$	$< 0$
A+B	$< 0$	$< 0$	$\geq 0$
A-B	$\geq 0$	$< 0$	$< 0$
A-B	$< 0$	$\geq 0$	$\geq 0$

Tomando en cuenta la tabla anterior, es fácil extender la ALU31 (la correspondiente al bit de mayor peso) para que detecte si hubo desbordamiento, simplemente mirando el tipo de operación que se desea hacer (suma o resta) y los signos de los operandos de entrada y el de la salida. Si llamamos a éste último  $sign_{sal}$ , y pasamos las cuatro posibles condiciones de desbordamiento a un mapa de Karnaugh, obtenemos:

		Resta $a_{31}$				
		00	01	11	10	
$b_{31}$	$sign_{sal}$	00	0	0	1	0
		01	1	0	0	0
		11	0	0	0	1
		10	0	1	0	0

La función obtenida será:  $Desbordamiento = Resta' \cdot a'_{31} \cdot b'_{31} \cdot sign_{sal} + Resta' \cdot a_{31} \cdot b_{31} \cdot sign'_{sal} + Resta \cdot a'_{31} \cdot b_{31} \cdot sign_{sal} + Resta \cdot a_{31} \cdot b'_{31} \cdot sign'_{sal}$ .

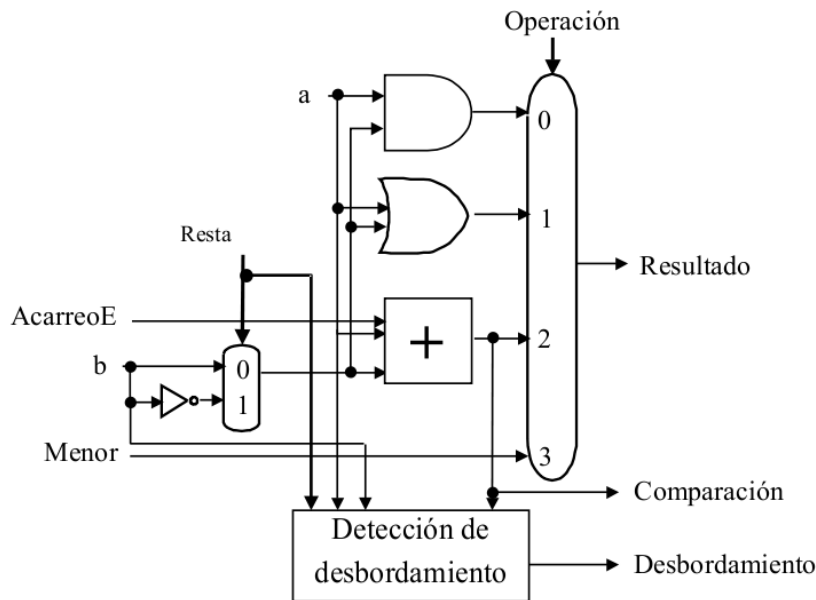


Figura B1.11: ALU31 con detección de desbordamiento.

La implementación con puertas correspondiente aparece en la figura B1.10. La ALU31, con circuito detector de desbordamiento, queda como se indica en la figura B1.11. Y la ALU completa de 32 bits que realiza sumas, restas, operaciones lógicas AND y OR (bit a bit), y comparaciones de magnitud e igualdad quedaría, finalmente, como aparece en la figura B1.12.

La relación entre las entradas de control y la operación que se llevaría a cabo en la ALU está especificado en la siguiente tabla:

Entradas			Operación
Resta	$Op_1$	$Op_0$	
0	0	0	AND
0	0	1	OR
0	1	0	SUMA
1	1	0	RESTA
1	1	1	SLT

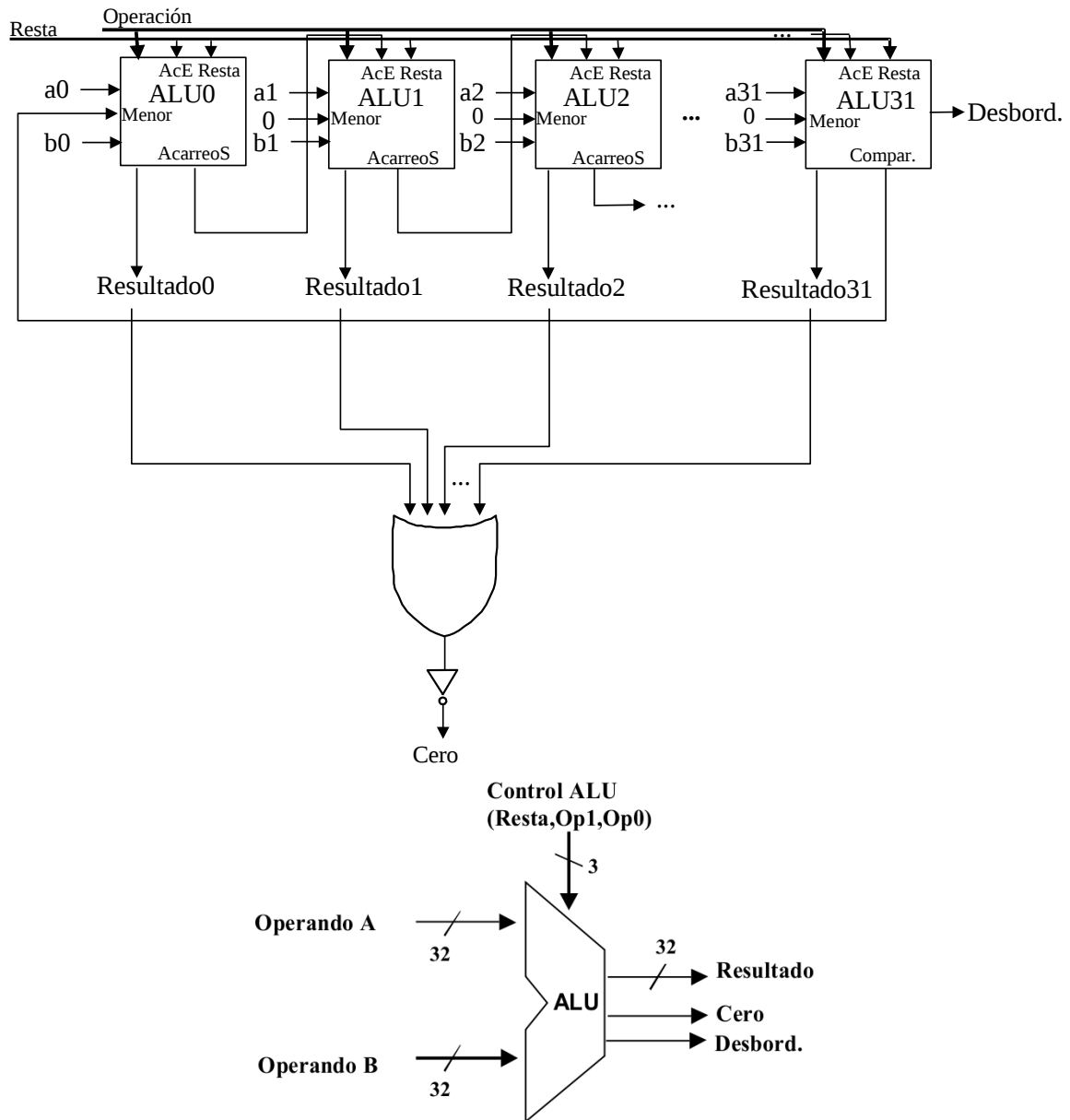


Figura B1.12: Esquema final de la ALU de 32 bits y bloque lógico correspondiente.

### B1.2.6. Ejercicios

Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto el mismo, los ficheros con los diferentes módulos/circuitos y, para cada uno de ellos, un cronograma en donde se muestre el funcionamiento del mismo bajo diferentes situaciones con el fin de que se pueda apreciar el correcto funcionamiento del circuito.

1. Utilizando el programa *TkGate* crear un módulo que implemente el sumador completo de un bit de la figura B1.5. Llamar a dicho módulo *SumadorCompleto*
2. Crear un segundo módulo con la unidad de detección de desbordamiento de la figura B1.10. Llamar a dicho módulo *Desbordamiento*
3. Utilizando ambos módulos, implementar la ALU de un bit de la figura B1.9 (pero eliminado la salida de comparación y añadiendo, en cambio, la salida *AcarreoS*, como en la figura B1.8) y la ALU de un bit de la figura B1.11. Encapsular ambas ALUs creando los módulos *ALU1bit* y *ALUFinal*, respectivamente.
4. Finalmente, implementar una ALU de 8 bits siguiendo el esquema mostrado en la figura B1.12. Mostrar el correcto funcionamiento del circuito mediante la ejecución de diferentes operaciones con varios valores de entrada (deberá incluirse en el portafolios al menos 6 de estas comprobaciones, incluyendo al menos un caso de desbordamiento y adjuntando los cronogramas correspondientes).

## Ejercicios

### E1.1. Sistemas Digitales: Circuitos Combinacionales

- Para cada una de las siguientes funciones, dar su expresión mínima primero en forma de suma de productos, y después en forma de producto de sumas (hacer uso de sus mapas de Karnaugh, simplificando por unos y por ceros, respectivamente):
  - $F(A, B, C) = A \cdot B' \cdot C + A' \cdot B \cdot C' + A' \cdot B \cdot C + A \cdot B' \cdot C'$
  - $F(A, B, C, D) = \sum(0, 2, 5, 7, 8, 10, 13, 15)$
  - $F(A, B, C, D) = \prod(2, 3, 6, 8, 9, 12)$
  - $F(A, B, C, D) = \sum(1, 4, 5, 6, 7, 9, 10, 13)$
  - $F(A, B, C, D) = A' \cdot B \cdot C' \cdot D + A \cdot B' \cdot C + A \cdot B' + A \cdot B' \cdot C' \cdot D$
  - $F(A, B, C, D, E) = \sum(1, 5, 7, 9, 13, 15, 16, 17, 20, 21, 22, 25, 29, 31)$
- Simplificar las funciones a), d) y f) del ejercicio 1, pero esta vez suponiendo que los minterminos 0,1,2,3,4,27,28,29,30,31 son condiciones de no importa. Comparar la complejidad de las funciones obtenidas con las que se obtuvieron en el ejercicio 1.
- Dibujar el circuito correspondiente a las funciones obtenidas en los apartados a), b) y f) del ejercicio 1 mediante puertas AND, OR y NOT.
- Para el ejercicio anterior, obtener circuitos equivalentes usando:
  - Sólo puertas NAND.
  - Sólo puertas NOR.
- Construir las funciones b) y d) del ejercicio 1, utilizando
  - Una ROM con el mínimo tamaño posible.
  - Un PLA con el mínimo tamaño posible.
- Implementar las funciones a) y b) del ejercicio 1 utilizando:
  - Un decodificador y una puerta OR del tamaño adecuado.
  - Multiplexores de los tamaños adecuados y ninguna puerta adicional (no se dispone de las entradas negadas).
  - Multiplexores con la mitad de entradas de datos.
- Muchos circuitos incluyen con frecuencia una entrada adicional que sirve para activar o desactivar la(s) salida(s) del módulo. Por ejemplo, la función desempeñada por un decodificador se inhibe haciendo que todas sus salidas pasen al estado cero. Así, la salida Y0 de un decodificador de 2 a 4 estaría dada por  $Y0 = X0' \cdot X1' \cdot E$ , donde E es la entrada de activación. Construir un decodificador de 3 a 8 que incluya una entrada de activación E. Mostrar su tabla de verdad.
- Determinar la ecuación para una función lógica con cuatro entradas A, B, C y D que toma valor uno si, y sólo si, un número impar de entradas es 1 (generador de paridad). ¿Se te ocurre alguna puerta lógica que pueda simplificar el circuito obtenido?

9. Un codificador se dice de prioridad si admite que más de una entrada sea distinta de cero, y en ese caso da prioridad a alguna de esas entradas a la hora de codificar la salida. Diseñar un codificador de prioridad con cuatro entradas  $E_0, E_1, E_2$  y  $E_3$  y dos salidas  $S_0$  y  $S_1$ , que dé prioridad a las líneas de entrada de menor peso, y en el cual la entrada  $(E_0, E_1, E_2, E_3) = (0, 0, 0, 0)$  no esté permitida (y por tanto pueda usarse como condición de no importa).
10. Diseñar un circuito combinacional, utilizando únicamente puertas NOR, con 4 variables de entrada, tal que tenga como salida  $F(A, B, C, D)$  un 1 si el número ABCD interpretado en binario natural es primo, y 0 si no lo es (p.e.,  $(ABCD) = (1001) \Rightarrow F = 0$ , porque 9 no es primo, pero  $(ABCD) = (1011) \Rightarrow F = 1$ , porque 11 sí lo es). Suponer que el 0 y uno no son números primos.
11. Diseñar un circuito exhibidor BCD de 7 segmentos. El circuito debe tomar como entrada un número de 4 bits, interpretarlo en BCD (es decir, sus entradas válidas son desde 0000 (0 decimal) hasta 1001 (9 decimal), el resto son entradas no permitidas), y generar 7 señales ( $a, b, c, d, e, f, g$ ) correspondientes a los 7 segmentos del exhibidor luminoso mostrado en la figura E1.1, de modo adecuado para que se visualice el número decimal correspondiente (por ejemplo, para la entrada 0001 (1), deben encenderse solamente los segmentos b y c, para el 7 los segmentos a, b y c, etc. Resolver el problema mediante dos diseños distintos:
  - a) Utilizando un PLA de tamaño  $4 \times 10 \times 7$ .
  - b) Simplificando mediante mapas de Karnaugh de modo independiente cada una de las funciones obtenidas. Nota.- Tener en cuenta que las condiciones de no importa para las combinaciones 1010 (10) a 1111 (15) deben usarse para simplificar los circuitos obtenidos.

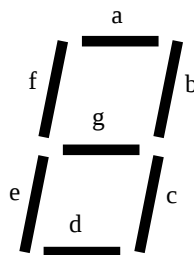


Figura E1.1: Exhibidor luminoso de 7 segmentos.

12. Calcular el retardo de un sumador de 32 bits construido a partir de la concatenación en serie de 32 sumadores completos de un bit que siguen el esquema visto en la figura B1.5. Suponer un retardo despreciable para las conexiones, de 5 ns para las puertas NOT, y de 10 ns para las AND y las OR, tengan el número de entradas que tengan.
13. La unidad de detección de desbordamiento de la ALU vista en prácticas puede simplificarse si, en lugar de trabajar con los bits  $b_{31}$  y Resta, trabajase directamente con la salida del multiplexor controlado por éste último bit (y cuyas entradas de datos eran  $b_{31}$  y  $b'_{31}$ ). Si llamamos a esta salida  $b_{mux}$ , construir un circuito detector de desbordamiento más simple que el visto, cuyas entradas sean  $b_{mux}$ ,  $a_{31}$  y  $sign_{sal}$ .
14. El C.I. CMOS 4532 (cuyo patillaje se muestra en la figura E1.2) es un codificador 8 a 3 con prioridad que asigna la prioridad más alta a la entrada  $D_7$  y la más baja a la entrada  $D_0$ . Además, posee las señales  $EI$  (*chip-enable input*),  $EO$  (*enable-out*) y  $GS$  (*group select*) cuya función es la siguiente:
  - Cuando  $EI$  es uno:
    - La representación binaria de la entrada de más prioridad aparece en las líneas de salida  $Q_2 - Q_0$ .

- La línea de selección de grupo  $GS$  se pone a uno si alguna de las entradas de prioridad está presente (a 1).
- La línea de permiso de salida ( $EO$ ) es uno cuando ninguna entrada de prioridad está presente.
- Cuando  $EI$  es cero: todas las salidas permanecen a cero.

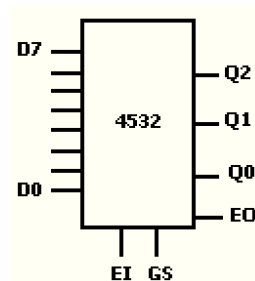


Figura E1.2: Patillaje del C.I. CMOS 4532.

Utilizando el mínimo número de C.I. CMOS 4532, diseñar un sistema de comunicación para las 16 habitaciones de un hotel, de modo que, cuando algún cliente desee que se persone el botones, pulse un botón, y en la conserjería se indique el número de habitación correspondiente codificado en binario. Si varios clientes llaman a la vez, sólo se indicará el de más categoría, que será el alojado en la habitación de mayor número. Cuando el botones esté libre (no le llama nadie) se indicará mediante una señal “LIBRE”.

15. Se dispone de 4 bits codificados en binario natural y se desea que la codificación utilizada sea en código Gray (o código binario reflejado). Realizar el circuito que satisfaga esta necesidad. Nota: el código Gray es el que se utiliza para numerar los Mapas de Karnaugh y cumple la propiedad de que entre un número y el siguiente/anterior sólo cambia un bit.
16. Implementar, mediante una PLA mínima, un convertidor de Gray (4 bits) a binario.
17. Diseñar un circuito restador elemental de 1 bit, con salida de resta (R) y “debe” (B). Utilizando este circuito como base, implementar un circuito restador de números de 4 bits.
18. Diseñar un circuito comparador de números de 2 bits utilizando puertas elementales, para su posterior integración en MSI. Debe aceptar como entrada los 4 bits de las palabras y dará como salidas el resultado de la comparación:  $A > B$ ,  $A = B$  y  $A < B$ . Añadir al circuito resultante unas entradas de comparación previa de los bits menos significativos, para facilitar la conexión en cascada. Mostrar como se utilizaría el circuito final para comparar números de 4 bits.
19. Diseñar, mediante un decodificador de 4 a 16 líneas, un circuito capaz de realizar la suma de 2 números de 2 bits, con salida de suma y acarreo.
20. Diseñar un circuito que tenga como entrada un valor  $N > 0$  (de tres bits) y dé como salida  $N^2 - 1$ .
  - a) Mediante puertas lógicas sin limitaciones.
  - b) Mediante puertas NAND.
  - c) Mediante puertas NOR.
  - d) Mediante una PLA mínima.
  - e) Mediante una ROM mínima.

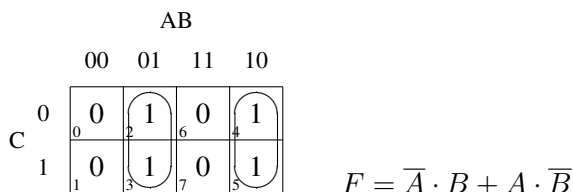
### E1.2. Solución a ejercicios seleccionados

1. Para cada una de las siguientes funciones, dar su expresión mínima primero en forma de suma de productos, y después en forma de producto de sumas (hacer uso de sus mapas de Karnaugh, simplificando por unos y por ceros, respectivamente):

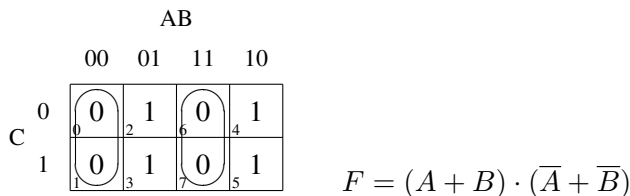
a)  $F(A, B, C) = A \cdot B' \cdot C + A' \cdot B \cdot C' + A' \cdot B \cdot C + A \cdot B' \cdot C'$

**Solución:**

Simplificando por unos:

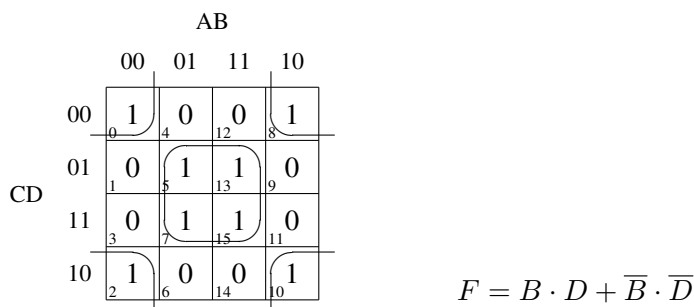


Y simplificando por ceros:

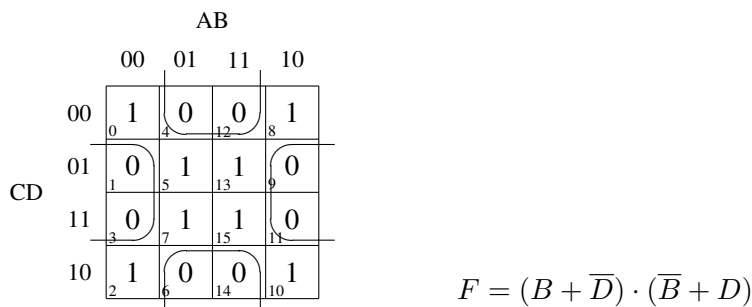


b)  $F(A, B, C, D) = \sum(0, 2, 5, 7, 8, 10, 13, 15)$

Simplificando por unos:



Y simplificando por ceros:



c)  $F(A, B, C, D) = \prod(2, 3, 6, 8, 9, 12)$

Simplificando por unos:

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

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

Y simplificando por ceros:

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

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

d)  $F(A, B, C, D) = \sum(1, 4, 5, 6, 7, 9, 10, 13)$

Simplificando por unos:

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

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

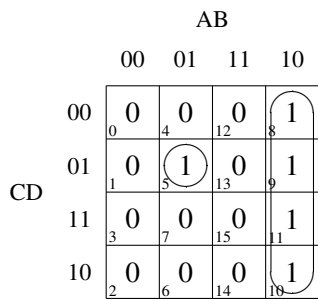
Y simplificando por ceros:

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

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

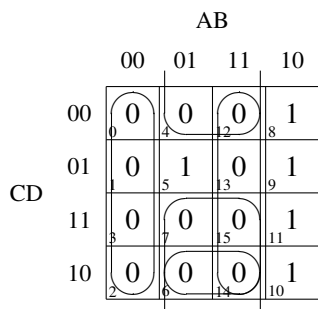
e)  $F(A, B, C, D) = A' \cdot B \cdot C' \cdot D + A \cdot B' \cdot C + A \cdot B' + A \cdot B' \cdot C' \cdot D$

Simplificando por unos:



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

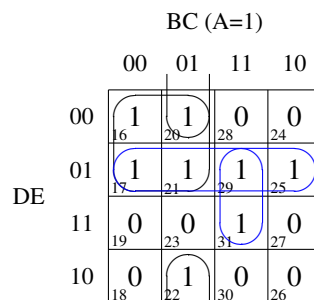
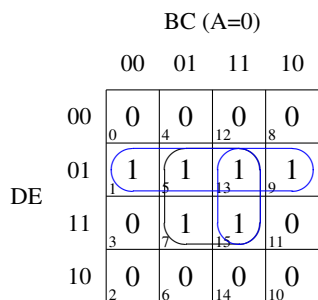
Y simplificando por ceros:



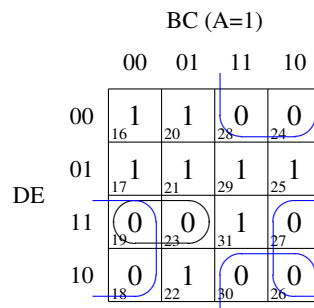
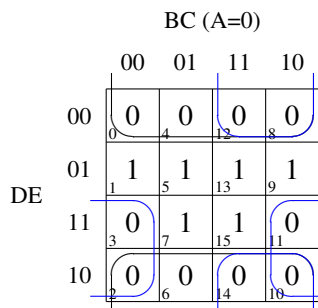
$$F = (A + B) \cdot (\bar{A} + \bar{B}) \cdot (\bar{B} + \bar{C}) \cdot (\bar{B} + D)$$

f)  $F(A, B, C, D, E) = \sum(1, 5, 7, 9, 13, 15, 16, 17, 20, 21, 22, 25, 29, 31)$

Simplificando por unos:



En donde se ha marcado en azul aquellas simplificaciones comunes a ambos mapas. La función obtenida es:  $F = \bar{D} \cdot E + \bar{A} \cdot C \cdot E + A \cdot \bar{B} \cdot \bar{D} + B \cdot C \cdot E + A \cdot \bar{B} \cdot C \cdot \bar{E}$ . Y simplificando por ceros:



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

4. Para las funciones obtenidas en los apartados a), b) y f), obtener circuitos equivalentes usando:

- a) Sólo puertas NAND.
- b) Sólo puertas NOR.

**Solución:**

Existen dos posibles formas de solucionar este problema. La primera de ellas es ir sustituyendo cada una de las puertas obtenidas (AND, OR y NOT) por sus equivalente con puertas NAND (NOR) y después intentar simplificar el circuito resultante eliminando puertas redundantes. La segunda opción consiste en realizar la simplificación adecuada para que negando dos veces la solución y aplicando Morgan una vez para desarrollar la negación interior obtener la expresión buscada. Esta opción tiene la ventaja adicional de obtener de forma directa la implementación con puertas NAND (NOR), no siendo necesario realizar el proceso de simplificación.

a) Sólo puertas NAND.

En este caso bastan con utilizar las funciones simplificadas en forma de suma de productos (simplificar por unos):

$$F = \overline{A} \cdot B + A \cdot \overline{B} = \overline{\overline{\overline{\overline{\overline{A} \cdot B + A \cdot \overline{B}}}}} = \overline{(\overline{A \cdot B}) \cdot (\overline{A \cdot \overline{B}})}$$

$$F = B \cdot D + \overline{B} \cdot \overline{D} = \overline{\overline{\overline{\overline{\overline{B \cdot D + \overline{B} \cdot \overline{D}}}}} = \overline{(B \cdot D) \cdot (\overline{B} \cdot \overline{D})}$$

$$F = \overline{\overline{\overline{\overline{\overline{\overline{\overline{D} \cdot E + \overline{A} \cdot C \cdot E + A \cdot \overline{B} \cdot \overline{D}} + B \cdot C \cdot E + A \cdot \overline{B} \cdot C \cdot \overline{E}}}}}}}} = \overline{\overline{\overline{\overline{\overline{\overline{\overline{D} \cdot E + \overline{A} \cdot C \cdot E + A \cdot \overline{B} \cdot \overline{D}} + B \cdot C \cdot E + A \cdot \overline{B} \cdot C \cdot \overline{E}}}}}}}} = \overline{(\overline{D \cdot E}) \cdot (\overline{\overline{A \cdot C \cdot E}}) \cdot (\overline{A \cdot \overline{B} \cdot \overline{D}}) \cdot (\overline{B \cdot C \cdot E}) \cdot (\overline{A \cdot \overline{B} \cdot C \cdot \overline{E}})}$$

b) Sólo puertas NOR.

Para la simplificación con puertas NOR, utilizaremos la simplificación por ceros que da como resultado un producto de sumas:

$$F = (A + \overline{B}) \cdot (\overline{A} + B) = \overline{\overline{\overline{\overline{\overline{(A + \overline{B}) \cdot (\overline{A} + B)}}}}} = \overline{\overline{\overline{\overline{\overline{(A + \overline{B}) + (\overline{A} + B)}}}}}}$$

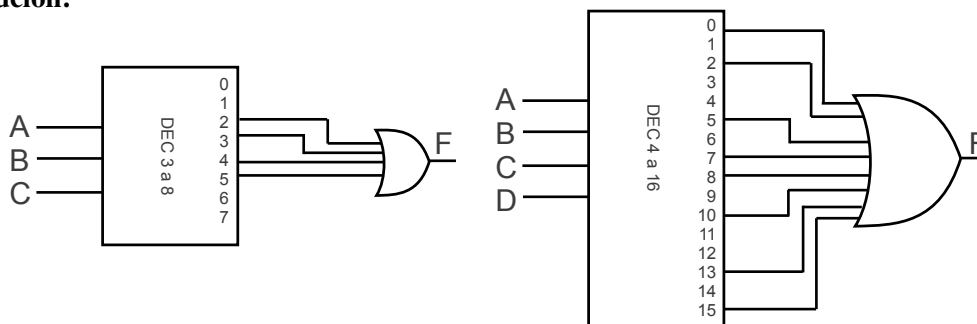
$$F = (B + \overline{D}) \cdot (\overline{B} + D) = \overline{\overline{\overline{\overline{\overline{(B + \overline{D}) \cdot (\overline{B} + D)}}}}} = \overline{\overline{\overline{\overline{\overline{(B + \overline{D}) + (\overline{B} + D)}}}}}}$$

$$F = \overline{\overline{\overline{\overline{\overline{(\overline{B} + E) \cdot (C + \overline{D}) \cdot (A + B + E) \cdot (\overline{A} + B + \overline{D} + \overline{E})}}}}}} = \overline{\overline{\overline{\overline{\overline{(\overline{B} + E) \cdot (C + \overline{D}) \cdot (A + B + E) \cdot (\overline{A} + B + \overline{D} + \overline{E})}}}}}} = \overline{(\overline{\overline{B + E}}) + (\overline{C + \overline{D}}) + (\overline{A + B + E}) + (\overline{\overline{A + B + \overline{D} + \overline{E}}})}$$

6. Implementar las funciones a) y b) del ejercicio 1 utilizando:

a) Un decodificador y una puerta OR del tamaño adecuado.

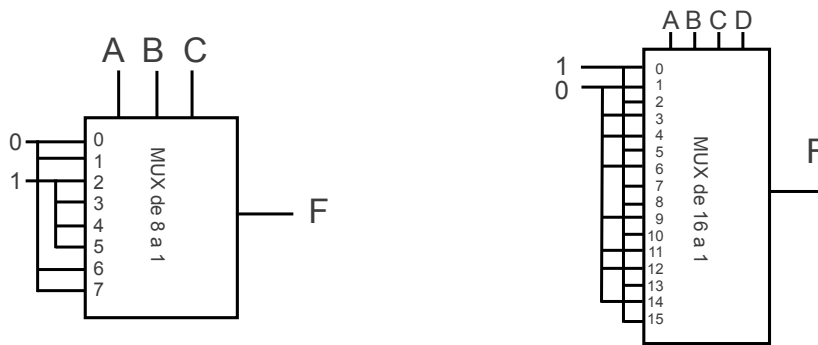
**Solución:**



b) Multiplexores de los tamaños adecuados y ninguna puerta adicional (no se dispone de las entradas negadas).

**Solución:**

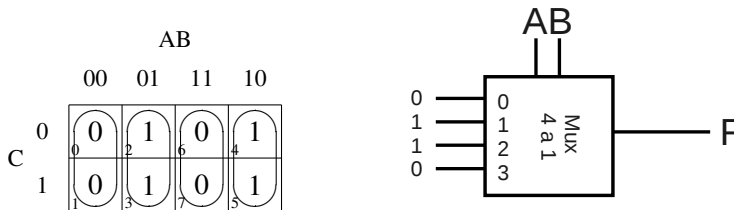
Supondremos que cuando hablan de multiplexores de los tamaños adecuados, se refieren a multiplexores de  $2^n$  entradas, donde  $n$  es el número de variables de la función. Entonces, las soluciones pedidas son:



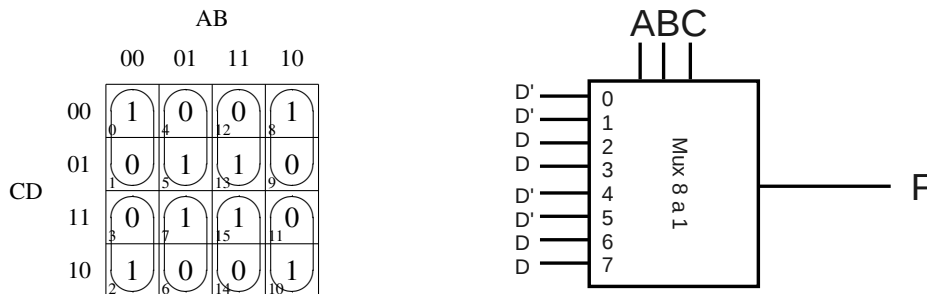
c) Multiplexores con la mitad de entradas de datos.

**Solución:**

Elegiremos en el primer caso como variable excluida la C, de tal manera que el circuito resultante será:



En el caso del segundo circuito, la variable que decidimos excluir es la D:



10. Diseñar un circuito combinacional, utilizando únicamente puertas NOR, con 4 variables de entrada, tal que tenga como salida F(A,B,C,D) un 1 si el número ABCD interpretado en binario natural es primo, y 0 si no lo es (p.e., (ABCD)=(1001) => F=0, porque 9 no es primo, pero (ABCD)=(1011) => F=1, porque 11 sí lo es). Suponer que el 0 y uno no son números primos.

**Solución:**

El mapa de Karnaugh de la función indicada sería:

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

en donde aparecen a 1 las celdas correspondientes a los números primos: 2, 3, 5, 7, 11 y 13. Simplificando por ceros obtenemos:

$$F = (B + C) \cdot (\overline{B} + D) \cdot (\overline{A} + D) \cdot (\overline{A} + \overline{B} + \overline{C}) = \overline{\overline{(B + C)} + \overline{(\overline{B} + D)} + \overline{(\overline{A} + D)} + \overline{(\overline{A} + \overline{B} + \overline{C})}}$$

Función que puede implementarse directamente con puertas NOR.

15. Se dispone de 4 bits codificados en binario natural y se desea que la codificación utilizada sea en código Gray (o código binario reflejado). Realizar el circuito que satisfaga esta necesidad. Nota: el código Gray es el que se utiliza para numerar los Mapas de Karnaugh y cumple la propiedad de que entre un número y el siguiente/anterior sólo cambia un bit.

**Solución:**

Según la indicado, el código de Gray de 4 bits será:

ENTRADAS				SALIDAS			
A	B	C	D	F3	F2	F1	F0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Calculando los mapas de Karnaugh de las cuatro funciones de salida obtenemos:

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

$$F0 = C \oplus D$$

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

$$F1 = B \oplus C$$

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

$$F2 = A \oplus B$$

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

$$F3 = A$$