ESTRUCTURA Y TECNOLOGÍA DE COMPUTADORES

Sebastián Dormido
Mª Antonia Canto
José Mira
Ana E. Delgado

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

SANZ Y TORRES

Tema 4

Unidad aritmético-lógica

La *unidad aritmético-lógica* (ALU[1]) es la parte del computador donde se efectúan las operaciones aritméticas y lógicas sobre los datos. Las otras unidades del computador (unidad de control, memoria y unidad de E/S) son las encargadas de suministrar datos a la entrada de la ALU y recibirlos nuevamente una vez procesados. La Figura 4.1 muestra en términos generales como se interconecta la ALU con el resto de la CPU.

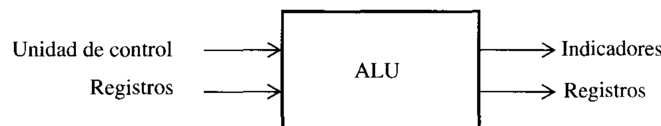


Figura 4.1: Entradas y salidas de la ALU

Los datos llegan a la ALU a través de registros y los resultados que se generan también se almacenan en registros. Estos registros son memorias temporales dentro de la CPU que se conectan mediante el bus de datos con la ALU. Cuando la ALU finaliza una operación, activa determinados indicadores que pueden ser utilizados por la unidad de control. La unidad de control envía señales que controlan las operaciones y el movimiento de datos de entrada y salida de la ALU. En este tema se estudian los algoritmos y los circuitos asociados que realizan las cuatro operaciones aritméticas básicas, tanto en coma fija como en coma flotante. Un número en coma flotante está constituido por un par de números en coma fija, la mantisa m y el exponente e y se utiliza para representar números de la forma $m \times B^e$, donde B es la base que está implícita. La representación en coma flotante aumenta el rango de los números que se pueden expresar para una longitud de palabra dada, aunque requieren circuitos aritméticos mucho más complejos que cuando se emplea coma fija. Con el fin de proporcionar una representación única para cada número en coma flotante se realiza un proceso de normalización.

También se analizan las operaciones de desplazamiento y de comparación.

1. A lo largo de todo el tema se utilizará el término ALU que es el acrónimo de (*Arithmetic-Logical Unit*) para designar a la unidad aritmético-lógica.

Sección 4-1

Sumadores binarios

Un sumador binario se puede considerar como un conversor de código que recibe a la entrada dos números binarios x e y de n bits cada uno (los operandos)

$$x = x_{n-1}\, x_{n-2} \ldots\ldots x_1\, x_0$$

$$y = y_{n-1}\, y_{n-2} \ldots\ldots y_1\, y_0$$

y produce una salida s de $n + 1$ bits que es la suma de los operandos (ver Figura 4.2).

$$s = s_n\, s_{n-1} \ldots\ldots s_1\, s_0$$

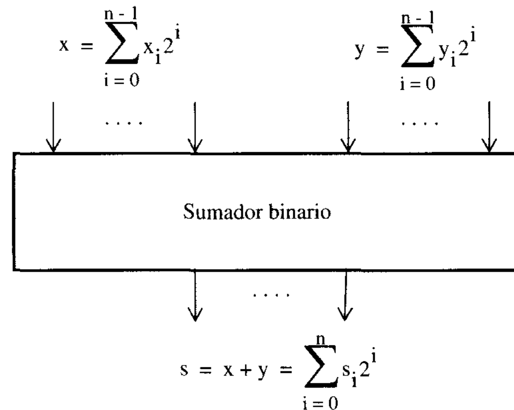


Figura 4.2: Sumador binario de n bits

En la realización de un sumador binario aparece un compromiso entre velocidad y coste. En principio, la forma más rápida de realizar cualquier función lógica, cuando el valor de sus salidas está únicamente determinado por el valor presente de sus entradas, es mediante un circuito combinacional de dos niveles (AND-OR). Se observa en la Tabla 4.1 que el bit s_j de la suma (j = 0, 1, ..., n) es función de las $2(j+1)$ variables x_i, y_i (i = 0, 1, ..., j).

Por lo tanto, en una realización canónica que incluyese todos los términos productos, el bit s_j de la suma requeriría $2^{2(j+1)}$ puertas AND de $2(j+1)$ entradas y una puerta OR de $2^{2(j+1)}$ entradas. Este número de puertas resulta excesivo incluso para valores moderados de n, por lo que es inviable esta solución y es preciso desarrollar otros métodos que, aunque más lentos, sean menos costosos.

s_0	=	$s_0\,(x_0, y_0)$
s_1	=	$s_1\,(x_1, y_1, x_0, y_0)$
....		
s_{n-1}	=	$s_{n-1}\,(x_{n-1}, y_{n-1}, \ldots\ldots, x_1, y_1, x_0, y_0)$
s_n	=	$s_n\,(x_{n-1}, y_{n-1}, \ldots\ldots, x_1, y_1, x_0, y_0)$

Tabla 4.1: Dependencia de los bits de suma de los bits de los operandos

La estrategia básica que se emplea para sumar dos números de n bits es dividir el problema en un conjunto de tareas más sencillas, de forma tal que una vez resueltas se tenga, por simple composición, la solución del problema original. La partición más usual es transformar el problema de *suma de dos números de n bits* en *n problemas idénticos de suma de dos números de 1 bit*.

Cada una de estas n sumas más sencillas necesitan información de los sumadores de las etapas previas (*arrastres*). Estas señales de arrastre (denominadas también de *acarreo*) que se producen en los sumadores elementales de orden inferior (de los bits menos significativos al que se considera), son el origen de las dificultades que surgen en la operación de suma.

A continuación se analizan algunas realizaciones del circuito lógico básico para la suma de dos números de 1 bit (semisumador), su generalización cuando se considera el arrastre de la etapa previa (sumador binario completo) y como se efectúa la suma de dos números de n bits interconectando adecuadamente sumadores binarios completos. En la Figura 4.3 se muestra el esquema de la *representación de puntos* de la suma de dos números binarios de n bits. Consta de los siguientes elementos:

a) Una matriz de puntos de 2 filas $\times$ n columnas, donde cada uno de los puntos representa un bit individual de los dos números (x e y) a sumar.

b) Una linea horizontal que separa las representaciones de la entrada y de la salida.

c) Un vector fila de puntos de $n + 1$ columnas que representan los bits de la suma (s).

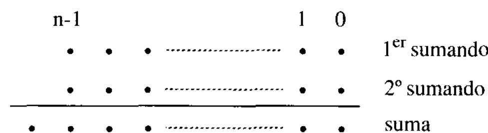


Figura 4.3: Representación de puntos de un sumador binario de n bits

4.1.1 Semisumador binario (SSB)

Es el circuito aritmético más sencillo. Consta de dos entradas binarias (x e y) y dos salidas; una de ellas (s) es el resultado de la "suma módulo 2" de las dos entradas y la otra (c) es el denominado "arrastre" (o acarreo) que indica si el resultado de la suma es igual a 2 (ver Tabla 4.2).

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabla 4.2: Tabla de verdad del SSB

En la Figura 4.4 se da el diagrama de un semisumador binario (SSB) y su representación de puntos.

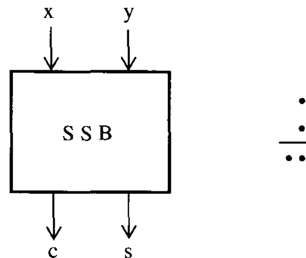


Figura 4.4: Diagrama de bloques y representación de puntos de un SSB

Las funciones lógicas de s y c, deducidas directamente de la tabla de verdad, se pueden expresar mediante las siguientes ecuaciones:

$$s = \overline{x}\,y + x\,\overline{y} = x \oplus y$$

$$c = x\,y$$

donde el operador $\oplus$ representa la función OR-exclusiva.

El bit de suma s es 1 cuando x o y pero no ambos es 1, y el bit de arrastre c es 1 cuando ambos x e y son 1. El SSB se puede realizar de múltiples formas dependiendo de las puertas lógicas que se tengan, y de si se dispone o no del complemento de los bits de entrada. En la Figura 4.5 se muestran diferentes realizaciones del SSB.

4.1.2 Sumador binario completo (SBC)

Se diferencia del SSB porque tiene una tercera entrada c_{i-1} llamada "arrastre de la etapa anterior", que le permite encadenarse con otros SBC para el diseño de circuitos de suma de números de n bits ($n > 1$).

El SBC acepta como entradas un bit de cada uno de los operandos (x_i e y_i) y un bit de arrastre (c_{i-1}) de la etapa previa, y genera como salida un bit de suma (s_i) y un bit de arrastre (c_i) para la etapa siguiente de acuerdo con la Tabla 4.3. En la Figura 4.6 se da el diagrama de bloques y la representación de puntos de un SBC.

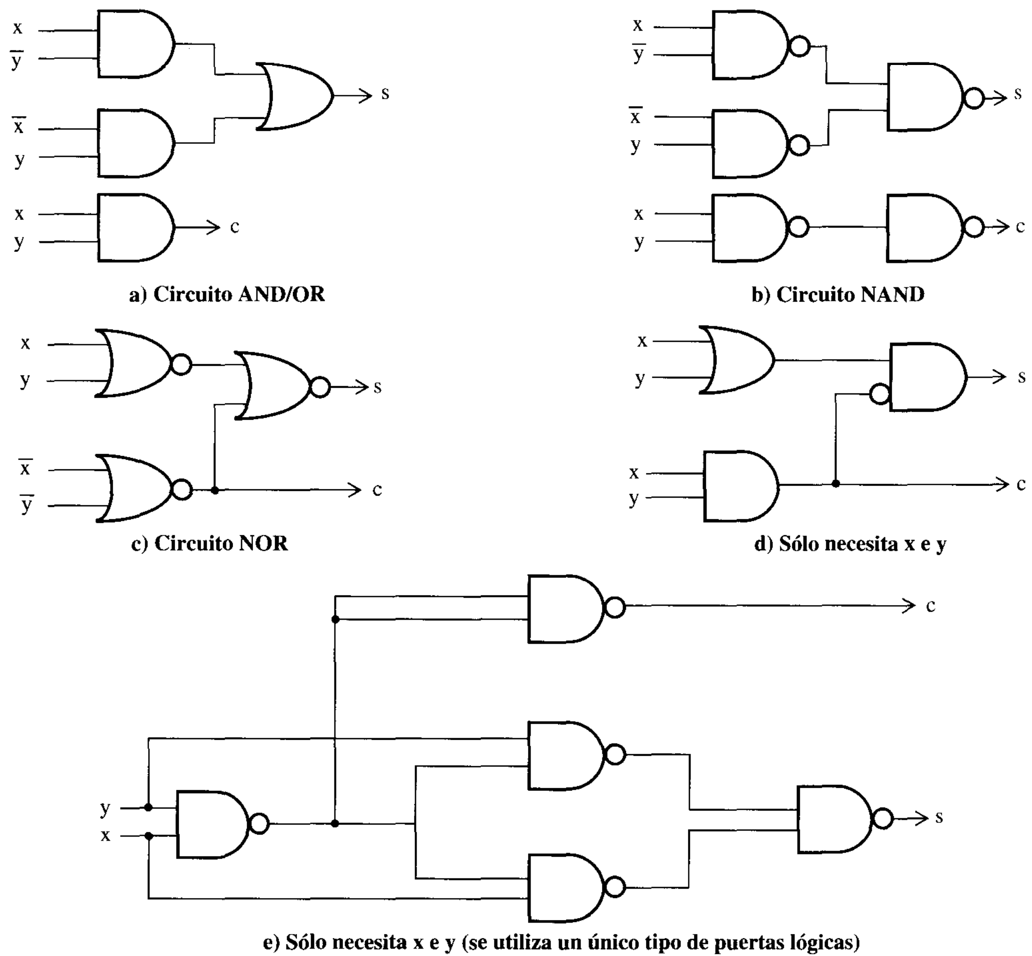


Figura 4.5: Diferentes realizaciones lógicas de un SSB

x_i	y_i	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabla 4.3: Tabla de verdad del SBC

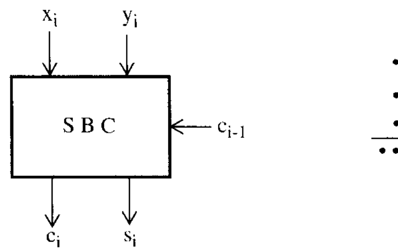


Figura 4.6: Diagrama de bloques y representación de puntos de un SBC

Las funciones lógicas de s_i y c_i, deducidas del mapa de Karnaugh de la Figura 4.7a , se pueden expresar mediante las siguientes ecuaciones:

$$s_i = \bar{x}_i\,\bar{y}_i\,c_{i-1} + \bar{x}_i\,y_i\,\bar{c}_{i-1} + x_i\,\bar{y}_i\,\bar{c}_{i-1} + x_i\,y_i\,c_{i-1}$$

$$c_i = x_i\,y_i + x_i\,c_{i-1} + y_i\,c_{i-1}$$

con lo que s_i y c_i se sintetizan mediante los circuitos lógicos de dos niveles que se muestran en la Figura 4.7b :

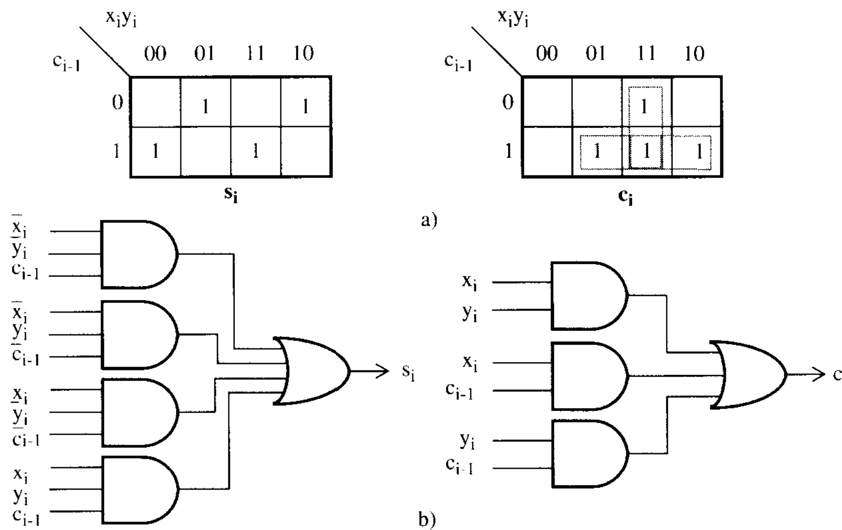


Figura 4.7: Diseño lógico de un SBC a) Mapa de Karnaugh b) Circuito lógico

El SBC también se puede realizar utilizando 2 SSB's conectados en cascada, tal como se muestra en la Figura 4.8, pero el circuito obtenido resulta más lento, debido a que las señales de entrada tienen que atravesar más niveles de puertas para generar la salida. A partir de las expresiones de s_i y c_i deducidas anteriormente se puede obtener:

$$s_i = (\bar{x}_i\,\bar{y}_i + x_i\,y_i)\,c_{i-1} + (x_i\,\bar{y}_i + \bar{x}_i\,y_i)\,\bar{c}_{i-1} = x_i \oplus y_i \oplus c_{i-1} \qquad (4.1)$$

$$c_i = x_i\,y_i + (x_i\,\bar{y}_i + \bar{x}_i\,y_i)\,c_{i-1} = x_i\,y_i + (x_i \oplus y_i)\,c_{i-1} \qquad (4.2)$$

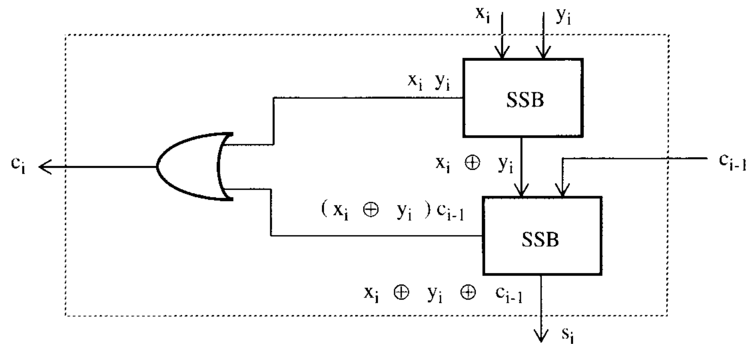


Figura 4.8: Realización de un SBC con 2 SSB´s

En la Tabla 4.4 se dan los retardos (medidos como el número de niveles de puertas lógicas que hay que pasar desde las entradas hasta las salidas) para los circuitos de las Figuras 4.7 y 4.8.

	s_i	c_i
Realización en 2 niveles	3	2
Realización con 2 SSB's	6	5

Tabla 4.4: Niveles de retardo en las dos realizaciones del SBC

4.1.3 Sumador binario serie

Con un SBC de 1 bit se puede diseñar un circuito sencillo que realiza la suma de dos números de *n* bits. Los bits del mismo peso, x_i e y_i, de los dos números a sumar se deben introducir sucesivamente en la entrada del SBC, comenzando por los bits menos significativos x_0 e y_0 (ver Figura 4.9).

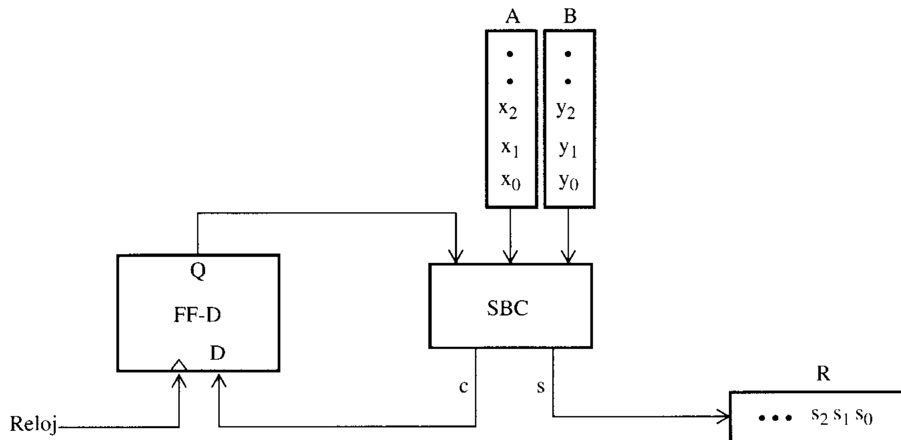


Figura 4.9: Sumador binario serie

Es necesario incluir un retardo unitario (elemento de memoria) para retener el arrastre c producido por el SBC de un intervalo al siguiente. Al comenzar la suma, la salida Q del elemento de memoria se inicializa a 0. El registro R donde se almacena el resultado puede ser el mismo registro A, en cuyo caso se denomina registro *acumulador*. Si se representa por Δ el retardo producido por el elemento de memoria (fijado fundamentalmente por la frecuencia del reloj), y por δ el tiempo que tarda el SBC en producir la suma de un bit, el tiempo que se necesita para realizar la suma de dos números de n bits será $n(\Delta + \delta)$. En un sumador binario serie la complejidad del circuito es independiente del número n de bits que hay que sumar, pero no sucede lo mismo con el tiempo de cálculo que crece linealmente con n.

4.1.4 Sumador binario paralelo con propagación del arrastre

La mayoría de las aplicaciones requieren sumas más rápidas que las que se pueden conseguir con los sumadores binarios serie. La manera clásica de resolver este problema es conectando una cadena de SBC's, de forma que se introduzcan en paralelo todos los bits de cada uno de los dos operandos.

Un sumador binario paralelo es un circuito que suma de forma simultánea todos los bits. Para sumar dos números de n bits, el circuito se puede realizar mediante el encadenamiento de n SBC's tal como se indica en la Figura 4.10.

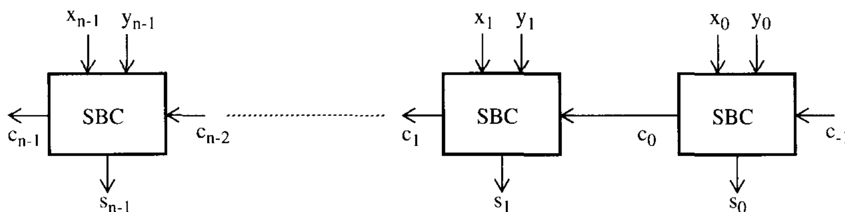


Figura 4.10: Sumador binario paralelo con propagación del arrastre (SPA)

Analizando la estructura de este sumador se observa que la salida s_{n-1} del SBC que opera sobre los bits más significativos no será válida hasta que se conozca el arrastre c_{n-2} producido por el SBC de la etapa anterior. Este mismo razonamiento se puede aplicar a cualquier etapa del sumador binario paralelo, por lo que en el caso más desfavorable, es decir, cuando se propaga un arrastre desde la etapa menos significativa hasta la más significativa, el resultado de la suma no será efectivo hasta que haya pasado un tiempo $\tau = n\delta$, siendo δ el tiempo que tarda un SBC en generar el arrastre para la etapa siguiente. En la Figura 4.11 se muestra un ejemplo del caso más desfavorable para $n = 8$.

```
  01001101
  10110011
----------
1 00000000
```

Figura 4.11: Ejemplo de caso más desfavorable de propagación del arrastre

Por este motivo, a este tipo de sumador binario paralelo se le conoce como *sumador con propagación del arrastre* (SPA). El retardo resultante en el sumador binario paralelo depende linealmente de n, por lo que será elevado para grandes valores de n y especialmente significativo cuando se conectan algunos módulos sumadores para formar un sumador más grande.

Se pueden conseguir sumadores binarios paralelos más rápidos que el sumador con propagación de arrastre utilizando circuitos que emplean más puertas lógicas con mayor número de entradas. La clave del compromiso rapidez-coste en los sumadores de alta velocidad estriba en el mecanismo utilizado para acelerar la generación de los arrastres.

4.1.5 Sumador-restador binario paralelo con propagación del arrastre

El sumador de la Figura 4-10 funciona correctamente en el caso de números sin signo y de números positivos, porque el 0 del bit de signo de un número positivo tiene el mismo efecto que un 0 a la izquierda en un número sin signo. La forma adecuada de sumar números negativos (que tienen un 1 en el bit de signo) depende del tipo de representación que se utilice (magnitud signo, complemento a 1ó complemento a 2).

Como $x - y = x + (-y)$, si se dispone de un circuito que calcula la suma de números negativos se puede efectuar la operación de restar. En el caso de la representación de números negativos en complemento a 2 la resta resulta especialmente sencilla, porque el valor negativo es muy simple de realizar. En efecto, si $y = y_{n-1}\, y_{n-2} \ldots\, y_1\, y_0$ es un número entero representado en complemento a 2, entonces $-y$ se define por:

$$-y = \overline{y}_{n-1}\,\overline{y}_{n-2}\ldots\,\overline{y}_1\,\overline{y}_0 + 1$$

así pues, para obtener $-y$ a partir de y, se efectuan las dos operaciones siguientes:

a) Sustituir todos los bits de y por su complemento (es decir, cambiar $0 \rightarrow 1$ y $1 \rightarrow 0$)

b) Sumar 1 al número resultante

Teniendo esto en cuenta, se puede diseñar fácilmente un único circuito sumador-restador (ver Figura 4.12).

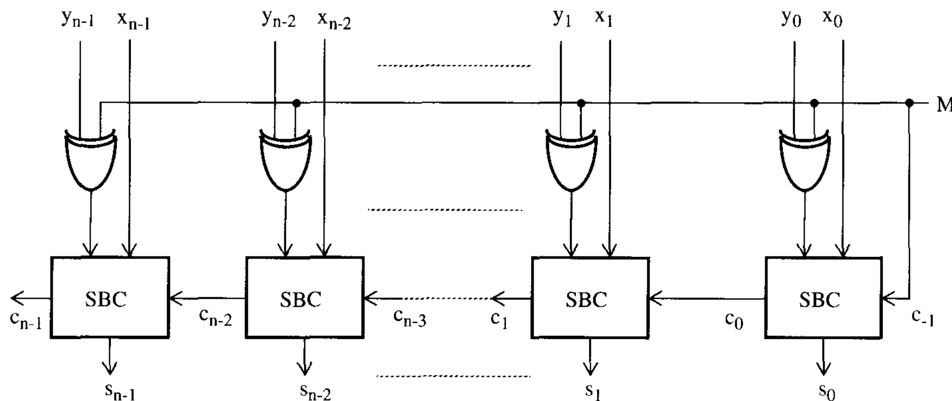


Figura 4.12: Sumador-restador binario paralelo con propagación de arrastre

Con cada módulo SBC se incluye una puerta OR-Exclusiva; la de la etapa i tiene como entradas M e y_i. La entrada M controla la operación. Si $M = 0$ el circuito es un sumador y cuando $M = 1$ el circuito se convierte en un restador. Cuando $M = 0$, como $y_i \oplus 0 = y_i$, los SBC's reciben el valor de y, la entrada de arrastre c_{-1} es 0, y el circuito realiza la operación $x + y$.

Cuando $M = 1$, como $y_i \oplus 1 = \overline{y}_i$ y $c_{-1} = 1$, se complementan todos los bits del número y se suma 1 debido

al arrastre de entrada del circuito. El circuito realiza la operación $x + C2(y)$, donde $C2(y)$ es el complemento a 2 de y. Para números sin signo, esto representa:

$$(x - y) \quad \text{si} \quad x \geq y$$

$$C2(y - x) \quad \text{si} \quad x < y$$

Para números con signo el resultado es $(x - y)$ si no se produce rebose (overflow). La simplicidad del circuito lógico y la rapidez para efectuar tanto la suma como la resta de números con signo en complemento a 2, es la razón de por qué esta representación es la más utilizada en las unidades aritméticas de los computadores actuales.

Podría pensarse que la representación en complemento a 1 es tan buena como la de complemento a 2 para utilizarla en un circuito sumador-restador combinado. No obstante, aunque en este caso la complementación es trivial, el resultado obtenido después de la suma no es siempre correcto. No se puede ignorar el arrastre de salida c_{n-1}. Si $c_{n-1} = 1$, hay que sumar 1 al resultado para corregirlo. Si $c_{n-1} = 0$, el resultado que se obtiene es correcto. El requisito de este ciclo de corrección, que es condicional según sea el arrastre de salida de la suma, hace más complicado el circuito sumador-restador en complemento a 1 que en complemento a 2.

Detección del rebose en el circuito sumador-restador con propagación del arrastre

Cuando se suman números sin signo el arrastre de la última etapa sirve como un indicador de rebose. Sin embargo, en el caso de números con signo la cosa es algo más complicada. Obviamente, la suma de números con signos distintos no puede nunca originar un rebose, porque el valor aboluto de la suma será siempre más pequeño que el mayor valor absoluto de los dos sumandos. En el caso de que los sumandos tengan el mismo signo es cuando se puede producir un rebose; pueden ocurrir dos casos:

a) *x e y son positivos.* Como los bits de signo x_{n-1} e y_{n-1} son iguales a cero, no se puede generar ningún arrastre en la posición del bit de signo ($c_{n-1} = 0$). Sin embargo, el bit de signo de la suma s_{n-1} será 1 si se genera o propaga un arrastre desde la etapa anterior ($c_{n-2} = 1$) y el resultado es incorrecto (la suma de dos números positivos da un número negativo).

b) *x e y son negativos.* Como los bits de signo x_{n-1} e y_{n-1} son iguales a uno, se genera siempre un arrastre en la posición del bit de signo ($c_{n-1} = 1$). El bit de signo de la suma s_{n-1} será 0 si no se genera o no se propaga un arrastre desde la etapa anterior ($c_{n-2} = 0$) y el resultado es incorrecto (la suma de dos números negativos da un número positivo).

La Tabla 4.5 resume las condiciones de rebose expresadas en a) y b). La expresión lógica de la señal de rebose R deducida directamente de la Tabla 4-5 es:

$$R = x_{n-1}\, y_{n-1}\, \overline{s}_{n-1} + \overline{x}_{n-1}\, \overline{y}_{n-1}\, s_{n-1} \qquad (4.3)$$

Es posible expresar la señal de rebose R como una función de los arrastres c_{n-2} y c_{n-1} mediante la siguiente función lógica:

$$R = c_{n-2} \oplus c_{n-1} \qquad (4.4)$$

En la Figura 4.13 se muestra el circuito sumador-restador de la Figura 4-12 con el circuito de detección de rebose incorporado. Es importante detectar la aparición de un rebose en un computador, para poder decidir qué acción realizar cuando éste se produzca. Por esta razón es usual que la unidad aritmética disponga de un registro de 1 bit (registro de rebose) que se pone a 1 cuando se genera un rebose.

x_{n-1}	y_{n-1}	s_{n-1}	**Rebose (R)**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Tabla 4.5: Condiciones de rebose

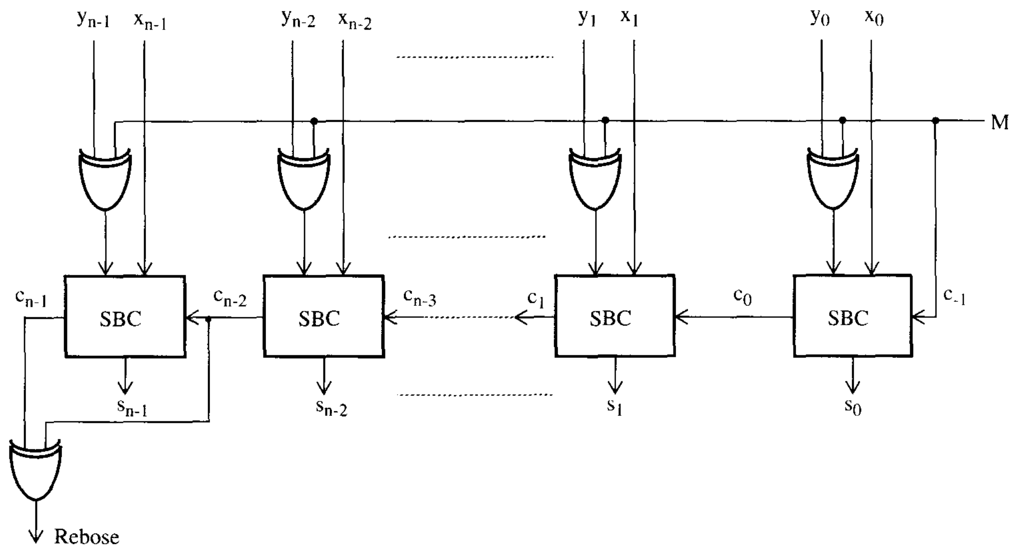


Figura 4.13: Sumador-restador binario paralelo con propagación del arrastre y detección del rebose

Sección 4-2 Sumadores de alta velocidad

La naturaleza secuencial de la propagación del arrastre es el problema más difícil que hay que resolver cuando se desea acelerar la suma. Las diferentes estrategias que se han propuesto para resolver este problema se pueden clasificar de la forma siguiente:

1) *Aceptación de la existencia de los arrastres.* En lugar de intentar acelerar la generación de los arrastres simplemente se les acepta tal como son. Se diseña el sumador de manera que su salida no se utiliza hasta que haya pasado el tiempo suficiente para que se generen los arrastres cuando se produce su secuencia más desfavorable (es decir que los arrastres se hayan propagado desde el bit menos significativo al más significativo de los números a sumar). En algunos microprocesadores de 8 bits su frecuencia de reloj se ajusta de forma tal, que las salidas del sumador no se utilizan hasta que la secuencia más larga de posibles arrastres ha tenido tiempo de propagarse, esté o no realmente presente.

2) *Anticipación del arrastre.* Los sumadores de esta clase poseen una circuitería extra que "desvía" la generación de los arrastres de las etapas del sumador paralelo (o grupo de etapas) que lo que hacen es propagarlos.

3) *Suma condicional.* Análogamente a los sumadores con anticipación de arrastre, este tipo de sumadores produce la suma en un tiempo $O(\log_2 n)$, siendo n la longitud de los operandos. En pasos sucesivos se obtienen 1, 2, 4, ..., 2^k dígitos correctos hasta que se completa la suma.

4) *Detección de la finalización del arrastre.* Añadiendo un circuito lógico adicional es posible detectar cuando ha finalizado la propagación de los arrastres a lo largo del sumador. En ese momento las salidas del sumador tendrán su valor final correcto y se podrán utilizar.

5) *Minimización del número de arrastres.* Cuanto mayor sea la base del sistema de numeración utilizado para representar los operandos, menor será el número de arrastres que se producen durante la suma. Por este motivo se pueden considerar sistemas de numeración de base 2^k, para los que k dígitos binarios consecutivos corresponden a un dígito en dicha base. La operación de suma se efectúa directamente en la nueva base.

6) *Arrastre almacenado.* Cuando el número de sumandos es mayor que 2 es posible retardar la propagación del arrastre hasta el final de la operación. De esta manera, se puede considerar que el tiempo necesitado para la asimilación de los arrastres diferidos se reparte uniformemente entre todos los operandos.

4.2.1 Características de los arrastres

Antes de considerar algunos de los métodos mencionados en el apartado anterior para acelerar la suma, conviene examinar con cierto detalle las características de los arrastres.

a) Un arrastre se *generará* en la posición *i*-ésima si se cumple que la suma aritmética de los bits correspondientes (x_i e y_i) es mayor que uno, es decir, si $x_i + y_i > 1$.

b) Un arrastre que llega a la posición *i*-ésima desde la posición (*i-1*)-ésima, se *propagará* a la posición (*i*+1)-ésima si se cumple que la suma aritmética de los bits correspondientes (x_i e y_i) es igual a uno, es decir, si $x_i + y_i = 1$.

	$y_i = 0$	$y_i = 1$
$x_i = 0$	-	p
$x_i = 1$	p	g

Tabla 4.6: Tipos de arrastres

La Tabla 4.6 muestra de forma resumida las condiciones bajo las cuales no hay arrastres (-), se propaga el arrastre recibido (*p*) o se genera un arrastre (*g*). Estas tres acciones (-, *p* y *g*) son mutuamente excluyentes, de manera que solamente puede ocurrir una de ellas en cada posición. De la Tabla 4.6 se sigue que aquellas etapas del sumador para las que $x_i = y_i = 1$ comenzarán una secuencia de arrastre. La propagación de los arrastres continuará a través de las etapas en las que $x_i \neq y_i$ y parará cuando llegue a una etapa en la que $x_i = y_i$. El ejemplo de la Figura 4.14 muestra 4 secuencias de arrastre de longitudes 2, 4, 1 y 2 que comienzan todas simultáneamente.

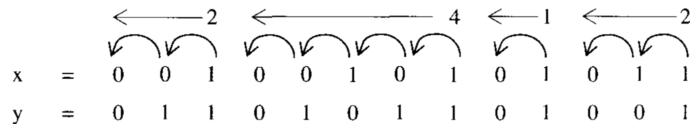


Figura 4.14: Secuencias de arrastres

Esta última consideración, de la concurrencia de la secuencia de arrastres, es importante y se usa en algunas estrategias para acelerar la suma (sumadores con detección del final de los arrastres). La secuencia de arrastre de longitud 2 a la izquierda de la Figura 4.14 y su adyacente de longitud 4 no es una única secuencia de longitud 6 sino dos secuencias simultáneas.

4.2.2 Sumadores con anticipación del arrastre

Es la forma más extendida de diseñar sumadores de alta velocidad. Su principio básico consiste en reducir el retardo producido por la propagación de los arrastres de los SBC's de menor peso a los de mayor peso. La forma de hacerlo es generando la entrada de arrastre de la etapa *i*-ésima directamente a partir de los bits de entrada a las etapas precedentes *i* - 1, *i* - 2, ..., 1, 0 en lugar de tener que esperar a que el arrastre se propague a lo largo de dichas etapas (de ahí el nombre de anticipación del arrastre).

En la ecuación (4.2) de la salida c_i del arrastre de un SBC cuando se realiza a partir de dos SSB's, se pueden distinguir dos términos. El primero de ellos $x_i y_i$ depende solamente de las entradas de la etapa i y corresponde al arrastre generado en dicha etapa (g_i). El segundo de los términos establece que un arrastre procedente de la etapa previa (c_{i-1}) se propagará a la siguiente si $x_i \oplus y_i = 1$ y representa[2] el arrastre propagado por la etapa i (p_i). Por tanto la suma s_i y el arrastre c_i de la etapa i se pueden expresar por las expresiones:

$$s_i = p_i \oplus c_{i-1} \qquad c_i = g_i + p_i c_{i-1} \tag{4.5}$$

Si se observa la realización del SBC a partir de dos SSB's (ver Figura 4.8) se comprueba que el primer SSB genera los términos g_i y p_i, por lo que se puede diseñar un SBC modificado que añade a sus salidas estos términos tal como se muestra en la Figura 4.15.

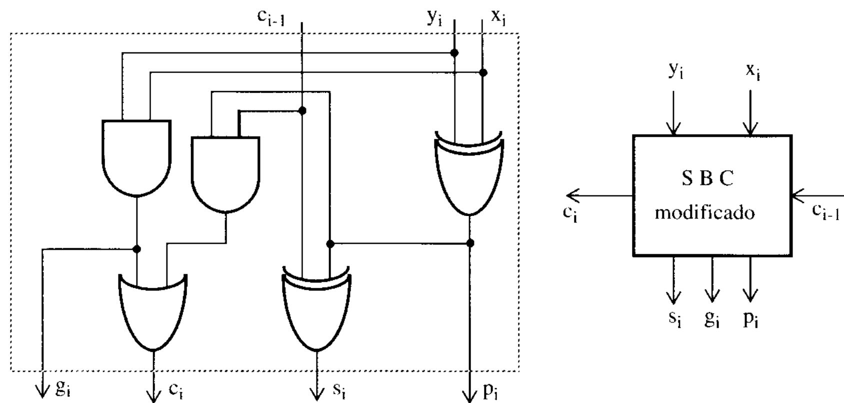


Figura 4.15: SBC modificado

De la ecuación (4.5) se deduce que todos los bits de la suma se generan en paralelo si todas las entradas de arrastre $c_{n-2}, ..., c_0, c_{-1}$ están disponibles de forma simultánea. Si se aplica la ecuación (4.5) recursivamente se tiene:

$$c_0 = g_0 + p_0 c_{-1}$$

$$c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{-1}$$

$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{-1}$$

....

$$c_i = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + ... + p_i p_{i-1} ... p_1 g_0 + p_i p_{i-1} ... p_0 c_{-1}$$

como puede apreciarse, cada término producto que aparece en c_i es la conjunción de generación y propagaciones de arrastres. Así, en la etapa i, se produce un arrastre c_i si:

1) La etapa i lo genera ($g_i = 1$)

2. En algunos textos se utiliza en lugar de la función de propagación del arrastre $p_i = x_i \oplus y_i$ la función de *transferencia del arrastre* $t_i = x_i + y_i$.

2) La etapa $i - k$ ($k = 1, 2,..., i$) lo genera ($g_{i-k} = 1$) y las etapas desde $i - k + 1$ hasta i, lo propagan ($p_{i-k+1} = 1, ..., p_{i-1} = 1, p_i = 1$)
3) Hay un arrastre inicial ($c_{-1} = 1$) y todas las etapas lo propagan ($p_0 = 1, ..., p_{i-1} = 1, p_i = 1$)

En la Figura 4.16 se muestra un circuito combinacional que realiza las ecuaciones anteriores para el caso de 4 bits. A este circuito se le denomina *circuito de aceleración de arrastres* (CAA). El retardo para la obtención de los arrastres en este circuito es de sólo dos niveles de puertas lógicas.

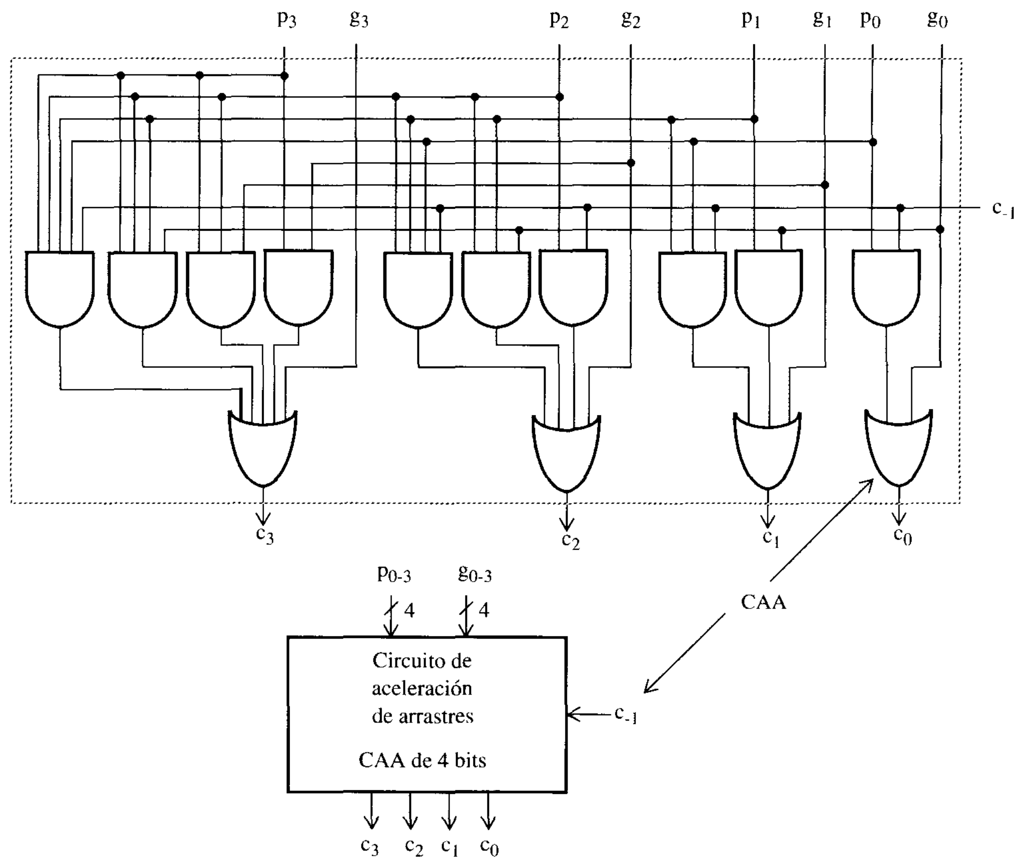


Figura 4.16: Circuito de aceleración de arrastres (CAA) de 4 bits

Si se combinan 4 módulos del tipo SBC modificado de la Figura 4.15 y el CAA de la Figura 4.16 se obtiene un *sumador con aceleración de los arrastres* (SAA) de 4 bits, tal como se muestra en la Figura 4.17. Se puede establecer la siguiente relación:

1 SAA de n bits = {n SBC's modificados} + 1 CAA de n bits

A medida que aumenta el número de bits del sumador va creciendo la complejidad del módulo CAA, por lo que en la práctica, para sumadores con un número elevado de bits no resulta viable la aplicación estricta de esta técnica. Una solución a este problema consiste en encadenar m SAA's de 4 bits con propagación de los arrastres (como el de la Figura 4.17) para formar sumadores de $n = 4 \times m$ bits. En la Figura 4.18 se muestra un sumador de 16 bits construido con esta técnica.

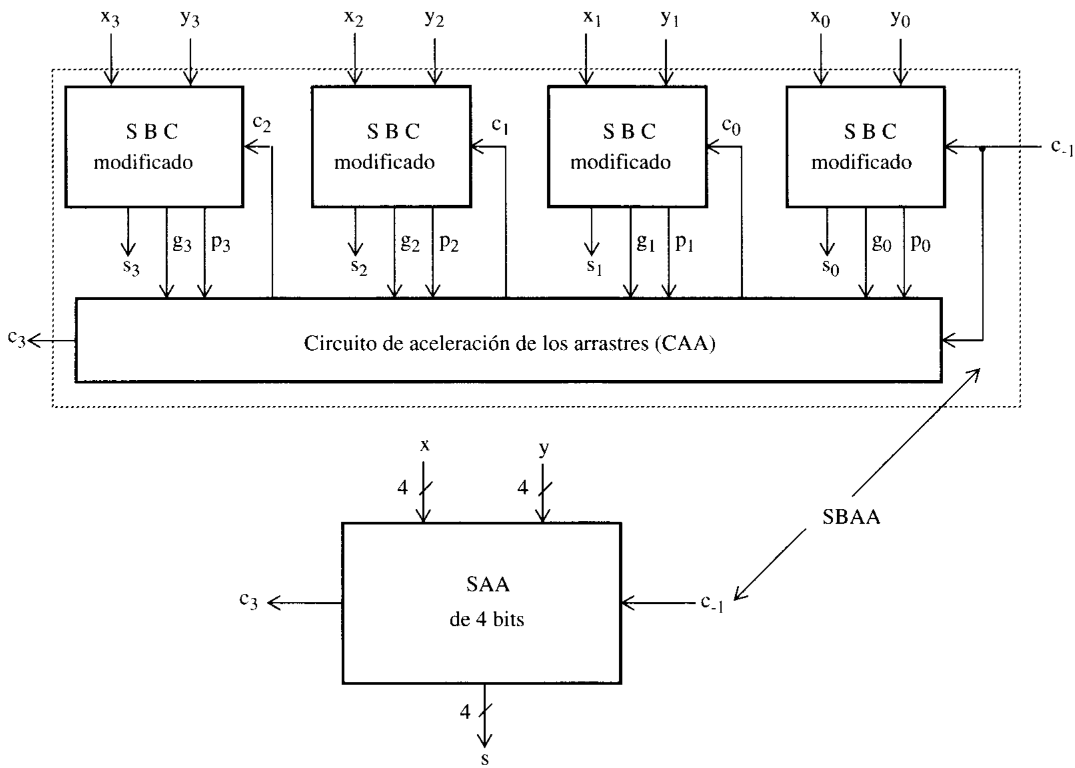


Figura 4.17: Sumador con aceleración de los arrastres de 4 bits (SAA)

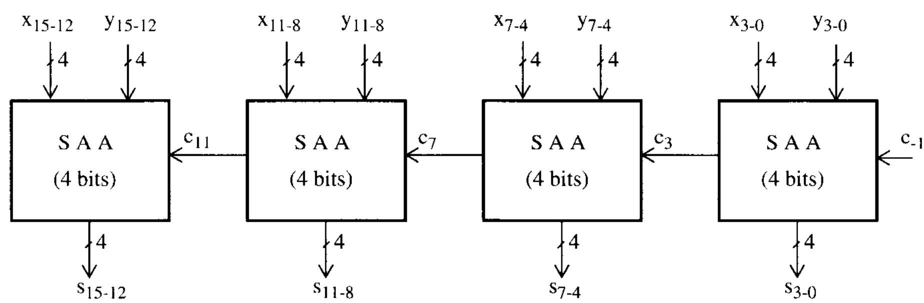


Figura 4.18: Sumador de 16 bits construido con 4 SAA's de 4 bits

Se observa que el SAA de mayor peso no puede dar su resultado mientras no conozca c_{11}, pero siempre el retardo que se incurre en la propagación de los arrastres es menor que si se dispone de 16 SBC's en cascada. A este tipo de sumadores se les conoce como *sumadores con propagación anticipada del arrastre*.

4.2.3 Sumadores de suma condicional

El principio en el que se fundamenta el sumador de suma condicional es generar dos conjuntos de salidas. Cada conjunto incluye k bits de suma y un arrastre de salida. Uno de los conjuntos asume que eventualmente el arrastre de entrada será cero, mientras que el otro supone que será uno. Una vez que se conozca el arrastre de entrada, se necesita seleccionar solamente el conjunto correcto de salidas (uno de los dos conjuntos calculados) sin tener que esperar a que el arrastre se propague a lo largo de las k posiciones. Obviamente no se debería aplicar esta idea a todos los n bits de los operandos al comienzo de la operación de suma ya que entonces se tendría que esperar hasta que el arrastre se propague a través de los n bits antes de que se pueda hacer la selección. Se necesita por lo tanto dividir los n bits dados en grupos más pequeños y aplicar la idea anterior a cada uno de ellos por separado. De esta forma, se puede hacer en paralelo la propagación del arrastre dentro de cada uno de los grupos lo que reduce el tiempo total de ejecución. Estos grupos se pueden subdividir aún más en subgrupos, para los que el tiempo de propagación del arrastre es incluso más pequeño. Las salidas de los subgrupos se combinan entonces para generar la salida de los grupos.

Una división natural de los n bits de los operandos sería en dos grupos de $n/2$ bits. Cada uno de estos grupos se puede subdividir en dos subgrupos de $n/4$ bits. Este proceso puede, en principio, continuarse hasta que se alcanza un grupo de tamaño 1 si n es una potencia de 2. En este caso se necesitan $k = \log_2 n$ pasos en el proceso, donde en el paso 1 se consideran los bits de los operandos de forma individual, en el paso 2 se tratan parejas de bits y así sucesivamente. El sumador de suma condicional proporciona así el resultado correcto de la suma en un tiempo $O(\log_2 n)$. Conviene observar, sin embargo, que un grupo dado no tiene necesariamente que dividirse en subgrupos de igual tamaño. Así pues el esquema de suma condicional se puede aplicar incluso si el número de bits no es una potencia de 2. Se ilustra a continuación la forma en que los grupos de 1 sólo bit se combinan en parejas de bits. Se utiliza la siguiente notación:

s_i^0 = bit de suma en la posición i, si el arrastre de entrada al grupo es 0

s_i^1 = bit de suma en la posición i, si el arrastre de entrada al grupo es 1

c_{i+1}^0 = arrastre de salida del grupo, si el arrastre de entrada al grupo es 0

c_{i+1}^1 = arrastre de salida del grupo, si el arrastre de entrada al grupo es 1

En la Tabla 4.7 se consideran dos posiciones de bits adyacentes. En el paso 1 cada una de las posiciones constituye un grupo separado.

i	7	6	
x_i	1	0	
y_i	0	0	
s_i^0	1	0	Suponiendo que el arrastre de entrada = 0
c_{i+1}^0	0	0	
s_i^1	0	1	Suponiendo que el arrastre de entrada = 1
c_{i+1}^1	1	0	

Tabla 4.7: Paso 1 del algoritmo de suma condicional para dos posiciones de bits adyacentes

En el paso 2, los dos bits se combinan en un grupo de tamaño 2 (ver Tabla 4.8)

$i, i-1$	7, 6	
x_i, x_{i-1}	1 0	
y_i, y_{i-1}	0 0	
s_i^0, s_{i-1}^0	1 0	Suponiendo que el arrastre de entrada es 0
c_{i+1}^0	0	
s_i^1, s_{i-1}^1	1 1	Suponiendo que el arrastre de entrada es 1
c_{i+1}^1	0	

Tabla 4.8: Paso 2 del algoritmo de suma condicional para una pareja de bits

En la Tabla 4.9 se muestra un ejemplo de suma condicional de dos números de 8 bits. El proceso tiene $\log_2 8 = 3$ pasos. El arrastre forzado (que es igual a 0 en este ejemplo) está disponible al comienzo de la operación, por lo que sólo se necesita que se genere un conjunto de salidas para el grupo de más a la derecha en cada paso.

	i	7	6	5	4	3	2	1	0
	x_i	1	0	1	1	0	1	1	0
	y_i	0	0	1	0	1	1	0	1
Paso 1	s_i^0	1	0	0	1	1	0	1	1
	c_{i+1}^0	0	0	1	0	0	1	0	0
	s_i^1	0	1	1	0	0	1	0	
	c_{i+1}^1	1	0	1	1	1	1	1	
Paso 2	s_i^0	1	0	0	1	0	0	1	1
	c_{i+1}^0	0		1		1		0	
	s_i^1	1	1	1	0	0	1		
	c_{i+1}^1	0		1		1			
Paso 3	s_i^0	1	1	0	1	0	0	1	1
	c_{i+1}^0	0				1			
	s_i^1	1	1	1	0				
	c_{i+1}^1	0							
Resultado		1	1	1	0	0	0	1	1

Tabla 4.9: Suma condicional de dos números de 8 bits

4.2.4 Sumadores con selección del arrastre

El sumador con selección del arrastre se basa esencialmente en el mismo principio que el sumador de suma condicional, es decir en la generación de más de un conjunto de bits de suma seguida por la selección de uno de ellos de acuerdo con los arrastres determinados. En realidad, el sumador de suma condicional se puede considerar como un caso especial del sumador con selección del arrastre. En el primero la formación inicial de los bits de suma ocurre con grupos de 1 bit y la selección de los bits de suma que son correctos es a través de un árbol en el que el tamaño del grupo se duplica en cada paso. En el segundo la formación inicial de los bits de suma ocurre en grupos de m bits ($m > 1$) y el tamaño del grupo utilizado durante la selección del arrastre puede aumentar por un factor mayor que dos.

En la Figura 4.19 se muestra un sencillo ejemplo de suma de dos números de 8 bits que ilustra el funcionamiento de los sumadores con selección del arrastre. El sumador de los 4 bits menos significativos puede ser un SPA (sumador con propagación del arrastre) o un SAA (sumador con anticipación del arrastre). Simultáneamente los 4 bits más significativos se suman una vez suponiendo que el arrastre de entrada es 1 y otra que es 0. Cuando se conoce el arrastre de salida de los 4 bits menos significativos, se puede utilizar para seleccionar mediante un multiplicador cuál de las dos secuencias calculadas es la correcta.

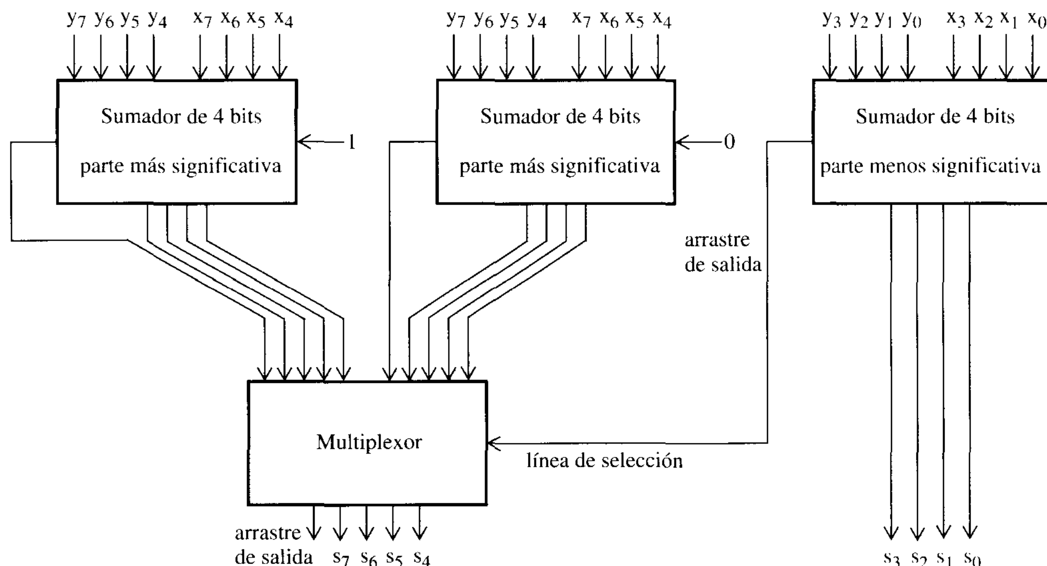


Figura 4.19: Diagrama de bloques de un sumador con selección del arrastre de 8 bits

4.2.5 Sumadores con detección de la finalización del arrastre

El sumador paralelo con *propagación del arrastre* se basa en una concepción síncrona (intervalo de suma constante para cualquier configuración de los operandos) que tiene en cuenta el caso más desfavorable de la propagación del arrastre. Esto implica que el tiempo de operación de la suma depende de forma lineal de n (número de bits de los sumandos).

Como un arrastre de máxima longitud (es decir, que se genera en la primera etapa y se propaga por todas las demás) no se da frecuentemente, en la mayoría de los casos sucederá que se espera de forma inútil un tiempo excesivo para el cálculo de la suma.

Ya en 1946 Burks, Goldstine y von Neumann, en su trabajo clásico sobre computadores con programa almacenado, analizaron el problema de la propagación del arrastre (usando números de 40 bits generados aleatoriamente), y demostraron que el "valor medio de la longitud máxima de propagación del arrastre" en la suma de dos números binarios de n bits es del orden de $\log_2 n$. Para $n = 32$ el valor medio será ≤ 5, mientras que para $n = 64$ será ≤ 6. La relación entre las longitudes de propagación del arrastre para el caso más desfavorable y el valor medio de la longitud máxima viene expresada por:

$$\tau = \frac{n}{\log_2 n}$$

Para $n = 16$ bits el retardo en un sumador con propagación del arrastre es de 32 (2 por etapa), mientras que en un sumador con detección de fin de arrastre el retardo es en promedio de 8 ($\log_2 16 = 4$ etapas $\times$ 2 retardo/etapa). Esto supone que en media la realización de la suma es cuatro veces más rápida.

El diseño de esta clase de sumador se basa en incorporar dentro del mismo, una unidad que detecte cuándo ha terminado la operación, y son por lo tanto de naturaleza asíncrona (ya que no es necesario un intervalo fijo para el tiempo de suma).

Además de los SBC's hay que añadir en cada etapa un circuito lógico que detecta los finales de arrastre. Cuando los sumadores individuales hayan acabado producirán un arrastre a la siguiente etapa (c_i) o un no arrastre (nc_i). La aparición de un c_i o un nc_i indica que esa etapa ha funcionado y cuando sucede en todas, señala la terminación de la operación. Al empezar la suma no existen ninguno de los dos arrastres (c_i o nc_i). En la Tabla 4.10 se dan las configuraciones que provocan la generación de c_i o nc_i y las de propagación de c_i.

x_i	y_i	c_{i-1}	c_i	nc_i	Arrastre
0	0	0	0	1	No se genera
0	0	1	0	1	$g_{0i} = \bar{x}_i \bar{y}_i$
0	1	0	0	0	
0	1	1	1	1	Se propaga
1	0	0	0	0	$p_i = x_i \oplus y_i$
1	0	1	1	1	
1	1	0	1	0	Se genera
1	1	1	1	0	$g_{1i} = x_i y_i$

Tabla 4.10: Generación de c_i y nc_i

Para controlar el comienzo de la suma se añade una señal I de inhibición. Tanto c_i como nc_i se inicializan a cero haciendo $I = 0$. Cuando $I = 1$ empieza la suma. Las expresiones lógicas de las señales c_i y nc_i son:

$$c_i = (x_i \oplus y_i)\, c_{i-1} + x_i\, y_i\, I = g_{1i}\, I + p_i\, c_{i-1} \tag{4.6}$$

$$nc_i = (x_i \oplus y_i)\, nc_{i-1} + \bar{x}_i\, \bar{y}_i\, I = g_{0i}\, I + p_i\, nc_{i-1} \forall\, i \neq 0 \tag{4.7}$$

y para $i = 0$:

$$c_0 = (x_0 \oplus y_0)\,(c_{-1}\,I) + x_0\,y_0\,I \qquad (4.8)$$

$$nc_0 = (x_0 \oplus y_0)\,(nc_{-1}\,I) + \overline{x}_0\,\overline{y}_0\,I \qquad (4.9)$$

el primer término de c_0 y nc_0 incluyen I, para garantizar que antes de empezar, cuando $I = 0$, se cumpla que $c_i = nc_i = 0\ \forall\ i$. En la Figura 4.20 se muestra el módulo de propagación del arrastre (PA) que realiza la síntesis de las ecuaciones (4.6) y (4.7).

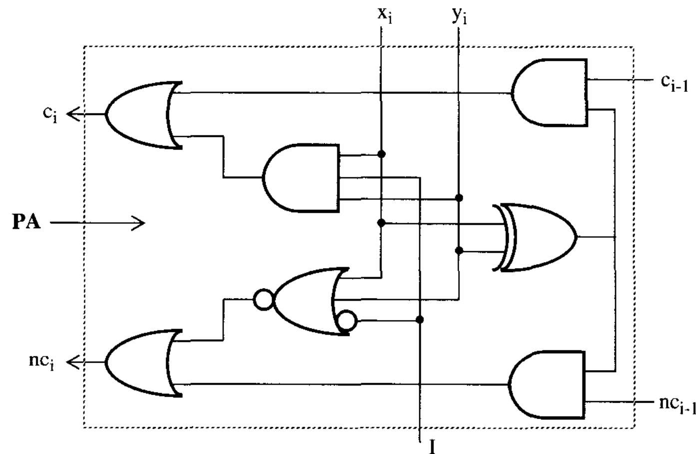


Figura 4.20: Celda de propagación del arrastre (PA)

Con el módulo PA se puede diseñar un sumador de n bits que detecta el fin del arrastre (ver Figura 4.21). La señal FS indica el fin de la suma porque en todos los módulos c_i ó nc_i son unos.

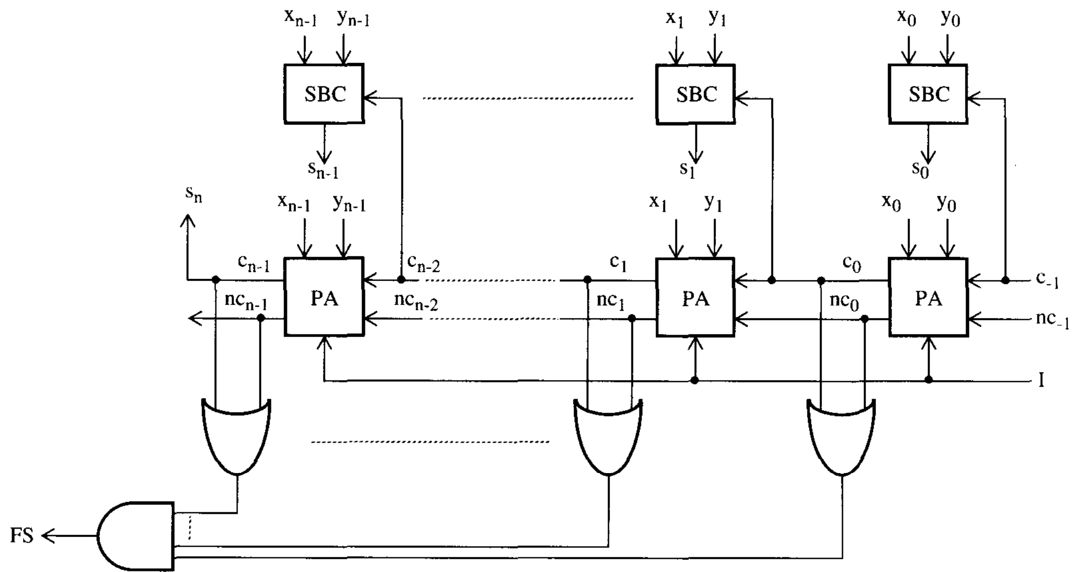


Figura 4.21: Sumador de n-bits con detector de fin de suma FS

Se genera un primer arrastre en la etapa i si $x_i = y_i = 1$, y se propaga hacia la izquierda pasando a través de las etapas donde $x_i\ y_i$ = 10 ó 01. Para evitar indicaciones erróneas deben darse simultáneamente las dos entradas al sumador. Cuando ha empezado la operación no pueden admitirse cambios en las entradas.

4.2.6 Sumadores que minimizan el número de arrastres

Cuando se suman dos números de 64 bits en un SBAA, la secuencia de propagación de arrastres más desfavorable es a través de las 64 etapas de que consta el sumador. Si los números se expresan en base 4, en lugar de hacerlo en base 2, tienen una longitud de 32 dígitos, por lo que la suma aritmética efectuada en dicha base también reduce la secuencia de arrastre más desfavorable a una longitud de 32.

En general, si se representan los números en base 2^p tendrán una longitud de $64/p$ dígitos, y si la suma aritmética se realiza en esa misma base la longitud de la secuencia de arrastre más desfavorable será de $64/p$. A pesar de la reducción que se produce en el tamaño de la secuencia de propagación de los arrastres a lo largo del sumador, cuando se opera en sistemas de numeración de base elevada, la circuitería lógica que se necesita para efectuar la suma aritmética en estas bases, es mucho más compleja que la empleada cuando se utiliza base 2. Además, es difícil conseguir circuitos aritméticos tan rápidos como los binarios.

Por ejemplo, un sumador que acepta como entrada dos grupos de 4 bits (cada grupo representa un dígito en base 16), tiene 9 entradas (incluyendo un arrastre de entrada) y 5 salidas (4 de la suma y 1 del arrastre de salida). No resulta viable realizar este circuito en dos niveles de puertas AND-OR ya que desde un punto de vista práctico no se construyen este tipo de puertas con más de 6 entradas.

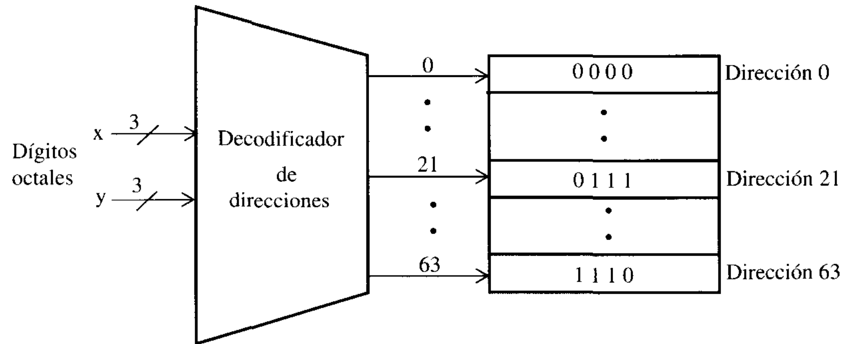
Una forma más atractiva de realizar la suma aritmética en bases superiores a 2 es utilizando una *tabla de consulta*, esto es, se proporciona una tabla que almacena el valor de la suma y de los arrastres para todos los posibles pares de sumandos. Si los operandos de entrada son números de m bits, dicha tabla tendría 2^m filas × 2^m columnas y 2^{2m} entradas con $(m+1)$ bits/entrada (ver Tabla 4.11 para el caso de una base octal, $m = 3$).

	$\mathbf{0_8 = 000_2}$	$\mathbf{1_8 = 001_2}$	$\mathbf{2_8 = 010_2}$	$\mathbf{3_8 = 011_2}$	$\mathbf{4_8 = 100_2}$	$\mathbf{5_8 = 101_2}$	$\mathbf{6_8 = 110_2}$	$\mathbf{7_8 = 111_2}$
$\mathbf{0_8 = 000_2}$	$000\,000_2$	$000\,001_2$	$000\,010_2$	$000\,011_2$	$000\,100_2$	$000\,101_2$	$000\,110_2$	$000\,111_2$
$\mathbf{1_8 = 001_2}$	$000\,001_2$	$000\,010_2$	$000\,011_2$	$000\,100_2$	$000\,101_2$	$000\,110_2$	$000\,111_2$	$001\,000_2$
$\mathbf{2_8 = 010_2}$	$000\,010_2$	$000\,011_2$	$000\,100_2$	$000\,101_2$	$000\,110_2$	$000\,111_2$	$001\,000_2$	$001\,001_2$
$\mathbf{3_8 = 011_2}$	$000\,011_2$	$000\,100_2$	$000\,101_2$	$000\,110_2$	$000\,111_2$	$001\,000_2$	$001\,001_2$	$001\,010_2$
$\mathbf{4_8 = 100_2}$	$000\,100_2$	$000\,101_2$	$000\,110_2$	$000\,111_2$	$001\,000_2$	$001\,001_2$	$001\,010_2$	$001\,011_2$
$\mathbf{5_8 = 101_2}$	$000\,101_2$	$000\,110_2$	$000\,111_2$	$001\,000_2$	$001\,001_2$	$001\,010_2$	$001\,011_2$	$001\,100_2$
$\mathbf{6_8 = 110_2}$	$000\,110_2$	$000\,111_2$	$001\,000_2$	$001\,001_2$	$001\,010_2$	$001\,011_2$	$001\,100_2$	$001\,101_2$
$\mathbf{7_8 = 111_2}$	$000\,111_2$	$001\,000_2$	$001\,001_2$	$001\,010_2$	$001\,011_2$	$001\,100_2$	$001\,101_2$	$001\,110_2$

Tabla 4.11: Suma de 2 números en base octal

Se necesita pues una memoria ROM de 2^{2m} palabras × $(m + 1)$ bits (de la Tabla 4.11 no hay que almacenar los dos bits más significativos ya que son siempre iguales a 0). Los 2 sumandos concatenados dan la dirección de memoria cuyo contenido es precisamente su suma (ver Figura 4.22 para el ejemplo anterior).

Esta técnica de emplear una ROM como tabla de consulta en la síntesis de circuitos combinacionales, encuentra múltiples aplicaciones en otros módulos aritméticos tales como los sumadores con múltiples operandos y los multiplicadores binarios que se verán posteriormente.



Ejemplo $x = 0\,1\,0$, $y = 1\,0\,1$ $\Rightarrow$ $x + y = 0\,1\,1\,1$ es el contenido de la dirección $xy = 0\,1\,0\,1\,0\,1_2 = 21_{10}$

Figura 4.22: ROM de 64 palabras × 4 bits/palabra para sumar 2 dígitos octales

4.2.7 Sumadores con arrastre almacenado

La suma de dos números binarios de n bits x^0 e y^0 se puede ver como un proceso iterativo en el que en cada paso i las salidas que se producen x^i e y^i, $i = 1, 2, ...$ viene dada por:

$$x^i(j) = x^{i-1}(j) \oplus y^{i-1}(j-1) \qquad j = 0, 1,.., n-1$$

$$y^i(j) = x^{i-1}(j) \text{ AND } y^{i-1}(j-1) \qquad j = 0, 1,.., n-1$$

donde $x^i(j)$ e $y^i(j)$ indican el bit j de las salidas en la etapa i. Las dos ecuaciones anteriores representan la función de un SSB que recibe como entrada el bit de suma del paso previo ($x^{i-1}(j)$) y el bit de arrastre del paso previo desplazado un lugar a la izquierda ($y^{i-1}(j-1)$). A continuación, en un ejemplo con dos numeros de 5 bits se muestra paso a paso el proceso iterativo de la suma de acuerdo con las expresiones anteriores.

0 1 1 0 1	1ersumando x^0
0 0 1 0 1	2° sumando y^0
0 1 0 0 0	1ª suma parcial x^1
0 0 1 0 1	1erarrastre parcial y^1
0 1 0 1 0	arrastre parcial desplazado
0 0 0 1 0	2^a suma parcial x^2
0 1 0 0 0	2° arrastre parcial y^2
1 0 0 0 0	arrastre parcial desplazado
1 0 0 1 0	3ª suma parcial x^3
0 0 0 0 0	3er arrastre parcial y^3 (todos ceros)

En este punto finaliza la suma porque el arrastre parcial es 0 en todas las posiciones. El procedimiento se puede mecanizar utilizando de forma repetida un grupo de SSB's o mediante una batería de sucesivos grupos de SSB's. En la Figura 4.23 se muestra un diagrama de bloques de esta solución. Inicialmente los 2 registros R_a y R_b contienen los números x e y. En los pasos siguientes, los contenidos de estos registros se sustituyen por la suma parcial x^j y el arrastre parcial y^j desplazado un lugar hacia la izquierda.

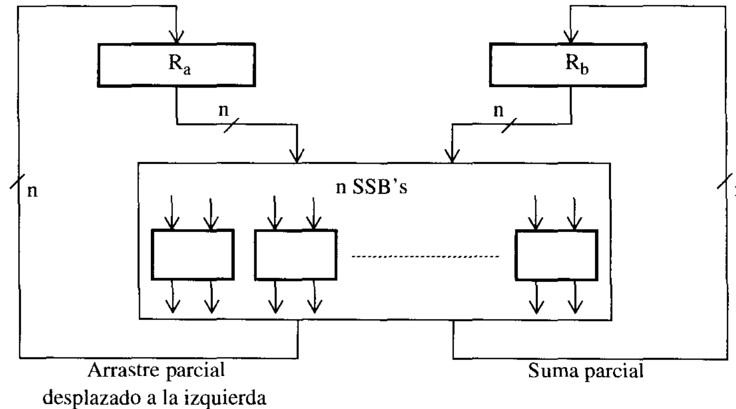


Figura 4.23: Suma de 2 números con arrastre almacenado

Este circuito no resulta útil, pues aunque emplea solamente n SSB's, es más lento que si se emplean n SBC's conectados directamente (ver Figura 4.10) debido al tiempo para las transferencias a los registros.

Cuando se tienen que sumar tres o más operandos simultáneamente (por ejemplo en la multiplicación) utilizando sumadores con sólo dos operandos, el tiempo que se emplea en la propagación del arrastre debe repetirse algunas veces. Si el número de operandos es k, entonces los arrastres tienen que propagarse $(k - 1)$ veces. Se han propuesto e implementado diferentes técnicas para la suma de operandos múltiples que intentan reducir el retardo que impone la propagación de los arrastres. Una de las más utilizadas es la de los *sumadores con arrastre almacenado* o *sumadores CSA* (Carry Save Adder). En esta clase de sumadores el arrastre sólo se propaga en el último paso del proceso de suma, mientras que en los pasos anteriores se genera de forma separada una suma parcial y una secuencia de arrastres. Así, un CSA toma como entrada tres operandos de n bits y genera como resultado dos secuencias de n bits cada una, la primera corresponde a la suma parcial y la segunda a la secuencia de los arrastres. Un segundo CSA acepta estas dos secuencias anteriores y otro operando de entrada y produce como salida una nueva suma parcial y una nueva secuencia de los arrastres. Un CSA es por lo tanto capaz de reducir el número de operandos que se suman de 3 a 2 sin utilizar ninguna propagación de los arrastres.

Un CSA se puede implementar de diferentes maneras. En su forma más simple, el elemento básico de un CSA es el SBC (sumador binario completo) con tres entradas x, y y z. La operación aritmética que realiza se puede describir como:

$$x + y + z = 2c + s$$

donde s y c son la suma y el arrastre respectivamente. Sus valores son:

$$s = (x + y + z) \bmod 2 \qquad \text{y} \qquad c = \frac{(x + y + z) - s}{2}$$

Las salidas son la representación binaria del número de 1´s que hay en las entradas. De esta forma se puede considerar, tal como se muestra en la Figura 4.24, que un SBC es un contador C(3,2). El primer número indica el número de entradas y el segundo el número de salidas que es preciso que tenga el módulo para poder contar cuantos 1´s hay presentes en sus entradas. *Un CSA de n bits consiste en n C(3,2) que operan en paralelo y que no poseen interconexiones entre sus arrastres de salida.* Esta idea de los contadores se puede generalizar a un $C(n,m)$ donde $m = \lceil \log_2 n \rceil$ ($\lceil x \rceil$ representa el menor entero mayor o igual a x).

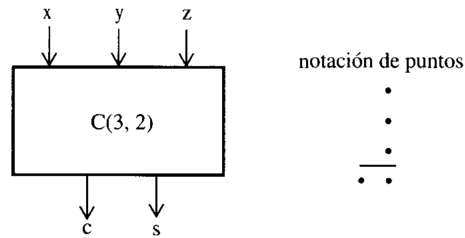


Figura 4.24: Diagrama de bloques y representación de puntos de un SBC

En la Figura 4.25 se muestra un CSA para la suma de 4 operandos x, y, z y w de 4 bits cada uno. Los dos niveles superiores son CSA´s de 4 bits, mientras que el tercer nivel es un sumador con propagación de los arrastres (SPA). Sin embargo este último nivel, para acelerar el proceso de la suma, se puede sustituir por un sumador con anticipación de los arrastres (SAA) o cualquier otro esquema de sumador de alta velocidad que se desee. Debe observarse que los bits de la suma parcial y los bits de los arrastres se interconectan de forma tal que se garantiza que sólo los bits con el mismo peso se suman por cualquiera de los contadores $C(3,2)$.

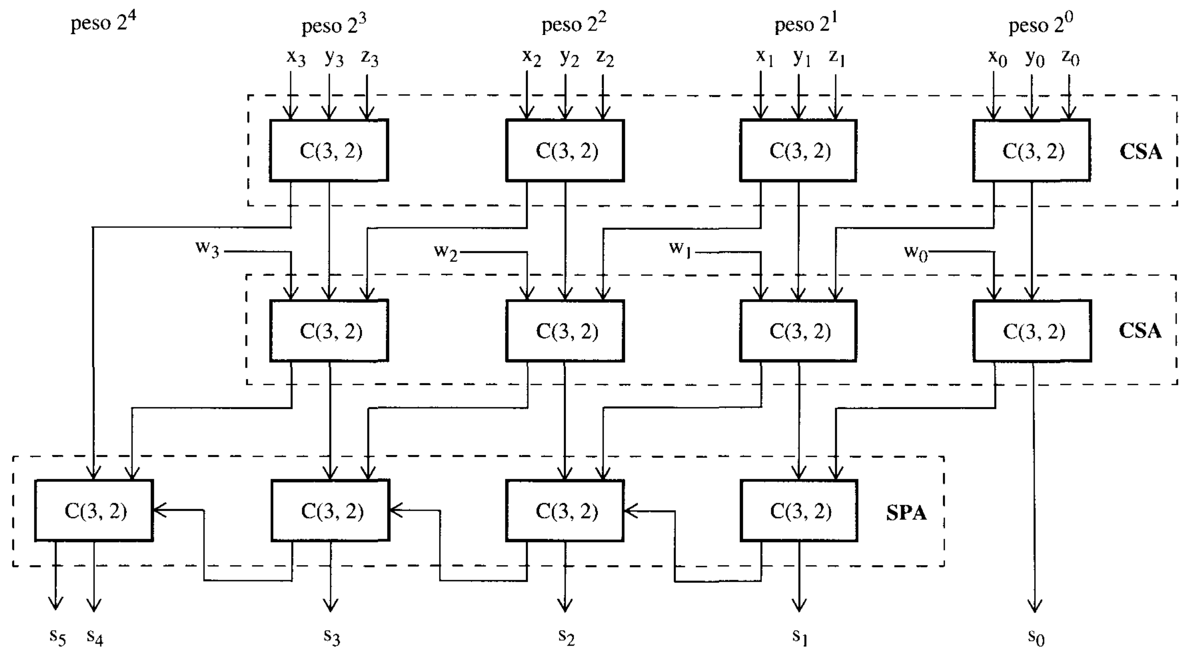


Figura 4.25: Un sumador CSA para 4 operandos de 4 bits

Para sumar los k operandos $x_1, x_2, ..., x_k$ se necesitan (k - 2) unidades de CSA´s y un SAA. Si los CSA´s se disponen en cascada como en la Figura 4.25, entonces el tiempo para sumar los k operandos es:

$$\tau = (k-2)\tau_{CSA} + \tau_{SPA}$$

donde τ_{SPA} es el retardo de operación de un SPA y τ_{CSA} el retardo de un CSA que es igual según lo visto al de un SBC. El resultado final puede requerir una longitud de $n + \lceil \log_2 k \rceil$ bits, ya que la suma de k operandos de n bits cada uno puede ser tan grande como $(2^n - 1)k$.

Una forma mejor de organizar los CSA´s y reducir el tiempo de operación es emplear una estructura arborescente denominada *árbol de Wallace*. Este tipo de organización se estudiará en la sección 4-5 cuando se consideren los multiplicadores de alta velocidad.

La Figura 4.26 muestra como utilizar solamente un nivel de CSA para sumar un conjunto de números de n bits. En primer lugar se aplican tres números a las entradas x, y y z. La suma y el arrastre generado por estos tres números se realimentan a las entradas x e y que se suman con el cuarto número que se introduce en la entrada z. Este proceso continúa hasta que se suman todos los números.

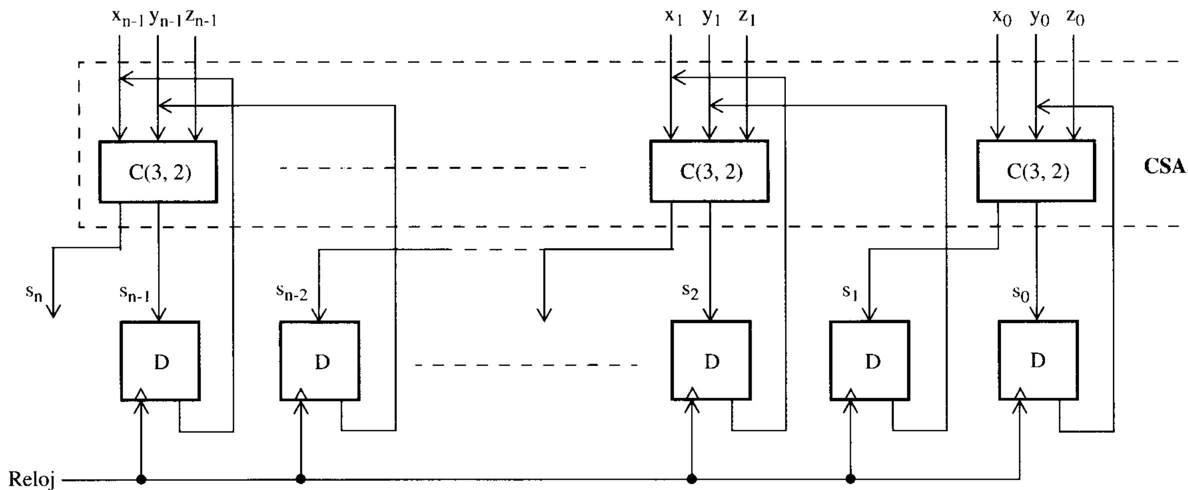


Figura 4.26: Diagrama de bloques de un CSA de n bits y de un sólo nivel que suma un conjunto de números

Sección 4-3

Sumadores en código BCD

Los sumadores que operan sobre números decimales codificados en binario, como el código BCD, son más complejos que los sumadores binarios. Los sumadores en código BCD se pueden diseñar como si fueran sumadores binarios, añadiéndoles unos circuitos de corrección que garanticen la correcta codificación de los resultados.

Si se suman dos dígitos BCD en un sumador binario de 4 bits el resultado es correcto si es menor que 10. El ejemplo siguiente muestra esta situación:

```
        x = 2          x = 0 0 1 0
        y = 6          y = 0 1 1 0
  -------------   -----------------
  c = 0  s = 8    c = 0  s = 1 0 0 0   (correcto)
```

Sin embargo, cuando la suma es mayor o igual que 10, no es correcta porque no representa a ningún dígito codificado en BCD y debe ser corregida. Es fácil comprobar que la corrección consiste en añadir el dígito BCD 6 al resultado anterior. El ejemplo que se da a continuación corresponde a este caso.

```
        x = 7          x = 0 1 1 1
        y = 8          y = 1 0 0 0
  -------------   -----------------
  c =1   s = 5    c = 0  s = 1 1 1 1   (incorrecto)
                       + 0 1 1 0       (sumar 6)
                  -----------------
                  c = 1  s = 0 1 0 1   (correcto)
```

En la Figura 4.27 se muestra el circuito que realiza la suma de dos dígitos codificados en BCD. Los dos dígitos BCD's se aplican al primer sumador binario de 4 bits. El sumador realizará la suma en binario y producirá un resultado comprendido entre 0 y 19 (= 9 + 9 + 1 arrastre a la entrada del sumador). La salida de este sumador se representa por k, z_8, z_4, z_2 y z_1; k es el arrastre del sumador y los subíndices de z indican los pesos 8, 4, 2 y 1 que se asignan a los cuatro bits en el código BCD.

El circuito lógico que detecta cuándo es necesario corregir el resultado ($c = 1$) se obtiene de la forma siguiente:

a) Se requiere una corrección si hay un arrastre de salida en el primer sumador ($k = 1$). Esto sucede cuando la suma de los dos dígitos BCD es mayor que 15.

b) También se precisa corrección cuando la suma está comprendida entre 10 y 15 ($c_1 = 1$). Hay que detectar las 6 configuraciones que van desde $z_8\, z_4\, z_2\, z_1 = 1010$ a $z_8\, z_4\, z_2\, z_1 = 1111$.

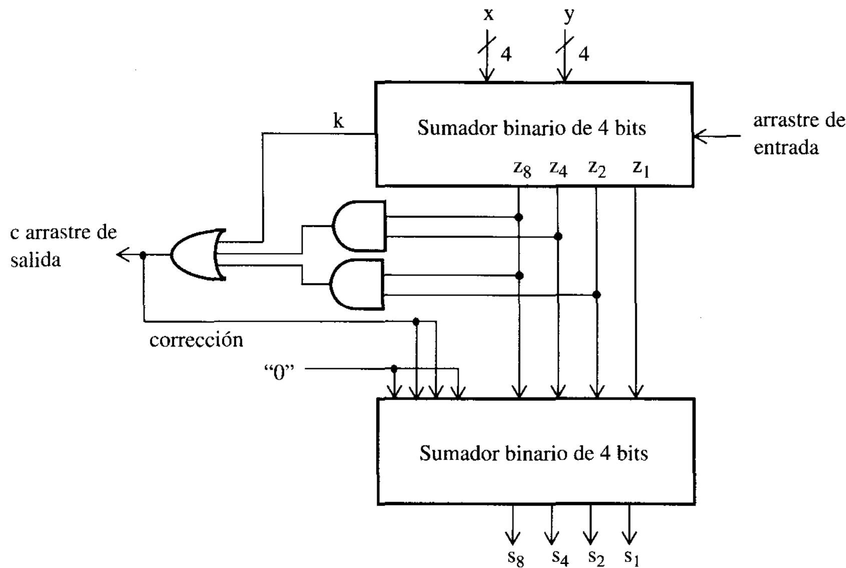


Figura 4.27: Sumador de dos dígitos en código BCD

El mapa de Karnaugh de la Figura 4.28 permite obtener la condición lógica (c_1) correspondiente.

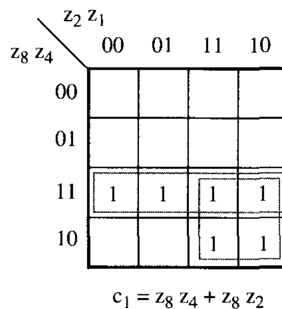


Figura 4.28: Corrección en la suma BCD

Teniendo en cuenta a) y b) se puede expresar la condición de corrección por la siguiente función booleana:

$$c = k + c_1 = k + z_8 z_4 + z_8 z_2$$

Cuando $c = 1$, es necesario añadir $6_{10} = 0110_2$ a la suma producida por el primer sumador y proporcionar un arrastre de salida para la siguiente etapa de dígitos BCD. El segundo sumador binario de 4 bits corrige el resultado añadiendo 6 cuando $c = 1$. Debe observarse que cuando $c = 0$, el segundo sumador no corrige el resultado dado por el primero, ya que lo que se le suma es cero.

4.3.1 Organización de los sumadores en código BCD

La suma de dos números de n dígitos decimales codificados en BCD, se puede hacer de tres formas:

a) *Sumador paralelo*

Contiene n sumadores BCD como el descrito en la Figura 4.27. El arrastre de salida de cada sumador, se conecta al arrastre de entrada del siguiente sumador (ver Figura 4.29). La suma se genera en paralelo, independientemente de la longitud de los operandos.

Para conseguir retardos pequeños en la propagación de los arrastres, de la misma forma que se vió al estudiar los sumadores binarios, los sumadores BCD's pueden incluir la lógica necesaria que permita la anticipación del arrastre.

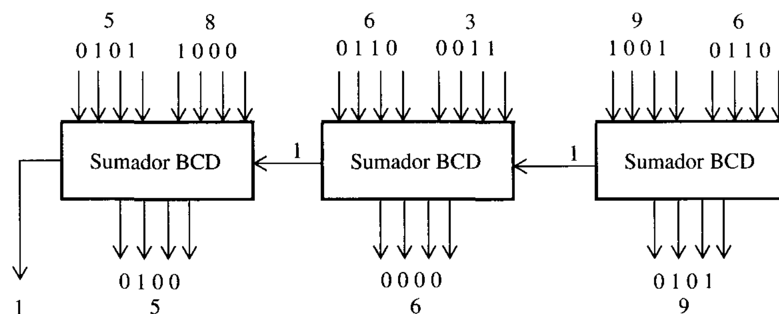


Figura 4.29: Sumador BCD paralelo

b) *Sumador dígito-serie, bit-paralelo*

En esta organización los dígitos se aplican de forma serie a un único sumador BCD, aunque los bits de cada dígito codificado se transfieren en paralelo (ver Figura 4.30).

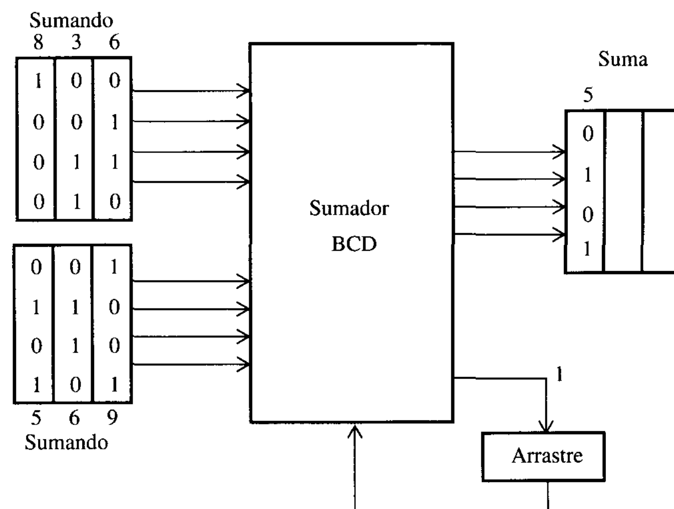


Figura 4.30: Sumador BCD dígito-serie, bit-paralelo

La suma se genera desplazando los números decimales, de uno en uno, a través del sumador BCD. Si son n los dígitos BCD's, se necesitarán n pasos para completar la operación de suma.

c) *Sumador dígito-serie, bit-serie*

Es la estructura de menor coste aunque también es la más lenta. En este caso, los bits se desplazan de uno en uno a través de un SBC (ver Figura 4.31). La suma binaria que se forma después de cuatro desplazamientos, hay que corregirla para obtener un dígito válido en código BCD.

Esta corrección, como ya se indicó, consiste en comprobar la suma binaria y si es mayor o igual a 10, sumar 6 y generar un arrastre para el siguiente par de dígitos.

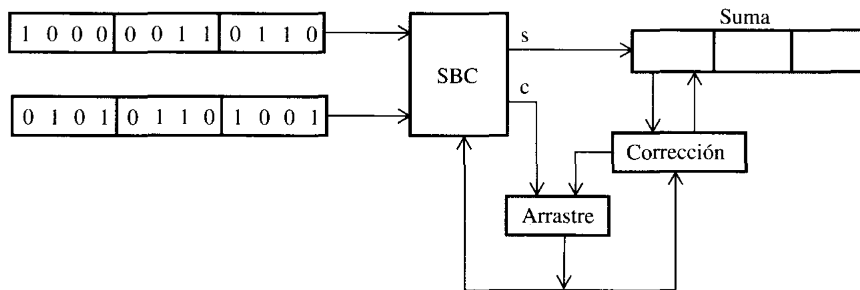


Figura 4.31: Sumador BCD dígito-serie, bit-serie

4.3.2 Restador en código BCD

La resta de dos números decimales codificados en BCD se puede realizar utilizando un circuito restador en código BCD, que aunque similar en su filosofía al sumador estudiado, es diferente. Una alternativa más económica consiste en formar el complemento a 9 ó 10 del sustraendo y sumarlo al minuendo (de manera similar al caso binario en el que se utilizaba el complemento a 1 ó a 2).

Como el código BCD no es un código autocomplementario, el complemento a 9 de un dígito no se obtiene complementando cada bit del código, se genera mediante un circuito que resta cada dígito BCD de 9. Esto se consigue complementando los bits en la representación codificada del dígito, a condición de que se incluya la corrección necesaria. Existen dos métodos para efectuar la corrección.

1) En el primer método, se suma el número 10 (1010) a cada dígito BCD complementado y no se considera el arrastre después de cada suma. En efecto, complementar cada bit de un número binario N de 4 bits es idéntico a restarlo del número 15 (1111). Si a continuación se le suma 10 se obtiene: $15 - N + 10 = 9 - N + 16$. Pero 16 representa el arrastre de salida que no se considera.

2) En el segundo método, se suma el número 6 (0110) a cada dígito BCD antes de complementarlo, es decir: $15 - (N + 6) = 9 - N$, que es el complemento a 9 de N.

También se puede obtener el complemento a 9 de un dígito BCD mediante un circuito combinacional que dispone de una señal de control M. La entrada a este circuito y_8, y_4, y_2 e y_1, es el dígito del sustraendo (o del segundo sumando) y su salida Y_8, Y_4, Y_2 e Y_1, es igual a la entrada cuando $M = 0$, e igual al complemento a 9 de la entrada cuando $M = 1$. Las ecuaciones lógicas del circuito combinacional "complementador a 9" son las siguientes (ver problema 15):

$$
\begin{aligned}
Y_1 &= y_1\overline{M} + \overline{y}_1 M \\
Y_2 &= y_2 \\
Y_4 &= y_4\overline{M} + (\overline{y}_4 y_2 + y_4\overline{y}_2) M \\
Y_8 &= y_8\overline{M} + \overline{y}_8\,\overline{y}_4\,\overline{y}_2 M
\end{aligned}
\tag{4.10}
$$

De estas ecuaciones se observa que $Y = y$ cuando $M = 0$. Si $M = 1$, las salidas de Y generan el complemento a 9 de y. Cuando se conecta este circuito a un sumador BCD, el resultado es una unidad aritmética decimal que suma o resta dos dígitos BCD (ver Figura 4.32). La operación de suma/resta está controlada por la señal M:

$$
\begin{aligned}
&\text{si} \quad M = 0 \quad \Rightarrow \quad s = x + y \\
&\text{si} \quad M = 1 \quad \Rightarrow \quad s = x + C9(y)
\end{aligned}
$$

donde $C9(y)$ representa el complemento a 9 de y. Para números con n dígitos decimales se necesitarán n módulos sumadores/restadores. El arrastre de salida c_{i+1} de una etapa, debe conectarse al arrastre de entrada c_i de la etapa de orden superior.

La mejor forma de realizar la resta de dos números decimales es la siguiente:

a) Hacer $M = 1$

b) Hacer $c_{-1} = 1$ (c_{-1} es el arrastre de entrada de la primera etapa)

si se cumplen estas condiciones, la salida del circuito será:

$$s = x + C10(y) \quad \Leftrightarrow \quad s = x - y$$

si no se considera el arrastre de salida de la última etapa.

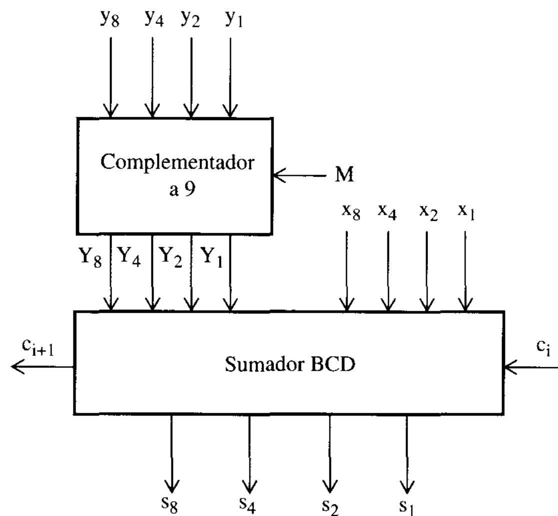


Figura 4.32: Sumador-restador de 2 dígitos en BCD

Sección 4-4

Multiplicadores binarios

La multiplicación es una operación compleja en comparación con la suma y la resta. Hay una gran variedad de algoritmos que permiten realizarla. El objetivo de esta sección es presentar los métodos más usuales, indicando sus ventajas e inconvenientes. Se comienza con la forma más simple de multiplicar dos números binarios sin signo (no negativos), es el procedimiento de multiplicación clásico de "lápiz y papel". Este método es muy ineficaz desde el punto de vista de velocidad de cálculo y coste del multiplicador, por lo que se propone una variante del mismo que evita alguno de sus problemas. Finalmente, para el caso de números con signo en complemento a 2 se presenta el algoritmo de Booth, que es el más clásico y el que ha dado lugar a un mayor número de generalizaciones.

4.4.1 Multiplicación de "lápiz y papel" de números sin signo

La Figura 4.33 muestra un ejemplo de multiplicación de dos números enteros binarios sin signo de 4 bits, tal como se efectúa utilizando el algoritmo clásico de la escuela de "lápiz y papel" (también se le conoce como algoritmo de "suma y desplazamiento"). *M* representa el multiplicando y *m* el multiplicador.

```
      M =    1 0 0 1    Multiplicando (9)
      m =    1 0 1 1    Multiplicador (11)
            ---------
             1 0 0 1
           1 0 0 1      Productos parciales
         0 0 0 0
       1 0 0 1
      ---------------
       1 1 0 0 0 1 1    Producto (99)
```

Figura 4.33: Multiplicación de dos números binarios sin signo

De este ejemplo se pueden hacer algunas consideraciones importantes:

1) La multiplicación genera 4 productos parciales p_i, uno por cada dígito del multiplicador. Estos productos parciales se suman después para obtener el producto final *P*.

2) Los productos parciales se definen fácilmente. Cuando el bit del multiplicador es 0, el producto parcial es 0. Cuando el multiplicador es 1, el producto parcial es el multiplicando:

$$p_i = M \times m_i$$

3) Cada producto parcial sucesivo se desplaza una posición a la izquierda en relación con el producto parcial precedente. El producto final *P* se obtiene sumando los productos parciales.

$$P = \sum_{i=0}^{n-1} p_i = M\sum_{i=0}^{n-1} m_i 2^i = M \times m$$

4) La multiplicación de dos números enteros binarios de n bits tiene una longitud de $2n$ bits.

En una versión estrictamente combinacional del algoritmo de lápiz y papel, los productos parciales se forman simultáneamente y se suman de forma concurrente. Cada bit del producto parcial $p_{ij} = m_i M_j$ se genera en una puerta AND que tiene como entradas un bit del multiplicador (m_i) y un bit del multiplicando (M_j) (ver Figura 4.34).

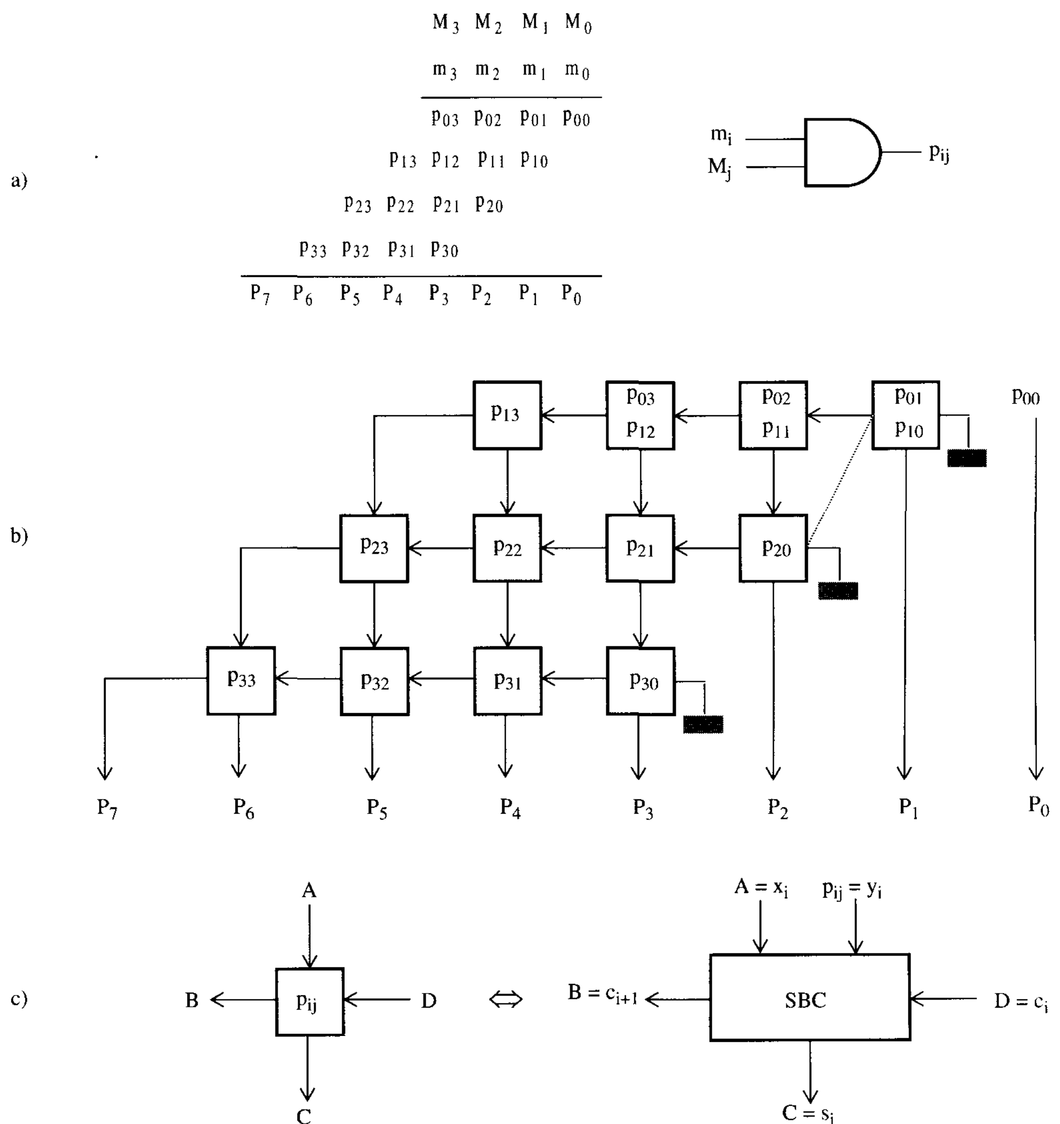


Figura 4.34: Multiplicador combinacional de 4 × 4 bits compuesto por 12 SBC's

La generación de los n productos parciales (p_i) y su suma para formar el producto final (P), se realiza en una red de $n(n-1)$ SBC's (12 para $n = 4$), tal como se muestra en la Figura 4.34. La estructura del multiplicador corresponde al algoritmo de suma y desplazamiento que se acaba de ver, porque cuando un bit del multiplicador vale 0 resulta un producto parcial de cero, y cuando vale 1 hace que el producto parcial sea el valor del multiplicando. El desplazamiento viene dado por la forma de interconectar los SBC's.

La velocidad de este multiplicador combinacional está determinada por el retardo en la propagación de los arrastres, es decir, el retardo desde p_{01} hasta p_{13} y desde p_{13} hasta P_7. Este camino está constituido por $2n$-2 = 6 etapas, cada una con el retardo propio de un SBC (se supone un retardo de 4 para cada SBC), lo que da un retardo de 24 al que hay que sumarle un retardo adicional correspondiente a la matriz AND de los bits p_{ij}. Se obtiene así un retardo total en la propagación de los arrastres de 25.

4.4.2 Mejoras en el algoritmo de "lápiz y papel"

Si se analiza críticamente el algoritmo anterior se observa que hay algunas cosas que se pueden modificar para conseguir una multiplicación más eficaz. En concreto se consideran las siguientes:

1) Se puede realizar la suma de los productos parciales tan pronto como se producen, en lugar de tener que esperar hasta el final. Esto elimina la necesidad de almacenar todos los productos parciales y en consecuencia se necesitan menos registros intermedios.

2) Se puede ahorrar tiempo en la generación de los productos parciales. Para cada 1 en el multiplicador se requieren las operaciones de suma y desplazamiento; pero para cada 0 sólo es necesario el desplazamiento.

La Figura 4.35 muestra el esquema de un multiplicador que tiene en cuenta estas modificaciones. El multiplicador y el multiplicando se cargan en dos registros de n bits (m y M). Se necesita un tercer registro A de n bits (registro acumulador) que inicialmente se carga a 0. Hay un registro C de 1 bit (registro de rebose) que se inicializa a 0 y que almacenará un posible bit de arrastre c_a que resulta de la suma. La operación del multiplicador se realiza de la forma siguiente:

a) La unidad de control, lee los bits del mutiplicador uno a uno.

b) Si m_0 es 1, el multiplicando M se suma con el contenido del registro A y el resultado de la suma se almacena en A.

c) Todos los bits de los registros C, A y m se desplazan simultáneamente 1 bit a la derecha, de la forma siguiente: $0 \rightarrow C$, $C \rightarrow A_{n-1}$, $A_{n-1} \rightarrow A_{n-2}$, ..., $A_0 \rightarrow m_{n-1}$, ..., $m_1 \rightarrow m_0$ y m_0 se pierde.

d) Si m_0 es 0, no se realiza la suma, sólo el desplazamiento.

e) Estas operaciones se repiten para cada bit del multiplicador m original. El producto resultante P, de $2n$ bits, estará contenido al final de este proceso en los registros A y m.

Esta forma de realizar el producto se puede expresar matemáticamente como sigue:

$$p_0 = 0$$

$$p_{i+1} = 2^{-1} (p_i + 2^n M \times m_i) \; i = 0, \ldots\ldots, n-1 \qquad (4.11)$$

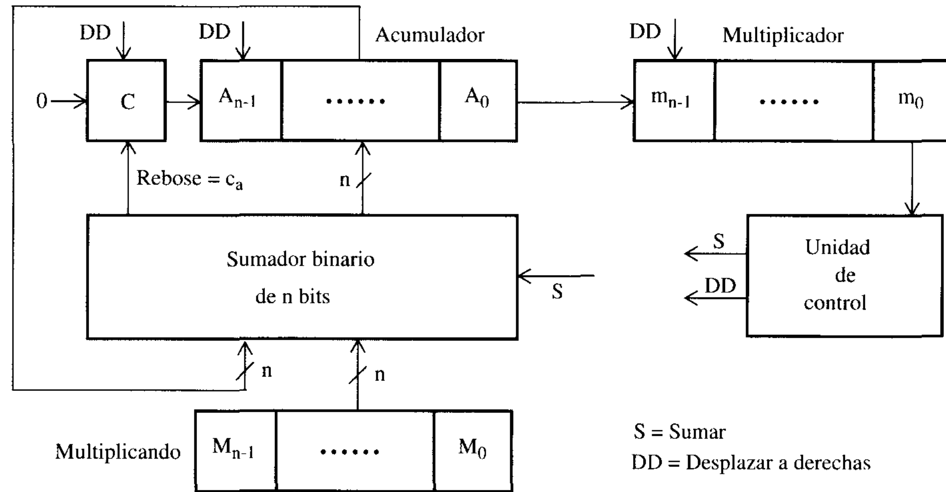


Figura 4.35: Esquema de un multiplicador que mejora el algoritmo de "lapiz y papel"

Se observa que cada paso consiste en la multiplicación de M por un bit m_i del multiplicador para formar Mm_i, seguido por una suma de dos operandos y por el desplazamiento de un bit a la derecha (2^{-1}). El factor 2^n que multiplica a $M \times m_i$ indica que el multiplicando M se tiene que alinear con la mitad más significativa del producto parcial que se encuentra almacenada en el registro A.

Este tipo de recurrencia, que utiliza desplazamientos a derecha en lugar de los desplazamientos a izquierda del algoritmo de "lápiz y papel", permite que la multiplicación pueda progresar desde el bit menos significativo del multiplicador mientras que el multiplicando retiene la misma posición en relación con el sumador. Esto se ilustra en la Figura 4.36 con el mismo ejemplo numérico que el empleado en el algoritmo de "lápiz y papel".

C	A	m			
0	0000	1011	Valores iniciales		
0	1001	1011	S	1er ciclo	
0	0100	1101	DD		
0	1101	1101	S	2° ciclo	
0	0110	1110	DD		
0	0011	0111	DD	3er ciclo	
0	1100	0111	S	4° ciclo	(Producto en A, m)
0	0110	0011	DD		

Figura 4.36: Ejemplo numérico del algoritmo de "lapiz y papel mejorado" (M contiene 1001)

En la Figura 4.37 se da un diagrama de flujo de la multiplicación de números binarios sin signo utilizando este algoritmo. Se ha utilizado la notación || para definir un registro compuesto constituido a partir de otros registros. Así por ejemplo $C||A||m$ representa un único registro obtenido al combinar en ese orden

los registros *C*, *A* y *m* (el bit más significativo está en *C* y el menos significativo a la derecha de *m*). El símbolo >> especifica un desplazamiento del dato hacia la derecha, indicándose a la derecha del símbolo el número de bits de desplazamiento. Al sumar el contenido de los registros *A* y *M* se puede producir un arrastre en la operación que se representa por c_a. *P* es un contador que controla el número de iteraciones del bucle.

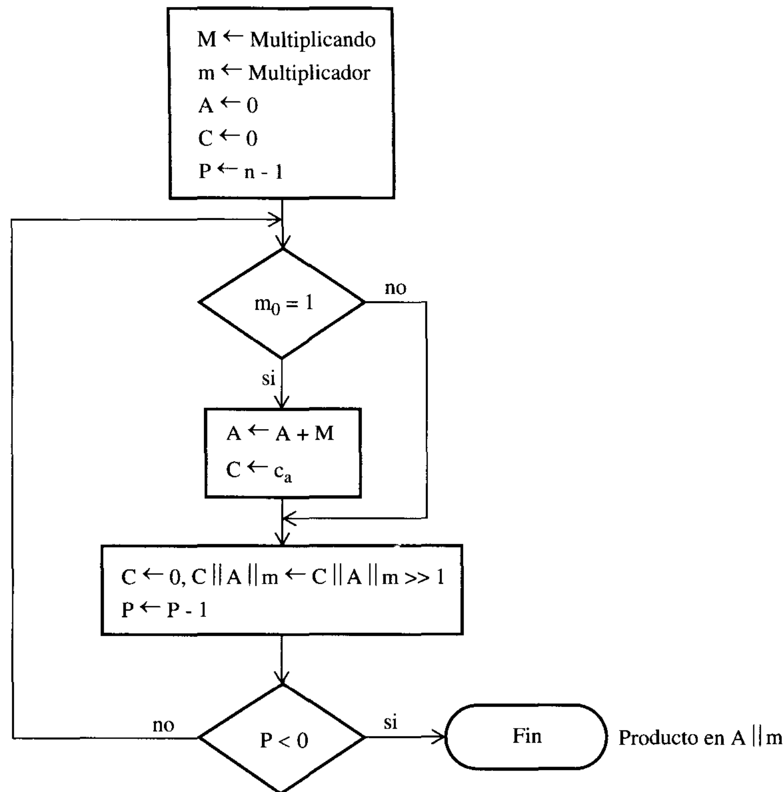


Figura 4.37: Diagrama de flujo de la multiplicación de números binarios sin signo

4.4.3 Multiplicación en complemento a 2: Algoritmo de Booth

Se ha visto ya que la suma y la resta se pueden realizar, en el caso de numeros con signo representados en complemento a 2, de la misma forma que si fuesen números sin signo. Considérese la siguiente suma:

$$\begin{array}{r} 1\,0\,1\,1 \\ +\,0\,0\,1\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

si estos dos números se consideran sin signo, se está sumando $11_{10} = 1011_2$ con $2_{10} = 0010_2$ para obtener $13_{10} = 1101_2$. Si los números están representados en complemento a 2, lo que se suma es $-5_{10} = 1011_2$ y $2_{10} = 0010_2$ y da como resultado $-3_{10} = 1101_2$. Desafortunadamente, este esquema tan sencillo no va bien con la multiplicación. Para verlo considérese otra vez la Figura 4.33 en la que se multiplica $9_{10} = 1001_2$ por

$11_{10} = 1011_2$ y da como resultado $99_{10} = 1100011_2$. Si se interpretan estos números en complemento a 2, se tiene $-7_{10} \times -5_{10} = -29_{10}$ ($1001_2 \times 1011_2 = 1100011_2$). Este ejemplo demuestra que la multiplicación realizada directamente produce resultados erróneos si el multiplicando y el multiplicador son números negativos. De hecho, tampoco se obtiene el resultado correcto si sólo uno de los dos es negativo. Para explicar esto conviene recordar como se puede expresar un número binario como una suma de potencias de 2. Así

$$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 2^3 + 2^1 + 2^0$$

Más aún, la multiplicación de un número binario por 2^n se efectúa desplazando ese número n bits a la izquierda. Teniendo esto en cuenta, en la Figura 4.38 se reconstruye la Figura 4.33 para hacer explícita la generación de los productos parciales en la multiplicación. La única diferencia en la Figura 4.38 es que se pone de manifiesto, que los productos parciales deben considerarse como números de $2n$ bits que se generan a partir de un multiplicando de n bits.

M = 1001	
m = 1011	
00001001	$1001 \times 1 \times 2^0$
00010010	$1001 \times 1 \times 2^1$
00000000	$1001 \times 0 \times 2^2$
01001000	$1001 \times 1 \times 2^3$
01100011	

Figura 4.38: La multiplicación de 2 enteros de 4 bits sin signo da un resultado de 8 bits

Así, si el multiplicando de 4 bits 1001 (considerado como un entero sin signo) se almacena en una palabra de 8 bits como 00001001, cada producto parcial (salvo el que corresponde a 2^0) consiste en este número desplazado a la izquierda con las posiciones no ocupadas a su derecha rellenas con ceros (por ejemplo, un desplazamiento de dos lugares a la izquierda da 00100100).

Se puede ver ahora que la multiplicación realizada de forma directa no funciona si el multiplicando es negativo. El problema es, que cada contribución del multiplicando negativo como un producto parcial debe ser un número negativo en un campo de $2n$ bits; los bits de signo de los productos parciales tienen que estar alineados. Esto puede observarse en la Figura 4.39a que muestra la multiplicación de 1001 por 0101. Si estos números se tratan como enteros sin signo, la multiplicación de $9 \times 5 = 45$ se obtiene correctamente. Sin embargo, si 1001 se interpreta como el número -7 en complemento a 2, entonces cada producto parcial debe ser un número negativo en complemento a 2 tal como se muestra en la Figura 4.39b. En este caso el resultado correcto se consigue rellenando cada producto parcial a su izquierda con unos.

1001	(9)	1001	(−7)
× 0101	(5)	× 0101	(5)
00001001	$(1001) \times 2^0$	11111001	$(-7) \times 2^0 = (-7)$
00100100	$(1001) \times 2^2$	11100100	$(-7) \times 2^2 = (-28)$
00101101	(45)	11011101	(−35)
(a) Enteros sin signo		(b) Enteros en complemento a 2	

Figura 4.39: Comparación de la multiplicación de enteros sin signo y en complemento a 2

Si el multiplicador es negativo el resultado de la multiplicación tampoco es correcto. La razón es que los bits del multiplicador no se corresponden ya con los desplazamientos o multiplicaciones que hay que realizar. Por ejemplo:

$$-5 = 1011 = -(0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -2^2 - 2^0$$

por lo que el multiplicador no se puede utilizar directamente de la forma que se ha descrito. Para superar este problema se tienen varias alternativas.

La más directa sería convertir a números positivos tanto el multiplicando como el multiplicador, realizar la multiplicación y formar el complemento a 2 del resultado si y sólo si difieren los signos de los dos números originales. A pesar de la sencillez conceptual de este método, es mejor utilizar otras técnicas que no necesitan el paso final de complementar a 2 el resultado. Uno de los métodos más utilizado es el *algoritmo de Booth*, que también tiene la ventaja de acelerar la multiplicación como se verá a continuación. En la Figura 4.40 se da un diagrama de flujo del algoritmo de Booth. Merecen destacarse las siguientes consideraciones:

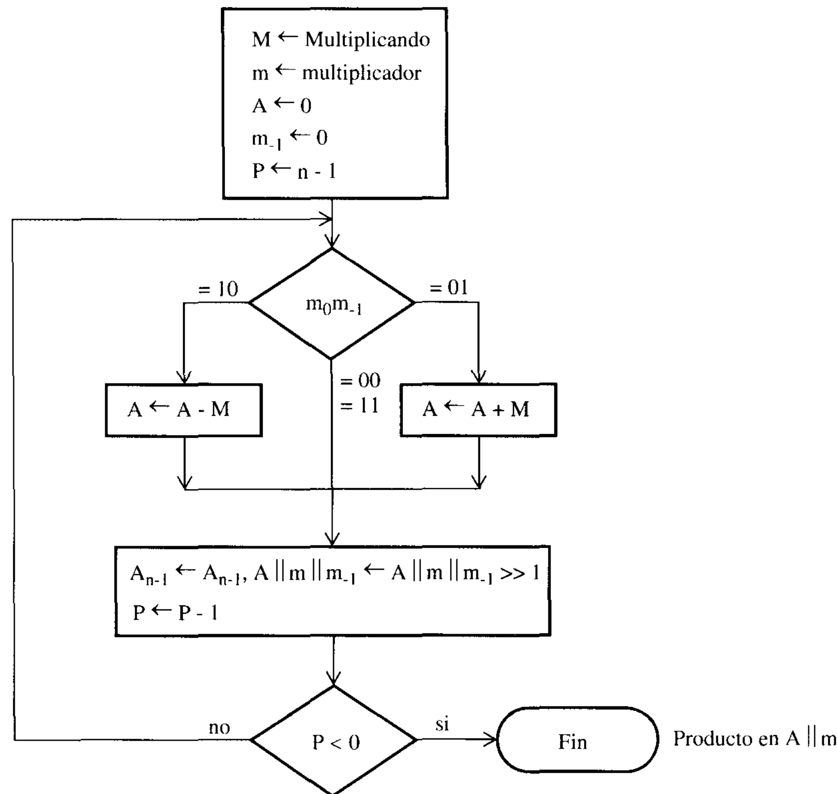


Figura 4.40: Diagrama de flujo del algoritmo de Booth

a) El multiplicando y el multiplicador se almacenan en los registros M y m respectivamente.

b) Hay un registro de 1 bit que se coloca a la derecha del bit menos significativo (m_0) del registro m. Se representan por m_{-1} y su utilidad se explica posteriormente.

c) El resultado de la multiplicación se obtiene en los registros A y m. Los registros A, m y m_{-1} se inicializan a 0.

d) La unidad de control del multiplicador examina uno a uno los bits del registro m. Sin embargo, en el algoritmo de Booth cuando se examina cada bit, se comprueba también el bit que está a su derecha.

e) Si los dos bits son iguales (1-1 ó 0-0), todos los bits de los registros A, m y m_{-1} se desplazan un bit a la derecha.

f) Si los dos bits difieren, el multiplicando se suma o se resta del registro A, según que los dos bits sean 0-1 ó 1-0 respectivamente.

g) El desplazamiento se produce siempre después de la suma o la resta. En cualquiera de los casos, el desplazamiento a derecha es tal que el bit más significativo de A, es decir A_{n-1}, permanece en A_{n-1} y se desplaza a A_{n-2}. Esto se necesita para preservar el signo del número en A y en m. Este desplazamiento se conoce como *desplazamiento aritmético*, ya que conserva el bit de signo (las operaciones de desplazamiento se verán en la sección 4-9).

En la Figura 4.41 se muestra la secuencia de pasos del algoritmo de Booth para multiplicar 6 por 3.

A	m	m_{-1}			
0000	0011	0	Valores iniciales		
1010	0011	0	$A \leftarrow A - M$		
1101	0001	1	Desplazar	1er ciclo	
1110	1000	1	Desplazar	2º ciclo	
0100	1000	1	$A \leftarrow A + M$		
0010	0100	0	Desplazar	3er ciclo	
0001	0010	0	Desplazar	4º ciclo	(Producto en A ‖ m)

Figura 4.41: Ejemplo del algoritmo de Booth (M contiene 0110)

En la Figura 4.42a se representa la misma operación de forma más compacta. El resto de la Figura 4.42 muestra otros ejemplos del algoritmo que, como se puede ver, funciona correctamente con cualquier combinación de números positivos y negativos. Es de destacar la eficacia del algoritmo de Booth, ya que ahorra las sumas cuando hay bloques de 1's ó 0's, con un promedio de una suma o resta por bloque.

A continuación se justifica por qué funciona correctamente el algoritmo de Booth. En primer lugar se considera el caso en que el multiplicador m es positivo. En particular, se supone que m consiste en una secuencia de 1's rodeada por 0's, por ejemplo 00111100. Como se sabe, la multiplicación se puede realizar sumando copias del multiplicando M desplazadas adecuadamente:

$$M \times (00111100) = M \times (2^5 + 2^4 + 2^3 + 2^2)$$

$$M \times (32 + 16 + 8 + 4) = M \times 60$$

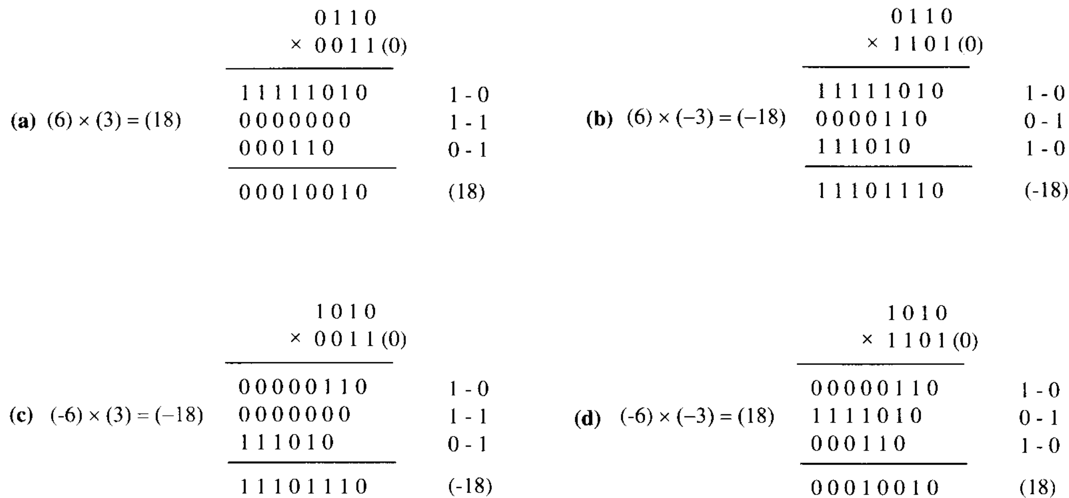


Figura 4.42: Diferentes ejemplos del algoritmo de Booth

el número de estas operaciones se puede reducir a dos si se observa que:

$$2^j + 2^{j-1} + \ldots + 2^{j-k} = 2^{j+1} - 2^{j-k} \qquad (4.12)$$

teniendo esto en cuenta, el resultado anterior se puede expresar como:

$$M \times (00111100) = M \times (2^6 - 2^2) = M \times (64 - 4) = M \times 60$$

En el ejemplo se ve que el producto se puede generar mediante una suma y una resta del multiplicando. El esquema se generaliza sin ninguna dificultad a cualquier número de secuencias de 1's en el multiplicador, incluyendo el caso de un único 1 que se trata también como una secuencia. Así

$$M \times (01110010) = M \times (2^7 + 2^6 + 2^5 + 2^1) = M \times (2^8 - 2^5 + 2^2 - 2^1)$$

El algoritmo de Booth realiza precisamente este esquema, efectuando una resta cuando se encuentra el primer 1 de una secuencia (1-0) y una suma al detectar el final de la secuencia (0-1). Para demostrar que este mismo principio funciona cuando el multiplicador es negativo, se necesita observar lo siguiente: sea *X* un número negativo representado en complemento a 2, es decir:

$$\text{Representación de } X = \{1\ x_{n-2}\ x_{n-3} \ldots x_1\ x_0\}$$

el valor de *X* se puede expresar como sigue:

$$X = -2^{n-1} + x_{n-2} \times 2^{n-2} + \ldots + x_1 \times 2^1 + x_0 \times 2^0 \qquad (4.13)$$

Como *X* es negativo, su bit más significativo es 1. Sin pérdida de generalidad se supone que el 0 más a la izquierda en *X* está en la posición *k*-ésima. Así *X* es de la forma:

$$\text{Representación de } X = \{1\ 1 \ldots 1\ 0\ x_{k-1}\ x_{k-2} \ldots x_1\ x_0\} \qquad (4.14)$$

su valor será:

$$X = -2^{n-1} + 2^{n-2} + \ldots + 2^{k+1} + x_{k-1} \times 2^{k-1} + \ldots + x_0 \times 2^0 \qquad (4.15)$$

teniendo en cuenta la ecuación (4.12), se puede ver que

$$2^{n-2} + 2^{n-3} + ... + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

reordenando los términos de esta última expresión:

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^{k+1} = - 2^{k+1} \quad (4.16)$$

sustituyendo la ecuación (4.16) en la ecuación (4.15), se obtiene:

$$X = - 2^{k+1} + x_{k-1} \times 2^{k-1} + ...+ x_0 \times 2^0 \quad (4.17)$$

Si se tiene en cuenta el algoritmo de Booth y la representación de X (ecuación (4.14)), es evidente que todos los bits de X hasta el 0 que está más a la izquierda se manejan adecuadamente, puesto que producen todos los términos en la ecuación (4.17) salvo (-2^{k+1}) y están así en la forma apropiada. Corresponde pues al primer caso visto de multiplicador positivo, si se considera el número hasta el 0 más a la izquierda como un número positivo. Cuando el algoritmo de Booth examina los bits que están más a la izquierda del 0 de la posición k-ésima, y encuentra el siguiente 1 (2^{k+1}), ocurre una transición 1-0 y tiene lugar una resta (-2^{k+1}). Éste es precisamente el término que quedaba en la ecuación (4.17).

4.4.4 Ejemplo: Algoritmo de Booth

Se considera la multiplicación de M×(-6). En la representación complemento a 2 y utilizando una longitud de palabra de 8 bits, $-6_{10} = 11111010_2$. De la ecuación (4.13) se sabe que:

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

por lo que

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

utilizando la ecuación (4.17) hasta el primer 0 más a la izquierda, la expresión anterior se puede expresar como:

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

Finalmente, siguiendo la misma línea de razonamiento con el segundo 0 se obtiene:

$$M \times (11111010) = M \times (-2^3 + 2^2 - 2^1)$$

Se puede ver que el algoritmo de Booth se adecúa a este esquema. Realiza una resta cuando se encuentra el primer 1 (transición 1-0), una suma cuando detecta una transición 0-1 y finalmente otra resta cuando ve el primer 1 de la siguiente secuencia de 1's. De esta forma el algoritmo de Booth efectúa menos sumas y restas que si se hubiese utilizado un algoritmo directo.

Sección 4-5

Multiplicadores de alta velocidad

En esta sección se analizan dos métodos básicos para acelerar la multiplicación. El primero genera más rápidamente la suma de los productos parciales utilizando sumadores del tipo de arrastre almacenado y una estructura de interconexión denominada "árbol de Wallace". El segundo reduce el número de productos parciales empleando un algoritmo de Booth modificado.

4.5.1 Suma rápida de los productos parciales

El elevado retardo en el multiplicador combinacional de la Figura 4.34 se debe a la propagación del arrastre a lo largo de todas las etapas. Este retardo se puede reducir casi a la mitad haciendo un cambio sencillo en la estructura de interconexión de los arrastres de los SBC's (que se indica en la Figura 4.34 por la línea discontinua).

En lugar de esperar a que el arrastre se propague a lo largo de todos los SBC's que se encuentran en su misma fila a su izquierda, se suma a la entrada de arrastre del SBC que está localizado en la fila de debajo y a su izquierda. De esta forma se aplaza la suma de los arrastres. El principio de posponer la suma de los arrastres se puede extender a todas las etapas del sumador excepto a la última. Los arrastres que se obtienen forman un operando de n bits, que se añade a la suma parcial de la última etapa en un sumador rápido dotado con aceleración de arrastre.

Esta técnica también permite utilizar una tercera entrada en cada uno de los SBC's de la etapa superior (primera fila), de manera que los tres primeros productos parciales se pueden sumar en la primera etapa. Dicha modificación reduce el número de etapas del sumador de n-1 a n-2. El esquema de esta estructura para $n = 4$ se muestra en la Figura 4.43. Se observa que corresponde al tipo de sumadores con arrastre almacenado o sumadores CSA que se estudiaron anteriormente en el apartado 4.2.7.

Utilizando un árbol de Wallace se puede aumentar aún más la velocidad del multiplicador. El árbol de Wallace es una interconexión de SBC's con arrastre almacenado que reduce n productos parciales a dos operandos. A estos sumadores se les denomina por su acrónimo en inglés: CSA (Carry Save Adder- sumador de arrastre almacenado).

El principio del árbol de Wallace es utilizar un CSA para reducir tres bits de igual peso a dos bits, uno de suma y otro de arrastre. En una multiplicación de $n \times n$ se generan n productos parciales que se pueden considerar como columnas adyacentes de bits con igual peso.

La altura máxima de una columna es n bits que se divide en grupos de tres bits. Estos grupos se pueden reducir simultáneamente a dos bits, lo que resulta en una nueva columna con una altura de $(2/3)n$ bits. La nueva columna generada se particiona otra vez en grupos de tres bits, y el proceso se repite hasta que la altura de la columna final es de dos bits.

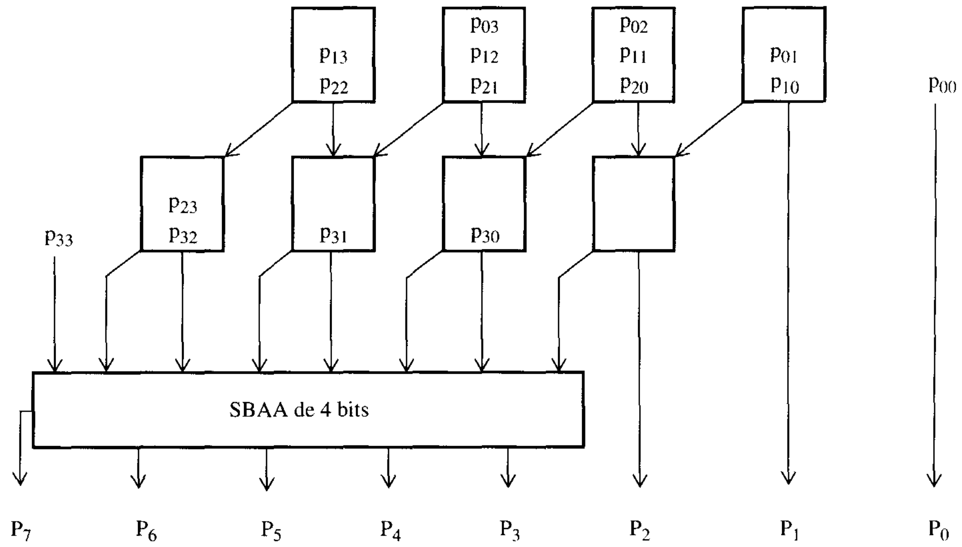


Figura 4.43: Multiplicador de 4 × 4 bits compuesto por CSA's y un SBAA de 4 bits

Para visualizar la interconexión del árbol de Wallace, en la Figura 4.44 se da una representación de puntos de un multiplicador de 8 × 8.

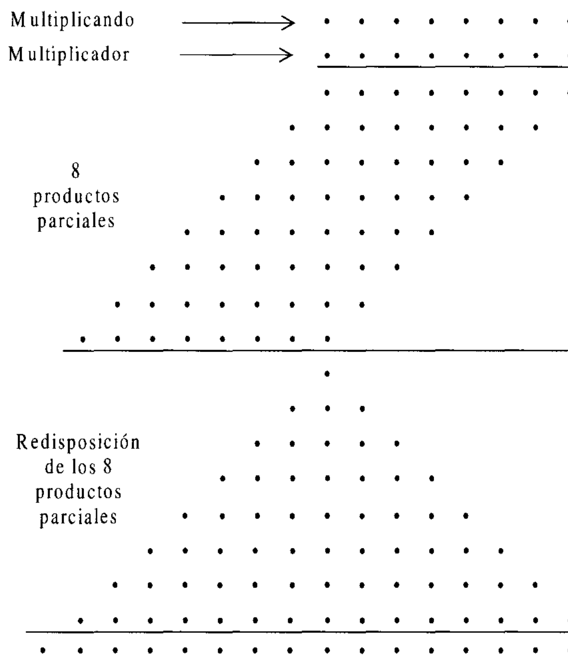


Figura 4.44: Representación de puntos de un multiplicador de 8 × 8 bits

En la Figura 4.45 se muestra una reducción del árbol de Wallace. Son necesarios 4 niveles de CSA's para reducir los 8 productos parciales a 2 operandos (2 niveles menos que si se hubiese utilizado el esquema de la Figura 4.43). El retardo de propagación del multiplicador de la Figura 4.45 es de 23 y se compone de los siguientes elementos:

1) Un retardo de 1 para generar los productos parciales.

2) Un retardo de 16 producido por 4 niveles de CSA's.

3) Un retardo de 6 producido en un sumador con anticipación de arrastres de 2 niveles.

En general para un árbol de Wallace el número de niveles de CSA's necesarios para reducir los *n* productos parciales a 2 viene dado por:

$$k = \left\lceil \frac{\log_2 n - 1}{\log_2 3 - 1} \right\rceil \qquad (4.18)$$

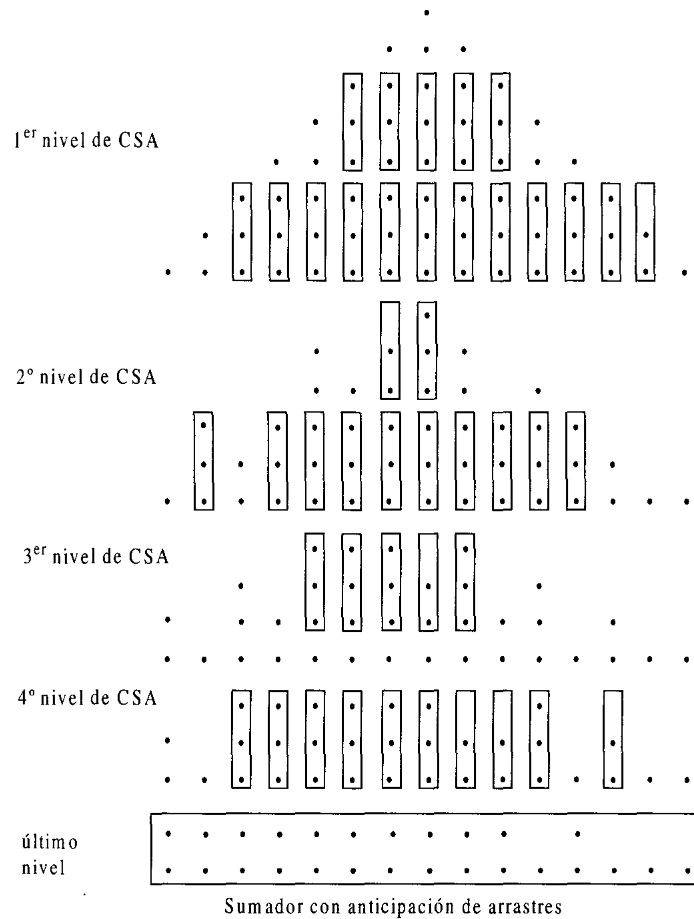


Figura 4.45: Reducción de la matriz de productos parciales de un multiplicador 8 × 8 utilizando sumadores del tipo CSA

donde $\lceil x \rceil$ es el menor entero mayor o igual a x. Así, en una multiplicacion de 32 × 32 bits, se necesitan 8 niveles de CSA's para reducir los 32 productos parciales a 2 operandos, en lugar de los 30 niveles de CSA's empleados cuando se utiliza el esquema de la Figura 4.45.

4.5.2 Algoritmo de Booth modificado

Otra técnica para acelerar la multiplicación consiste en una modificación del algoritmo de Booth, que permite reducir a la mitad el número de productos parciales (es decir, el número de niveles de CSA's) y duplicar por tanto la velocidad del multiplicador. El objetivo del algoritmo de Booth era examinar secuencias de bits en lugar de formar un producto parcial para cada bit. Cuando la secuencia es de 0's no hay ningún problema. Si la secuencia es de 1's, el algoritmo de Booth utiliza una propiedad especial de estas secuencias que viene dada por la expresión (4.12), de manera que el valor de dicha secuencia se obtiene restando el peso del 1 que está más a la derecha del módulo de la cadena; por ejemplo, la secuencia 1111 se calcula como $2^4 - 2^0 = 15$ y la secuencia 11100 se calcula como $2^5 - 2^2 = 28$. En el algoritmo de Booth modificado, se divide el multiplicador en subcadenas de 3 bits cada una y las 8 posibles combinaciones se calculan según la Tabla 4.12 .

2^1 m_{i+1}	2^0 m_i	2^{-1} m_{i-1}	Comentario
0	0	0	Sumar cero (no hay secuencia)
0	0	1	Sumar M (fin de secuencia)
0	1	0	Sumar M (secuencia)
0	1	1	Sumar 2M (fin de secuencia)
1	0	0	Restar 2M (comienzo de secuencia)
1	0	1	Restar M (-2M + M)
1	1	0	Restar M (comienzo de secuencia)
1	1	1	Restar cero (centro de la secuencia)

Tabla 4.12: Decodificación de 3 bits simultáneos del multiplicador

El algoritmo de Booth modificado requiere que el multiplicador se amplíe con un 0 a la derecha del bit menos significativo, y que se extienda el bit de signo a su izquierda, siempre que sea necesario, para formar el último grupo de 3 bits. Así, si el multiplicador tiene 8 bits, se divide en 5 grupos de 3 bits cada uno. Los grupos adyacentes tienen un bit en común (ver Figura 4.46).

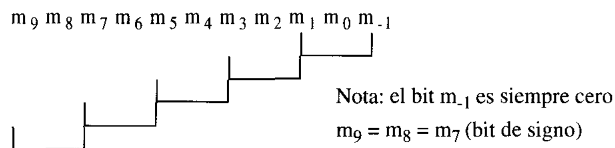


Figura 4.46: Partición del multiplicador ampliado en 5 grupos de 3 bits

El peso del bit del centro de cada grupo corresponde a la columna donde hay que empezar a colocar el producto parcial correspondiente. Se ve pues, que el algoritmo de Booth modificado incorpora un esquema de codificación del multiplicador que permite cada vez desplazamientos constantes de 2 bits, aunque se examinan 3 bits del multiplicador. En un multiplicador de 8 × 8 esto da 5 productos parciales, en lugar de los 8 que se necesitan sin codificación. Para justificar las entradas de la Tabla 4-10 hay que tener presente como opera el algoritmo de Booth :

- Se examinan los bits del multiplicador de derecha a izquierda, y se efectúa una resta del multiplicando cuando se encuentra el primer 1 de una secuencia (1-0) y una suma al detectar el final de la secuencia (0-1). Por ejemplo sea:

$$M \times (0011110) = M \times (2^4 + 2^3 + 2^2 + 2^1) = M \times (2^5 - 2^1)$$

En la Figura 4.47 se muestra, para este ejemplo, cómo se aplican las entradas de la Tabla 4.12 y se justifica que se obtiene el mismo resultado que aplicando el algoritmo de Booth.

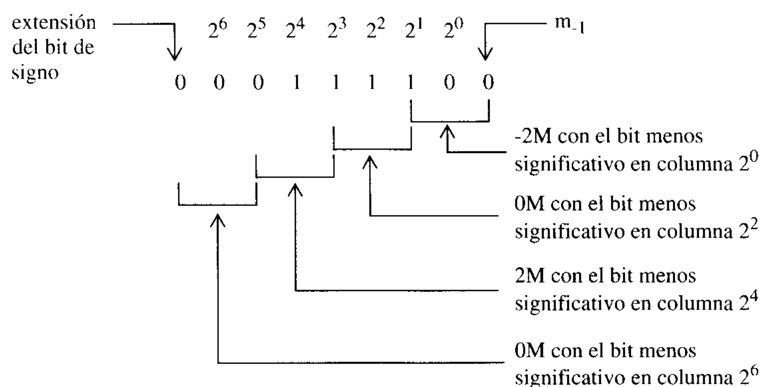


Figura 4.47: Ejemplo de aplicación del algoritmo de Booth modificado

A pesar de estas ventajas, el algoritmo de Booth modificado presenta un problema: se necesita una resta que se realiza, tal como se explicó en el apartado 4.1.5, sumando el C2 del número que se va a restar. Para formar el complemento a 2 de un número, en primer lugar se calcula el C1 (complementando cada bit del número) y a continuación se suma 1 al bit menos significativo. En la Tabla 4.12 se observa que cuando el valor del bit m_{i+1} es igual a 1 es necesario efectuar una resta, por lo que se puede utilizar este bit como señal de control M en el circuito sumador-restador de la Figura 4.12.

Un método rápido de obtener, de forma serie, el complemento a 2 de un número es el siguiente: se van examinando los bits del número a complementar de derecha a izquierda y se dejan inalterados hasta la aparición del primer 1, que queda inalterado, a partir de ese instante se complementan todos los bits. La Figura 4.48 muestra un ejemplo de aplicación de esta regla.

En complemento a 2 el bit de signo se debe extender hasta la longitud completa del resultado final. En la Figura 4.49 se muestran diferentes ejemplos numéricos del algoritmo de Booth modificado, para algunas combinaciones de números positivos y negativos en el multiplicando y en el multiplicador.

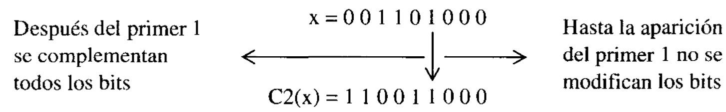


Figura 4.48: Forma rápida de obtener el complemento a 2 de un número

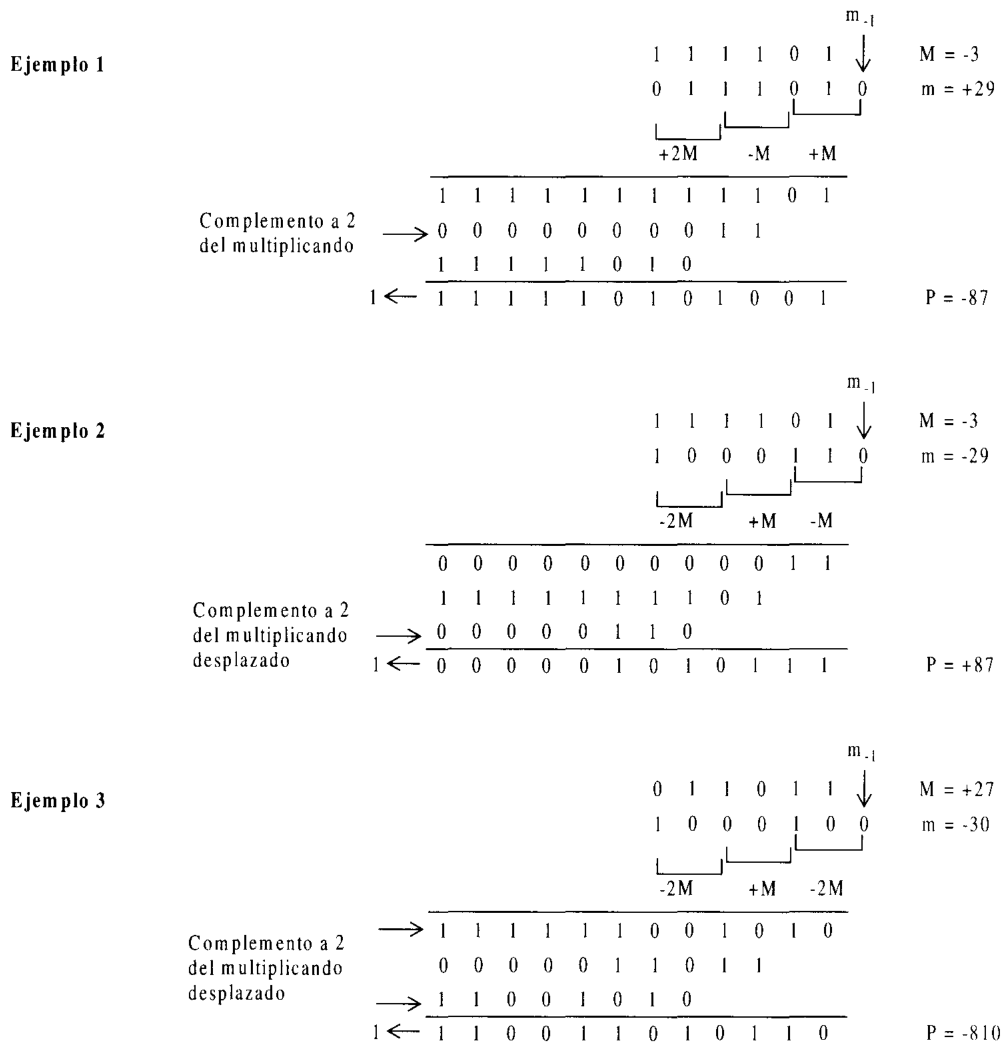


Figura 4.49: Ejemplos numéricos del algoritmo de Booth modificado

La codificación de 3 bits al tiempo, se puede extender a 4 bits cada vez, lo que reduciría el número de productos parciales de $n/2$ a $n/3$. Sin embargo, la codificación de 4 bits, requiere generar $3M$ que no es tan sencillo como generar $2M$.

Sección 4-6

Divisores binarios

Dados dos operandos, el dividendo D y el divisor d, el objetivo de los circuitos de división es calcular el cociente Q y el resto R tales que:

$$D = d \times Q + R$$

donde se requiere que el resto sea menor que el divisor, es decir $0 \leq R < d$. La expresión anterior indica una fuerte conexión entre las operaciones de multiplicar y dividir, donde el dividendo (D), el cociente (Q) y el divisor (d) se corresponden respectivamente con el producto (P), el multiplicador (m) y el multiplicando (M). Esta relación implica que los circuitos que se utilizan en ambas operaciones son análogos: en la multiplicación se suma repetidamente el multiplicando desplazado para generar el producto, y en la división el divisor desplazado se resta de forma repetida del dividendo para obtener el cociente. El método más simple de división binaria es el mismo que se realiza cuando se efectúa la división (decimal) con lápiz y papel.

4.6.1 Ejemplo: División de dos números enteros binarios sin signo

La Figura 4.50 muestra un ejemplo de división de dos números enteros binarios sin signo (39 y 6). El procedimiento que se utiliza para obtener el cociente es el siguiente:

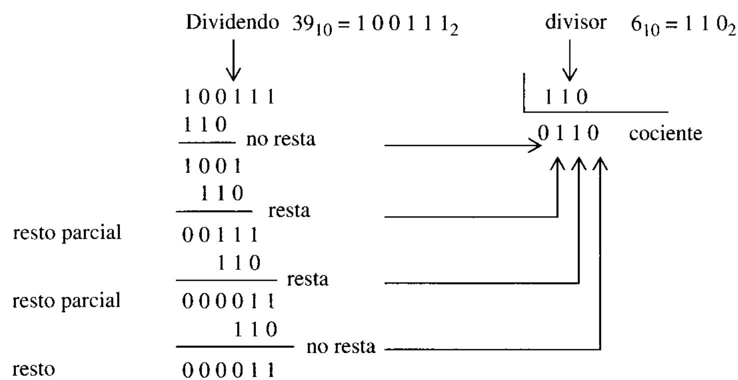


Figura 4.50: División por el método de lápiz y papel de 2 números binarios sin signo

1) Se examinan los bits del dividendo de izquierda a derecha, hasta que el conjunto de bits examinado representa un número mayor o igual que d; se dice entonces que d es capaz de *dividir* al número.

2) Hasta que ocurre este suceso se van colocando 0's en el cociente de izquierda a derecha.

3) Cuando el suceso tiene lugar, se coloca un 1 en el cociente y se resta el divisor del dividendo parcial. El resultado se conoce como *resto parcial.*

4) A partir de este punto la división sigue una estructura cíclica. En cada ciclo se añaden bits adicionales del dividendo al resto parcial, hasta que el resultado es mayor o igual que el divisor. Como antes, el divisor se resta de este número para producir un nuevo resto parcial.

5) El proceso continúa hasta que se acaban todos los bits del dividendo. ♦

La forma operativa del ejemplo presenta el inconveniente de que en el cálculo del resto parcial (R_{i+1}), el divisor que se resta de R_i sufre un desplazamiento previo hacia la derecha de un número creciente de bits:

$$R_{i+1} = R_i - Q_{n-i}\,2^{-i}d \qquad i = 0, 1, ..., n$$

resulta más adecuado ir desplazando un lugar a la izquierda el resto parcial R_i en cada paso de la división y operar con el divisor fijo. Esto produce un efecto idéntico a desplazar a la derecha el divisor, es decir:

$$R_{i+1} = 2R_i - Q_{n-i}\,d \qquad i = 0, 1, ..., n$$

En la Figura 4.51 se muestra la misma división del ejemplo anterior realizada con este procedimiento.

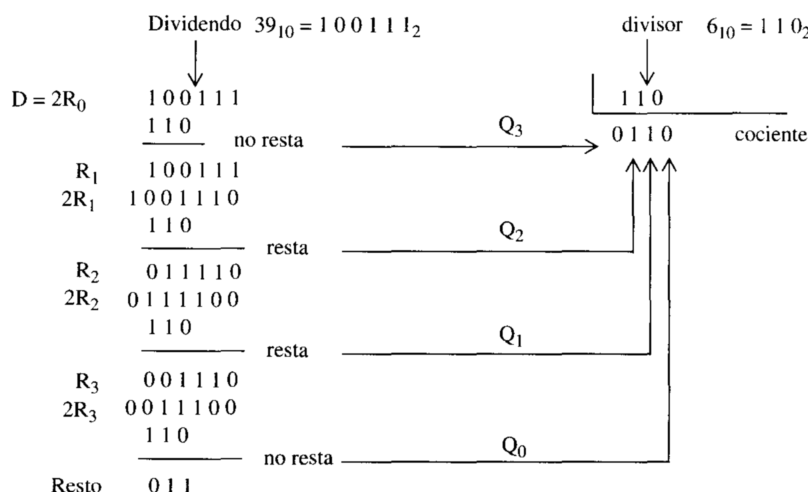


Figura 4.51: División por el método de lápiz y papel modificado

Conviene señalar que el último resto parcial R_n que se obtiene no da de forma inmediata el resto de la división, pues el resto verdadero R es: $R = R_n\,2^{-n}$. El problema principal en la división es calcular en cada paso el valor del dígito correspondiente del cociente. Cuando se utiliza aritmética binaria este problema se aligera bastante, ya que sólo hay dos dígitos posibles para el cociente (0 ó 1). Por lo tanto, es suficiente comparar la parte más significativa de $2R_i$ con el valor de d:

$$\text{si} \quad 2R_i < d \quad \Rightarrow \quad Q_{n-i} = 0$$

$$\text{si} \quad 2R_i \geq d \quad \Rightarrow \quad Q_{n-i} = 1$$

si n es grande, la forma más práctica de efectuar la comparación anterior es hacer la resta $2R_i - d$, utilizando un sumador-restador en lugar de utilizar un circuito combinacional de comparación de números de n bits. El valor de Q_{n-i} se determina según el resultado de la operación, es decir:

$$\text{si} \quad 2R_i - d \text{ es negativo} \Rightarrow Q_{n-i} = 0$$

$$\text{si} \quad 2R_i - d \text{ es positivo} \Rightarrow Q_{n-i} = 1$$

En la Figura 4.52 se muestra la estructura de registros que se necesita para realizar el método de división descrito. El registro de desplazamiento de n bits Q se utiliza inicialmente para almacenar el dividendo. El divisor se carga en el registro d y permanece fijo durante toda la operación. En cada paso de la división, el conjunto registro acumulador (A) y registro cociente (Q) se desplaza un bit a la izquierda. Esto transforma el resto parcial R_i en $2R_i$. Las posiciones que van quedando vacías en la parte derecha del registro Q se pueden utilizar para ir almacenando los bits que se van generando del cociente Q_{n-i}. Cuando finaliza la división, el registro A contiene el resto de la operación y el registro Q el cociente.

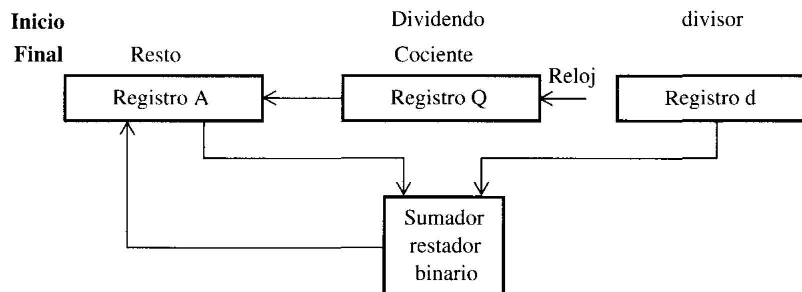


Figura 4.52: Estructura de registros para la operación de división

4.6.2 División por el método de restauración

Tal como se ha indicado, para evitar la utilización de circuitos comparadores de elevado coste, la comparación entre el dividendo y el divisor se realiza mediante una resta. En este caso el divisor se resta de la parte del dividendo que está bajo consideración.

Una respuesta positiva indica que el divisor es más pequeño, y se coloca un 1 en el cociente. Una respuesta negativa indica que el divisor es mayor, y que por lo tanto la resta no era necesaria y hay que volver a sumar el divisor al dividendo. La operación de suma *restaura* el valor original del dividendo, lo que da nombre al método. Este proceso se puede realizar sobre la estructura de registros de la Figura 4.52 mediante el algoritmo representado en la Figura 4.53.

4.6.3 Ejemplo: Algoritmo de restauración

En la Tabla 4.13 se muestra un ejemplo del algoritmo de restauración con los siguientes valores:

$$D = 11_{10} = 1011_2 \qquad d = 5_{10} = 0101_2$$

se necesitan registros de 4 bits y se obtiene como resultado:

$$Q = 2_{10} = 0010_2 \qquad \text{Resto} = 1_{10} = 0001_2$$ ♦

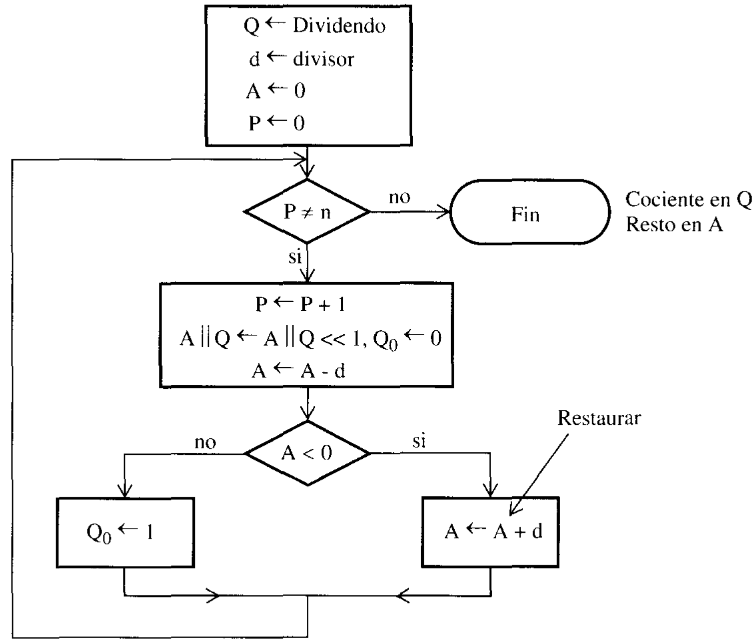


Figura 4.53: Diagrama de flujo del método de restauración

Paso	Acción	A	Q	d	P
0	Inicializar registros	0000	1011	0101	0
1	$P \leftarrow P + 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$ $A < 0 \Rightarrow A \leftarrow A + d$ (Restaurar)	 0001 1100 0001	 0110 0110 0110		1
2	$P \leftarrow P + 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$ $A < 0 \Rightarrow A \leftarrow A + d$ (Restaurar)	 0010 1101 0010	 1100 1100 1100		2
3	$P \leftarrow P + 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$ $A \geq 0 \Rightarrow Q_0 = 1$	 0101 0000 0000	 1000 1000 1001		3
4	$P \leftarrow P + 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$ $A < 0 \Rightarrow A \leftarrow A + d$ (Restaurar)	 0001 1100 0001 **Resto**	 0010 0010 0010 **Cociente**		4

Tabla 4.13: Ejemplo de aplicación del método de restauración

4.6.4 División por el método de no restauración

Puede evitarse el paso de restaurar del método anterior. Para ello conviene darse cuenta (ver Figura 4.53) que una restauración de la forma:

$$(R_i)_A \leftarrow (R_i)_A + d \tag{4.19}$$

va seguida en el paso siguiente por una resta

$$(R_{i+1})_A \leftarrow (2R_i)_A - d \tag{4.20}$$

donde el 2 corresponde al desplazamiento a izquierda de los registros A y Q.

$(R_i)_A$ representa que el cálculo de los sucesivos restos parciales se realiza sobre el registro A (ver Figura 4.52). Las ecuaciones (4.19) y (4.20) pueden combinarse en una sola para la obtención de R_{i+1}.

$$(R_{i+1})_A \leftarrow (2R_i)_A - d = 2((R_i)_A + d) - d \tag{4.21}$$

Esta idea es la base del método de no restauración, en el que si el bit $Q_{n-i} = 1$ el resto parcial R_{i+1} se evalúa utilizando (4.20), y si $Q_{n-i} = 0$ mediante (4.21). En la Figura 4.54 se da el algoritmo del método de no restauración.

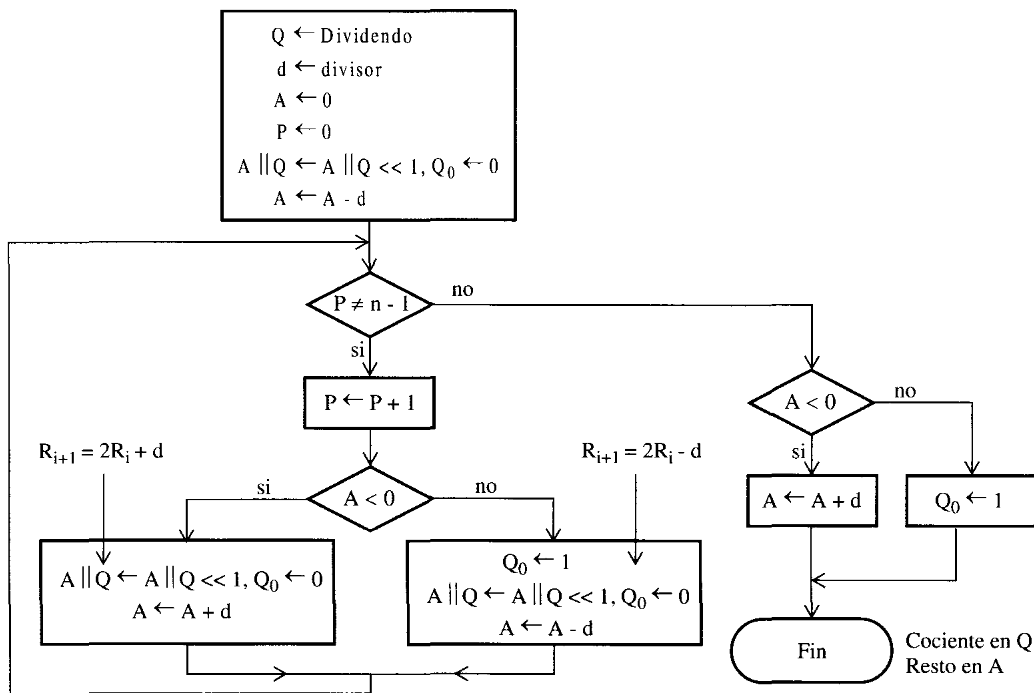


Figura 4.54: Diagrama de flujo del método de no restauración

Como los restos parciales pueden ser negativos y al final del algoritmo se desea tener un resto positivo, se modifica el último paso al salir del bucle para asegurar esta condición. Se puede ver que en este algoritmo se efectúa una suma en cada paso de la división, mientras que en el método de restauración había que realizar una suma para cada dígito del cociente que era 1 y dos sumas cuando era 0.

4.6.5 Ejemplo: Algoritmo de no restauración

En la Tabla 4.14 se muestra el mismo ejemplo que se resolvió por el método de restauración, utilizando ahora el método de no restauración. ♦

Paso	Acción	A	Q	d	P
0	Inicializar registros $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$	0000 0001 1100	1011 0110 0110	0101	0
1	$P \leftarrow P + 1$ $A < 0$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A + d$	 1000 1101	 1100 1100		1
2	$P \leftarrow P + 1$ $A < 0$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A + d$	 1011 0000	 1000 1000		2
3	$P \leftarrow P + 1$ $A \geq 0$ $Q_0 \leftarrow 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d$	 0000 0001 1100	 1001 0010 0010		3
4	$A < 0$ $A \leftarrow A + d$	 0001 **Resto**	 0010 **Cociente**		3

Tabla 4.14: Ejemplo de aplicación del método de no restauración

Sección 4-7

Estructura de la unidad aritmético-lógica (ALU)

Los circuitos que emplea un procesador para ejecutar un programa se combinan en la unidad aritmético-lógica (ALU). La complejidad de la ALU viene impuesta de una parte, por los tipos de operaciones que puede ejecutar y, de otra, por la forma en que las efectúa. En el caso de que se empleen algoritmos análogos para la realización de las diferentes operaciones de la ALU se puede simplificar su diseño. En concreto para la multiplicación y la división es posible compartir los mismos recursos de cálculo.

La Figura 4.55 muestra la estructura de uno de los diseños de ALU más utilizado. Además de las entradas para los operandos la ALU posee diversas señales de control (sumar, restar,....), que se utilizan para seleccionar las diferentes funciones aritmético-lógicas que se desean ejecutar.

Por simplicidad, en la Figura 4.55 no se han incluido los circuitos (por otro lado inmediatos) que hacen las distintas funciones lógicas (AND, OR, NOT,...). Las operaciones de multiplicar y dividir se pueden realizar utilizando alguno de los algoritmos estudiados en los apartados anteriores a base de operaciones elementales de suma y resta.

Para el almacenamiento de los operandos y del resultado se usan 3 registros:

a) El registro acumulador (*A*)

b) El registro multiplicador/cociente (*MQ*)

c) El registro de datos (*RDAT*)

Los registros *A* y *MQ* pueden desplazarse de forma conjunta a derecha e izquierda como si fuesen un único registro. *RDAT* es un registro auxiliar empleado por la ALU.

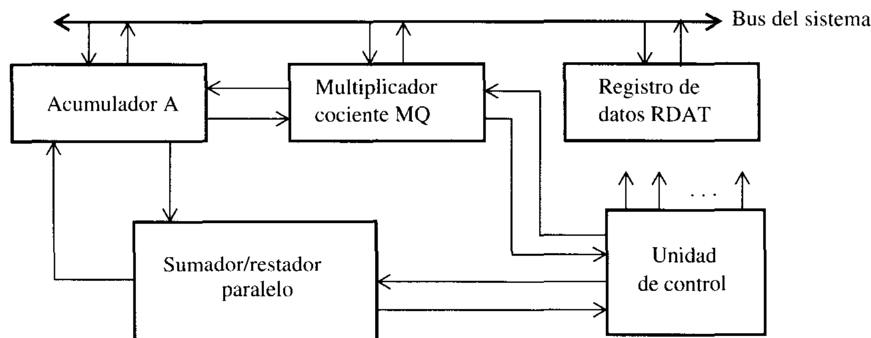


Figura 4.55: Estructura básica de una ALU

4.7.1 ALU´s integradas

Es factible empaquetar una ALU completa, del tipo que se acaba de describir, dentro de un único circuito integrado. El número de conexiones externas necesarias para controlar la ALU y poder acceder así a sus registros internos, limita la longitud de los datos que pueden tratarse por una ALU de este tipo. Es normal que dicha longitud sea de 2 ó de 4 bits. Esta anchura puede parecer pequeña, sin embargo, hay que tener presente que estas unidades se diseñan de manera que se puedan conectar fácilmente en cascada para formar una ALU capaz de procesar en paralelo un número arbitrario de bits. El objetivo de construir este tipo de módulos es doble:

1) Si se disponen de módulos aritméticos generales, es posible utilizarlos en muchas aplicaciones. Esto implica reducir el número de módulos distintos que se construyen, y aumentar así su volumen de producción con la consiguiente reducción en los costes.

2) Los módulos se pueden utilizar en procesadores, en los que la operación específica que va a ejecutar la unidad se selecciona dinámicamente por la unidad de control del procesador.

Una de las ALU's integradas que más se emplea es el circuito integrado *SN74x181* (ver Figura 4.56), que contiene las siguientes entradas:

a) 2 vectores de datos A y B de 4 bits cada uno

b) 4 señales de control (c_3, c_2, c_1, c_0) que permiten elegir una determinada función

c) 1 señal de control M para seleccionar entre funciones aritméticas y lógicas

d) 1 entrada c_e de arrastre

y como salidas:

e) 1 vector de resultado de la operación R de 4 bits

f) 1 salida c_s de arrastre

g) 2 señales P y G de propagación y generación de arrastre, que permiten construir circuitos más grandes con aceleración de arrastres

h) 1 señal EQ que indica la igualdad entre los operandos de entrada A y B

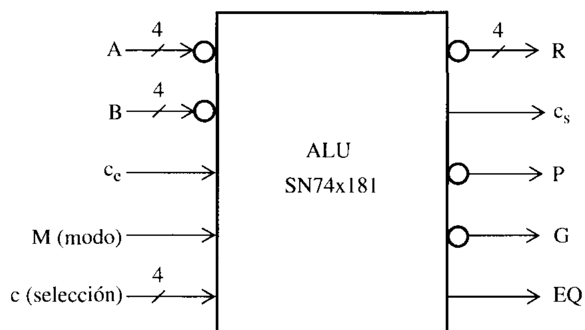


Figura 4.56: Señales de entrada-salida en la ALU SN74x181

En la Tabla 4.15 se muestra una descripción de la salida *R* del *SN74x181*. *R*, *A* y *B* representan los siguientes vectores:

$$R = (R_3, R_2, R_1, R_0)$$

$$A = (A_3, A_2, A_1, A_0)$$

$$B = (B_3, B_2, B_1, B_0)$$

v(R), v(A) y v(B) son sus respectivos valores decimales:

$$v(R) = \sum_{i=0}^{3} R_i 2^i \qquad v(A) = \sum_{i=0}^{3} A_i 2^i \qquad v(B) = \sum_{i=0}^{3} B_i 2^i$$

Selección $c_3c_2c_1c_0$	**M = 1** **Función lógica**	**M = 0** **Función aritmética** **v(R) = s mod 16**
0000	$R = \overline{A}$	$s = v(A) - 1 + c_e$
0001	$R = \overline{A} \vee \overline{B}$	$s = v(A \wedge B) - 1 + c_e$
0010	$R = \overline{A} \vee B$	$s = v(A \wedge \overline{B}) - 1 + c_e$
0011	$R = 1111$	$s = 1111 + c_e$
0100	$R = \overline{A} \wedge \overline{B}$	$s = v(A) + v(A \vee \overline{B}) + c_e$
0101	$R = \overline{B}$	$s = v(A \wedge B) + v(A \vee \overline{B}) + c_e$
0110	$R = A \oplus \overline{B}$	$s = v(A) - v(B) - 1 + c_e$
0111	$R = A \vee \overline{B}$	$s = v(A \vee \overline{B}) + c_e$
1000	$R = \overline{A} \wedge B$	$s = v(A) + v(A \vee B) + c_e$
1001	$R = A \oplus B$	$s = v(A) + v(B) + c_e$
1010	$R = B$	$s = v(A \wedge \overline{B}) + v(A \vee B) + c_e$
1011	$R = A \vee B$	$s = v(A \vee B) + c_e$
1100	$R = 0000$	$s = v(A) + v(A) + c_e$
1101	$R = A \wedge \overline{B}$	$s = v(A \wedge B) + v(A) + c_e$
1110	$R = A \wedge B$	$s = v(A \wedge \overline{B}) + v(A) + c_e$
1111	$R = A$	$s = v(A) + c_e$

Tabla 4.15: Funciones de la ALU SN74181

El significado de los operadores que aparecen en la Tabla 4.15 es el siguiente:

$A \wedge B$ AND de los vectores A y B

$A \vee B$ OR de los vectores A y B

v(A) + v(B)	suma aritmética de los vectores A y B
v(A) - v(B)	resta aritmética de los vectores A y B
$\overline{A}$, $\overline{B}$	NOT (A), NOT (B)

Las otras señales de salida vienen descritas por:

$c_s = 1$ si $s \geq 16$ ó $s < 0$

$c_s = 0$ en cualquier otro caso

$EQ = R_3 \wedge R_2 \wedge R_1 \wedge R_0$ detecta la igualdad de A y B

La entrada *M* del *SN74x181* selecciona entre operaciones aritméticas y lógicas. Cuando $M = 1$, se seleccionan las operaciones lógicas y cada salida R_i es función sólo de los correspondientes datos de entrada A_i y B_i. Ningún arrastre se propaga entre etapas y se ignora la entrada c_e. Las entradas c_3, c_2, c_1 y c_0 determinan una operación lógica particular. Se puede seleccionar cualquiera de las 16 funciones lógicas combinacionales de dos variables que hay.

Cuando $M = 0$, se seleccionan operaciones aritméticas, se propagan los arrastres entre las etapas y c_e se utiliza como un arrastre de entrada a la etapa menos significativa. Para aquellas operaciones que requieran más de 4 bits, se pueden disponer en cascada múltiples ALUS´s *SN74x181* de la misma forma que se conectaban cuatro sumadores con aceleración de arrastres de 4 bits (ver Figura 4.18) para construir un sumador de 16 bits. El arrastre de salida c_s de cada ALU se conecta al arrastre de entrada c_e de la siguiente etapa más significativa. A todas las ALU´s se les aplican las mismas señales (M, c_3, c_2, c_1 y c_0) para seleccionar la función que realiza.

Por ejemplo para efectuar la *suma en complemento a 2*, con las entradas c_3, c_2, c_1 y c_0 se selecciona la función $v(A) + v(B) + c_e$. La entrada c_e a la ALU menos significativa normalmente se pone a 0 durante la operación de suma. Si la operación fuese la *resta en complemento a 2*, con las entradas c_3, c_2, c_1 y c_0 se selecciona la función $v(A) - v(B) - 1 + c_e$. En este caso la entrada c_e a la ALU menos significativa normalmente se pone a 1, porque c_e actúa como el complemento del dígito que se lleva durante la operación de resta. El *SN74x181* proporciona otras operaciones aritméticas, tales como $v(A) - 1 + c_e$ que es útil en algunas aplicaciones (por ejemplo decrementar en 1). Dispone también de un conjunto de operaciones aritméticas un tanto esotéricas como $v(A \wedge \overline{B}) + v(A \vee B) + c_e$, que casi nunca se emplean en la práctica.

Debe observarse (ver Figura 4.56) que las entradas de los operandos A_3, A_2, A_1, A_0, B_3, B_2, B_1, B_0 y las salidas R_3, R_2, R_1, R_0 del *SN74x181* son activas en baja. El *SN74x181* se puede utilizar con operandos y salidas que son activas en alta. En este caso se debe construir una versión diferente de la Tabla 4.15. Cuando $M = 1$, las funciones lógicas que se obtienen son precisamente las duales de las que se muestran en la Tabla 4.15. Cuando $M = 0$, las funciones aritméticas son distintas de las que se dan en la Tabla 4.15.

Sección 4-8

Aritmética en coma flotante

La notación en *coma fija* resulta conveniente para representar valores enteros de magnitud pequeña y números fraccionarios escalados, es decir, con la posición de la coma decimal fijada a priori. Todos los algoritmos de las operaciones aritméticas que se han visto hasta ahora suponen que los números están representados en coma fija.

Estos mismos algoritmos se podrían aplicar también a los números reales representados en *coma flotante* a condición de tener en cuenta el escalado, lo que implica mantener en todo momento un conocimiento de la posición correcta donde se localiza la separación de las partes entera y decimal. Es posible manejar los problemas de escalado mediante programación, sin embargo el programa resultante tiende a ser ineficaz debido a los pasos adicionales que se requieren para mantener los factores de escala, e impone al programador una tarea innecesaria y propensa a errores.

Los procesadores en coma flotante manejan los factores de escala de forma automática. El hardware es más complejo, pero como contrapartida tiene la ventaja de que la operación del computador es mucho más eficaz. Estas consideraciones son las que justifican el diseño de una unidad aritmética en coma flotante.

Para poner de manifiesto la necesidad de la representación de números en coma flotante, conviene en primer lugar recordar como maneja un computador los números en coma fija. La forma más natural en principio para operar con números grandes en una máquina que posee una longitud de palabra pequeña es encadenar varias de estas palabras para disponer de un mayor rango para representar los números. Una primera cuestión que surge es la siguiente: ¿cómo trata el computador la parte decimal de un número?. En la representación en coma fija, afortunadamente esto no plantea ningún problema. A modo ilustrativo considérense los dos cálculos siguientes utilizando aritmética decimal.

Caso 1: Aritmética entera

```
  7 6 3 2 1 3 5
+ 1 7 9 4 8 2 1
  -------------
  9 4 2 6 9 5 6
```

Caso 2: Aritmética en coma fija

```
  7 6 3,2 1 3 5
+ 1 7 9,4 8 2 1
  -------------
  9 4 2,6 9 5 6
```

Aunque el primer caso utiliza *aritmética entera* y el segundo *aritmética decimal* (es decir con parte fraccionaria) los cálculos son totalmente idénticos. Este principio se puede extender a la unidad aritmética de un computador. Todo lo que un programador tiene que hacer es recordar donde está colocada la coma. Todas las entradas al computador se escalan de forma que se adapten a este convenio y todas las salidas están también escaladas de forma análoga. Las operaciones internas se realizan como si los números fuesen enteros. Esta organización es lo que se conoce como aritmética en coma fija, porque la coma binaria (que

separa la parte entera de la fraccionaria) se supone que permanece en la misma posición. Es decir hay siempre el mismo número de dígitos antes y después de la coma. La ventaja de la representación de números en coma fija es que para su implementación no se necesita un software o un hardware especialmente complejo. Un sencillo ejemplo pone de manifiesto esta idea de su sencillez.

Sea un número de 8 bits, donde los 4 más significativos representan la parte entera y los 4 menos significativos la parte fraccionaria. Se desea imprimir la suma de los dos números siguientes: 3,625 y 6,5. Como primer paso el programa de entrada convierte ambos números a su forma binaria:

$$3{,}625_{10} \rightarrow 11{,}101_2 \rightarrow 0011{,}1010_2 \text{ (en 8 bits)}$$

$$6{,}5_{10} \rightarrow 110{,}1_2 \rightarrow 0110{,}1000_2 \text{ (en 8 bits)}$$

El computador ahora considera estos números como 00111010 y 01101000 respectivamente. Conviene recordar que a todos los efectos la coma decimal es imaginaria. Estos números se suman de la forma usual y se obtiene:

```
  0 0 1 1 1 0 1 0
  0 1 1 0 1 0 0 0
-----------------
1 1 0 1 0 0 0 1 0 = 162 (si se interpreta como un número binario sin signo de 8 bits)
```

El programa de salida ahora toma el resultado y lo separa en su parte enterra 10100, y en su parte fraccionaria, 0010 e imprime la respuesta correcta 10,125. Conviene observar que un número representado en coma fija puede utilizar algunas palabras para conseguir un mayor rango de valores que el que permite una palabra simple. Sin embargo los números en coma fija tienen sus limitaciones. Considérese, por ejemplo el trabajo de un astrofísico que investiga la conducta del sol. Puede verse confrontado con cantidades tales como la masa del sol (1 990 000 000 000 000 000 000 000 000 000 g) y la masa de un electrón 0,000 000 000 000 000 000 000 000 000 910 956 g).

Si el astrofísico emplea para sus cálculos aritmética en coma fija, necesitaría un número extraordinariamente grande de bytes para representar todo el rango de números. Un único byte representa números en el rango 0-255 o aproximadamente de 0 a 1/4 de millar. Si el astrofísico necesitase trabajar simultáneamente con números astronómicamente grandes y microscópicamente pequeños, requeriría unos 14 bytes para la parte entera y 12 bytes para la parte fraccionaria, es decir ¡un número de 208 bits!. Una pista para resolver el problema del gran número de bits está en el hecho de que ambos números contienen un gran número de ceros y pocos dígitos significativos.

4.8.1 La representación de números en coma flotante

Los computadores a menudo manejan, representan, y almacenan números en un formato de *coma flotante*. De la misma forma que el número decimal 124,56 se puede representar como $0{,}12456 \times 10^3$, un computador maneja los números binarios de una forma similar. Por ejemplo, 1101101,1101101 se puede representar internamente como $0{,}11011011101101 \times 2^7$ (el 7 se almacena también en formato binario). La notación en coma flotante se denomina algunas veces *notación científica*. Antes de considerar más detalladamente los números en coma flotante es necesario considerar las ideas de *rango, precisión y exactitud* que están estrechamente relacionadas con la forma en que los números se representan en coma flotante.

- **Rango**. El rango de un número indica cuán grande o cuán pequeño puede ser. En el ejemplo del astrofísico se consideraban números tan grandes como 2×10^{23} y tan pequeños como 9×10^{-28},

lo que representa un rango de 10^{51} o 51 décadas. El rango de los números representados en un computador debe ser suficiente para la gran mayoría de los cálculos que son probables que se puedan efectuar. Si se va a utilizar el computador en una aplicación específica donde se conoce a priori que el rango de los datos es pequeño entonces el rango de los números válidos puede restringirse, simplificando los requisitos del sistema.

- **Precisión**. La precisión de un número se corresponde con el número de cifras significativas utilizadas para representarlo. Por ejemplo la constante π se puede escribir como 3,142 ó 3,141592. El último caso es más preciso que el primero porque representa a π en una parte en 10^7 mientras el primero lo hace en una parte en 10^4.
- **Exactitud**. La precisión es una medida de la fidelidad de una cantidad. Por ejemplo, se puede decir que π = 3,141 ó π = 3,241592. Se puede ver que en el primer caso se tiene un número que tiene poca precisión en comparación con el primero pero que sin embargo es más exacto. En un mundo ideal, precisión y exactitud deberían ir de la mano. Sin embargo en el mundo real con computadores que poseen longitudes de palabra finita se está confrontado con el diseño de algoritmos numéricos que garanticen la exactitud que la precisión disponible permite.

Un número *x* en coma flotante se representa en la forma siguiente:

$$x = m \times B^e$$

donde *m* recibe el nombre de *mantisa* o *argumento*, *e* es el *exponente* y *B* la *base o raíz*. La forma en que un computador almacena los números en coma flotante es dividiendo la secuencia binaria que lo representa en dos campos tal como se ilustra en la Figura 4.57.

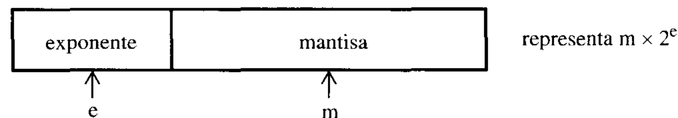


Figura 4.57: Almacenamiento de un número binario en coma flotante como exponente y mantisa

La base *B* está implícita y no necesita almacenarse, puesto que es la misma para todos los números. En lo sucesivo se supondrá que la base es dos. No es necesario que un número en coma flotante ocupe una única posición de memoria. Si la longitud de palabra es de 8 bits este tipo de representación no sería de utilidad. Con frecuencia una serie de palabras se agrupan para formar el número en coma flotante (ver Figura 4.58). La separación entre exponente y mantisa no necesita caer en la frontera de una palabra. Esto es, una mantisa podría típicamente ocupar tres bytes y el exponente un byte en un número en coma flotante que ocupa dos palabras de 16 bits.

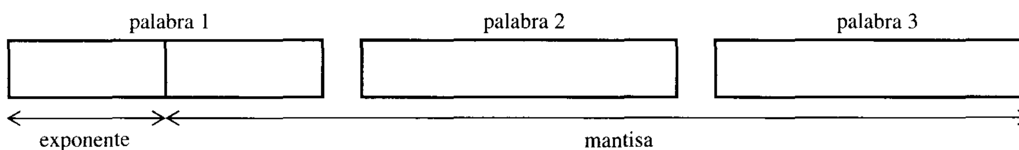


Figura 4.58: Utilización de algunas palabras para representar un número en coma flotante

4.8.2 La normalización de números en coma flotante

Por convenio la mantisa en coma flotante se *normaliza* siempre (a menos que sea igual a cero) de manera que se expresa en la forma 0,1.....× 2^e. Por el momento sólo se consideran mantisas positivas. Si el resultado de un cálculo fuera 0,01.....× 2^e se normalizaría para dar 0,1.....× 2^{e-1}. Análogamente, el resultado 1,01.....× 2^e se normalizaría a 0,101.....× 2^{e+1}. Una de las ventajas que proporciona normalizar la mantisa es que se optimiza la precisión con la que se opera en los cálculos. Por ejemplo, la mantisa de 8 bits no normalizados 0,00001010 tiene solamente 4 bits significativos, mientras que su valor normalizado 0,10100011 tiene 8 bits significativos. Existe sin embargo una ligera diferencia entre los números decimales normalizados tal como se utilizan en ciencias e ingenierías y los números binarios normalizados. Un número decimal en coma flotante se normaliza de forma que su mantisa está en el rango [1,00....0 9,99....9]. La mantisa normalizada *m* de un número binario positivo en coma flotante es de la forma:

$$m \in [0{,}100\ldots0 \quad 0{,}111\ldots1]$$

Es decir $1/2 \leq m < 1$. Una excepción especial tiene que hacerse con el caso del cero, ya que este número no se puede normalizar. En la representación en complemento a 2 la mantisa normalizada *m* de un número binario negativo en coma flotante se almacena de la forma:

$$m \in [1{,}011\ldots1 \quad 1{,}000\ldots0]$$

En este caso la mantisa negativa *m* pertenece al intervalo $-1/2 > m \geq -1$. Se ve pues que los números en coma flotante están restringidos a uno de los tres rangos descritos en la Figura 4.59.

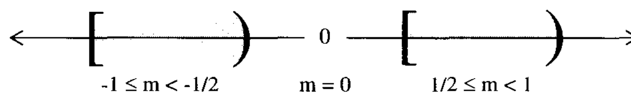


Figura 4.59: Rango de mantisas normalizadas en complemento a 2 que son válidas

En la representación magnitud-signo la mantisa normalizada de un número binario en coma flotante está en el rango [0,100...0 0,111...1] para números positivos y [1,100...0 1,111...1] para números negativos. Como se verá posteriormente, el formato IEEE para números en coma flotante emplea una representación del tipo magnitud-signo con un 1 a la izquierda de la coma (las mantisas caen en el rango [-1,11...1 -1.00...0] ó [1.00...0 1,11...1]. La mantisa *m* está ligada a pertenecer a los intervalos $-2 < m \leq 1$, $m = 0$ ó $1 \leq m < 2$.

4.8.3 Exponentes polarizados

Una representación en coma flotante de números debe considerar mantisas y exponentes tanto positivos como negativos. Por ejemplo en notación decimal esto corresponde a:

$$+0{,}123 \times 10^{12} \qquad -0{,}756 \times 10^{9}$$

$$+0{,}176 \times 10^{-3} \qquad -0{,}459 \times 10^{-7}$$

La mantisa de un número en coma flotante se representa a menudo como un número en complemento a 2. Sin embargo, el exponente a veces se representa en forma *polarizada*. Si se tiene un exponente de *m* bits, hay 2^m posibles números enteros sin signo desde 000...0 a 111...1. Supóngase ahora que estos números se

reetiquetan no desde 0 hasta 2^m-1 sino desde -2^{m-1} a $+2^{m-1}-1$ al restar un valor constante (o polarización) de 2^{m-1} de cada uno de los números. Lo que se tiene así es una serie binaria natural continua desde 0 a N que representa a los números desde P a N - P. Por ejemplo, si la serie (en decimal) fuera 0, 1, 2, 3, 4, 5, 6, 7 se podría restar P = 4 de cada número de la serie para generar una nueva serie -4, -3, -2, -1, 0, 1, 2, 3. Esto es realmente un nuevo método de representar números negativos añadiendo una constante al número más negativo para que el resultado sea igual a cero. En el ejemplo anterior, se ha añadido 4 a cada número de forma que -4 se representa por 0, -3 por +1 etc.

Se crea así un exponente polarizado sumando una constante al exponente verdadero de forma que el exponente polarizado viene dado por $e' = e + P$, donde e' es el exponente polarizado, e es el exponente verdadero y P un sumando de ponderación. El valor de P es con frecuencia 2^{m-1} ó $2^{m-1}-1$. En la Tabla 4.16 se muestra lo que sucede en el caso en que $m = 4$ y $P = 2^3 = 8$. Por ejemplo, si x = 1010,1111 se normaliza a 0,10101111 $\times$ 2^4. El exponente verdadero es e = +4 que se almacena como un exponente polarizado e' = 4+ 8 = 12 = 1100_2. La representación polarizada del exponente presenta algunas ventajas:

Representación binaria	Exponente verdadero	Forma polarizada
0000	-8	0
0001	-7	1
0010	-6	2
0011	-5	3
0100	-4	4
0101	-3	5
0110	-2	6
0111	-1	7
1000	0	8
1001	1	9
1010	2	10
1011	3	11
1100	4	12
1101	5	13
1110	6	14
1111	7	15

Tabla 4.16: Relación entre los exponentes verdadero y polarizado (P = 8)

1) El exponente más negativo se representa por cero. Por convenio el valor en coma flotante del cero en el sistema de exponente polarizado (ver Figura 4.60) se representa por 0,00....0 $\times$ 2^0 que es una mantisa y un exponente igual a cero (que es el exponente más negativo). En realidad el valor 0 se podría representar con m = 0 y cualquier exponente porque evidentemente $0 \times 2^n = 0$.

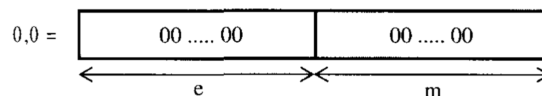


Figura 4.60: Representación del cero con un exponente polarizado

Sin embargo, puede ocurrir que al realizar determinadas operaciones que deben dar un resultado de 0, a causa de los errores de redondeo, se genere algún 1 en las posiciones menos significativos de la mantisa. Esto sugiere que el exponente más adecuado para representar el 0 sea precisamente aquél que tiene el mayor valor negativo. De esta manera se logra que los errores de redondeo producidos en la mantisa, den un número tan próximo a cero como sea posible.

2) La representación del cero es una secuencia de 0's tanto en la mantisa como en el exponente. Se evitan así ambigüedades en la representación del cero en coma flotante y se simplifica la comprobación de la igualdad de un número con 0. Teóricamente pueden existir muchas maneras de representar un cero en coma flotante (siempre que la mantisa sea cero y con cualquier valor del exponente). En algunos computadores, cuando un cálculo produce una mantisa igual a cero, el exponente se deja en el valor que tenía al final de la operación, lo que provoca que el cero no sea único. Una característica deseable, en el diseño de cualquier procesador, es tener una única representación para el cero. En aritmética en coma fija, el cero es único y se representa por un número en el que todos sus bits son ceros.

3) Cuando se suman o restan dos números en coma flotante se deben comparar los exponentes y hacerlos iguales, lo que resulta en una operación de *desplazamiento o alineamiento* para una de las mantisas. La comparación de los exponentes es relativamente directa, ya que los exponentes son siempre números positivos. Es suficiente un simple comparador.

4) Los exponentes que se almacenan forman una secuencia binaria natural. Esta secuencia es monótona creciente de forma que si se aumenta (disminuye) el exponente en 1 supone sumar (restar) 1 al exponente binario. En ambos casos el exponente binario polarizado se puede considerar que se comporta como un número binario sin signo.

4.8.4 Posibles sistemas en coma flotante

Para definir una representación en coma flotante para un determinado computador hay que seleccionar los siguientes elementos:

1) El número de palabras utilizadas
2) La representación de la mantisa (complemento a 2, magnitud-signo, etc)
3) La representación del exponente (polarizado, no polarizado, etc)
4) El número de bits dedicados a la mantisa y al exponente
5) La localización de la mantisa (antes o después del exponente)

El punto 4) merece algún comentario adicional. Una vez que se ha decidido el número total de bits en la representación en coma flotante (un número entero de palabras) hay que hacer una partición entre la mantisa y el exponente. Si se dedica un gran número de bits al exponente, el resultado es un número en coma flotante con un rango muy grande. Estos bits del exponente se han obtenido a expensas de la mantisa lo que reduce la precisión del número en coma flotante. Inversamente, si se aumentan los bits que están disponibles para la mantisa se mejora la precisión a costa del rango.

Debido a los cinco puntos anteriores, hay una multitud de formatos para la representación de números en coma flotante. Los ejemplos de la Figura 4.61 ilustran la representación de números en coma flotante

adoptada por algunos computadores (prácticamente cada computador utilizaba un formato diferente del resto). Con la aparición del microprocesador y la especificación introducida por el IEEE para números en coma flotante esta situación ha cambiado considerablemente.

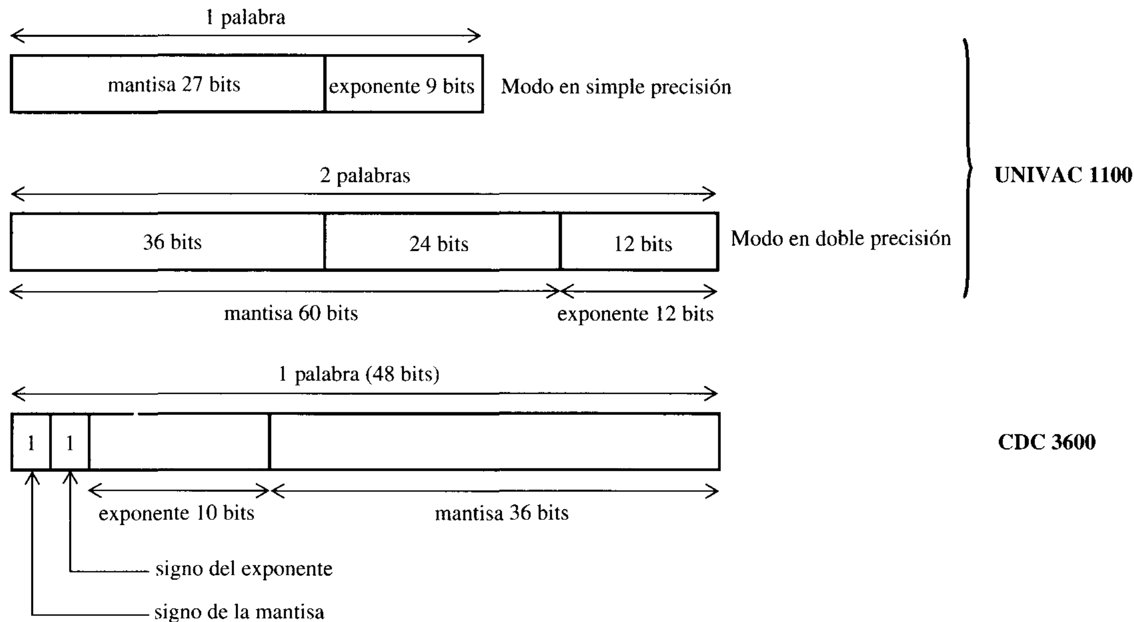


Figura 4.61: Algunos formatos de representación de números en coma flotante

El modo en simple precisión de Univac proporciona un rango de aproximadamente 10^{-76} a 10^{+76} con una precisión de 8 cifras decimales. En el modo en doble precisión el rango se aumenta hasta 10^{-614} a 10^{+614} y la precisión es equivalente a 18 cifras decimales. El modo en doble precisión se utiliza en aquellos cálculos numéricos donde se requiere una gran precisión. En general su empleo hace considerablemente más lenta la ejecución de los programas a menos que el computador posea una unidad en coma flotante especial de alta velocidad.

4.8.5 El formato IEEE de representación de números en coma flotante

El Institute of Electrical and Electronics Engineers (IEEE) ha propuesto un estándar de representación de números en coma flotante para las operaciones aritméticas en mini y microcomputadores (norma ANSI/ IEEE 754-1985). Con el fin de permitir su utilización en diferentes aplicaciones, IEEE especifica tres formatos básicos, llamados *simple*, *doble* y *cuádruple*. En la Tabla 4.17 se definen las características principales de estos tres formatos de representación de números en coma flotante. Los números se normalizan de forma que sus mantisas están contenidas en el rango $1 \leq m < 2$. Este rango corresponde a una mantisa con una parte entera igual a 1. Un número en coma flotante en el formato IEEE se define formalmente como:

$$x = -1^{s} \times 2^{e-P} \times 1,m$$

donde:

s = bit de signo, 0 = mantisa positiva, 1 = mantisa negativa

e = exponente polarizado por P

m = parte fraccionaria de la mantisa (la mantisa es 1,m donde el 1 de delante está implícito)

Tipo	Simple	Doble	Cuádruple
Anchura del campo en bits			
s = signo	8	11	15
e = exponente	1	1	1
d = bit delante de la mantisa	1	1	1
m = fracción de la mantisa	23	52	111
Anchura total	32	64	128
Bit de signo	0 = +, 1 = -	0 = +, 1 = -	0 = +, 1 = -
Exponente			
e máximo	255	2047	32767
e mínimo	0	0	0
polarización	127	1023	16383

Tabla 4.17: Formatos IEEE básicos en coma flotante

Así por ejemplo, un número en coma flotante de 32 bits en formato simple tiene una polarización de 127 y una parte fraccionaria para la mantisa de 23 bits. Hay dos puntos de particular interés que conviene precisar. El primero es que el formato IEEE adopta una representación magnitud-signo para la mantisa. Si $s = 1$ la mantisa es negativa y si $s = 0$ es positiva. El segundo es que la mantisa está siempre normalizada y pertenece al rango [1,000...000, 1,111...111]. Se observa que la normalización de un número en el formato IEEE es diferente a lo que se había comentado anteriormente. Si la mantisa está siempre normalizada, entonces el 1 de delante de la coma (la parte entera) es redundante cuando un número en el formato IEEE se almacena en memoria. Si se sabe que hay un 1 a la izquierda de la parte fraccionaria de la mantisa, no hay necesidad de almacenarlo (por esta causa a veces se le denomina *bit escondido*). De esta manera se elimina un bit en el almacenamiento, lo que permite mejorar la precisión de la mantisa al ampliarla en un bit. En la Figura 4.62 se muestra el formato de un número cuando se almacena en memoria.

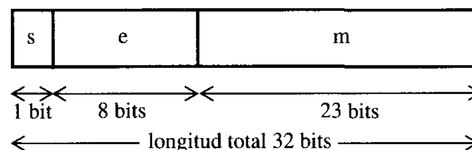


Figura 4.62: Formato IEEE del tipo simple de 32 bits

4.8.6 Ejemplo: Representación en el formato IEEE

Como ejemplo de utilización del formato simple de 32 bits de IEEE se considera la representación del número decimal -2345,125 en un computador que posee una longitud de palabra de 16 bits.

$-2345{,}125_{10} = -100100101001{,}001_2$ (número binario equivalente)

$= -1{,}00100101001001 \times 2^{11}$ (número binario normalizado)

Para especificar completamente el número hay que determinar el bit de signo s, el exponente polarizado e y la parte fraccionaria de la mantisa m.

1) La mantisa es negativa así que el bit de signo es $s = 1$

2) El exponente polarizado viene dado por $+11 + 127 = 138 = 10001010_2$

3) La parte fraccionaria de la mantisa es ,00100101001001000000000 (en 23 bits)

Este número se almacena en dos palabras consecutivas de 16 bits:

Con el fin de minimizar el espacio de almacenamiento en una memoria de 16 bits, los números en coma flotante se *empaquetan* de forma que el bit de signo, el exponente y la mantisa comparten parte de dos o más palabras de memoria. Cuando se va a realizar la operación en coma flotante, en primer lugar lo que se hace es desempaquetar los números y se separa la mantisa del exponente. Por ejemplo, el formato básico en simple precisión especifica una mantisa con una parte fraccionaria de 23 bits, que produce una mantisa de 24 bits cuando se desempaqueta y se reinserta el 1 que va por delante. Si el procesador en el que se van a procesar los números en coma flotante tiene una longitud de palabra de 16 bits, la mantisa desempaquetada ocupará 24 bits de los 32 bits que se disponen en las dos palabras.

Si cuando se desempaqueta un número, se permite que aumente el número de bits en su exponente y en su mantisa para rellenar todo el espacio disponible se dice que el formato es *extendido*. Al extender el formato de esta forma, se incrementa considerablemente el rango y la precisión del número en coma flotante. Por ejemplo, un número en formato simple se almacena en 32 bits. Cuando se desempaqueta la parte fraccionaria de la mantisa de 23 bits pasa a 24 bits al incluir el 1 que va por delante y la mantisa se amplía a 32 bits (como una palabra simple de 32 bits o como dos palabras de 16 bits).

Todos los cálculos se hacen con la precisión de los 32 bits de la mantisa extendida. Esto es particularmente ventajoso cuando se evaluan funciones trascendentes (p. ej. sen (x) ó cos (x)). Después de efectuar, en el formato extendido, una secuencia de operaciones el número en coma flotante se reempaqueta y almacena en memoria en su formato básico. En el formato simple de 32 bits, el exponente máximo e_{max} es +127 y el exponente mínimo e_{min} es -126 y no +128 y -127 como se podría esperar. El valor especial $e_{min} - 1$ (-127) se utiliza para codificar el cero y $e_{max} + 1$ para codificar más o menos infinito o una condición que se designa como *NaN* (Not a Number). Un *NaN* es una entidad especial que incorpora el formato IEEE y que permite la manipulación de formatos fuera del estándar.

4.8.7 Operaciones aritméticas en coma flotante

Dados los números:

$$x = m_x 2^{e_x} \qquad y = m_y 2^{e_y}$$

la Tabla 4.18 resume las operaciones básicas de la arimética en coma flotante (se supone $e_x \leq e_y$).

x + y	$(m_x 2^{e_x - e_y} + m_y) 2^{e_y}$
x - y	$(m_x 2^{e_x - e_y} - m_y) 2^{e_y}$
x × y	$(m_x \times m_y) 2^{e_x + e_y}$
x ÷ y	$(m_x \div m_y) 2^{e_x - e_y}$

Tabla 4.18: Operaciones aritméticas en coma flotante

En la Tabla 4.18 se observa que la mantisa del exponente más pequeño se desplaza e_x - e_y lugares a la derecha para la suma y la resta. Todas las operaciones aritméticas en coma flotante pueden producir una condición de *rebose*, si el resultado es demasiado grande (desbordamiento) o demasiado pequeño (subdesbordamiento), para poder representarse en la máquina. Los tipos de rebose a considerar son:

1) *Desbordamiento del exponente.* Un exponente positivo *e*, excede su valor máximo permitido. En algunos sistemas, el número *x* puede representarse como +∞ ó -∞.
2) *Subdesbordamiento del exponente.* Un exponente negativo *e*, excede su valor máximo posible. Esto significa que el número *x* es demasiado pequeño para poderse representar y se le puede considerar igual a 0.
3) *Subdesbordamiento de la mantisa.* Se puede producir en el proceso de alineamiento de las mantisas, si los dígitos se desplazan hacia la derecha más allá de su bit menos significativo. Cuando sucede esto se precisa alguna forma de redondear el resultado.
4) *Desbordamiento de la mantisa.* La suma de dos mantisas del mismo signo puede generar un arrastre del bit más significativo. Esto se puede corregir mediante una operación de renormalización que se explica posteriormente (desplazando un bit a la derecha la mantisa y ajustando el exponente).

4.8.8 Algoritmo de suma y resta en coma flotante

Al contrario de lo que pasa con los números enteros o en coma fija, los números en coma flotante no se pueden sumar en una única operación. Para poner esto de manifiesto considérese el siguiente ejemplo utilizando aritmética decimal. Sea x = 12345 e y = 567,89. En coma flotante estos dos números se pueden representar por:

$$x = 0{,}12345 \times 10^5 \qquad y = 0{,}56789 \times 10^3$$

Si estos dos números se fueran a sumar de forma manual no surgiría ningún problema:

$$\begin{array}{r} 12345 \\ +\quad 567{,}89 \\ \hline 12912{,}89 \end{array}$$

Sin embargo, cuando se expresan en formato normalizado se plantea el problema siguiente:

$$
\begin{array}{r}
0{,}1\,2\,3\,4\,5 \times 10^5 \\
+\ \underline{0{,}5\,6\,7\,8\,9 \times 10^3}
\end{array}
$$

no puede realizarse mientras los exponentes sean *diferentes*. Para efectuar una suma (o una resta) en coma flotante hay que realizar los siguientes pasos:

1) Seleccionar el número con menor exponente y desplazar su mantisa a la derecha, un número de pasos igual a la diferencia en valor absoluto de los exponentes de los dos operandos.
2) Hacer el exponente del resultado igual al mayor de los exponentes de los operandos.
3) Realizar la suma (resta) de las mantisas y determinar el signo del resultado.
4) Normalizar el resultado si es necesario.
5) Comprobar condiciones de rebose.

En el ejemplo anterior se tenía $x = 0{,}12345 \times 10^5$ e $y = 0{,}56789 \times 10^3$. El exponente de y es más pequeño que el de x lo que resulta en un incremento de 2 en su exponente y en la correspondiente división de su mantisa por 10^2 para obtener $0{,}0056789 \times 10^5$. Se puede ahora sumar x con el valor desnormalizado de y.

$$
\begin{array}{r}
x = 0{,}1\,2\,3\,4\,5\,0\,0 \times 10^5 \\
+\ y = \underline{0{,}0\,0\,5\,6\,7\,8\,9 \times 10^5} \\
0{,}1\,2\,9\,1\,2\,8\,9 \times 10^5
\end{array}
$$

Este resultado está ya en forma normalizada y no necesita ninguna renormalización posterior. Se observa que la respuesta se expresa con una precisión de 7 cifras significativas mientras que x e y se tienen sólo con 5 cifras significativas. Si el resultado se va a almacenar en un computador, su mantisa se tendría que reducir a 5 cifras significativas después de la coma (ya que se está trabajando con mantisas con 5 dígitos).

Cuando se realizan operaciones aritméticas en coma flotante de forma manual, con frecuencia se recurre a lo que se puede denominar *precisión flotante*. Si se necesita una mayor precisión simplemente se utilizan más dígitos en los cálculos. Los computadores emplean una representación fija para los números en coma flotante de manera que la precisión no se puede aumentar como consecuencia de un cálculo. Para poner esta idea de manifiesto considérese el siguiente ejemplo de suma de dos números binarios en coma flotante.

$$
\begin{array}{r}
x = 0{,}1\,1\,0\,0\,1 \times 2^4 \\
y = 0{,}1\,0\,0\,0\,1 \times 2^3
\end{array}
$$

El exponente de y se debe incrementar en 1 y su mantisa dividirse por 2 (deplazándola un lugar hacia la derecha) para conseguir que ambos exponentes sean iguales a 4.

$$
\begin{array}{r}
x = 0{,}1\,1\,0\,0\,1 \times 2^4 \\
y = \underline{0{,}0\,1\,0\,0\,0\,1 \times 2^4} \\
1{,}0\,0\,0\,0\,1\,1 \times 2^4
\end{array}
$$

En este caso se ha producido un desbordamiento de la mantisa y se debe proceder a un proceso de renormalización dividiendo la mantisa por 2 e incrementando en 1 el exponente.

$$x + y = 1{,}0\,0\,0\,0\,1\,1 \times 2^4 \quad \rightarrow \quad 0{,}1\,0\,0\,0\,0\,1\,1 \times 2^5$$

Se han ganado también dos cifras significativas lo que fuerza a que se tome algún tipo de acción. Por ejemplo se puede simplemente truncar el resultado para obtener:

$$x + y = 0{,}1\,0\,0\,0\,0 \times 2^5$$

En la Figura 4.63 se da un diagrama de flujo detallado del algoritmo de suma (resta) de números en coma flotante. De este diagrama de flujo conviene hacer las siguientes observaciones:

1) Como en muchas implementaciones el exponente comparte una palabra con la mantisa, es necesario separarlas antes de comenzar el proceso de la suma. Como ya se ha dicho anteriormente esta operación se denomina desempaquetamiento.

2) Si los dos exponentes difieren en más de $p + 1$, donde p es el número de cifras significativas en la mantisa, entonces el número menor es demasiado pequeño para afectar al mayor y por lo tanto el resultado es efectivamente igual al número mayor y no hay que realizar ningún tipo de acción. Por ejemplo, si la mantisa sólo tiene 4 dígitos, no tiene sentido sumar $0{,}1234 \times 10^{20}$ con $0{,}4567 \times 10^2$, porque el segundo sumando no produce ningún efecto.

3) Durante la renormalización se comprueba el exponente para ver si es menor que un valor mínimo o mayor que un valor máximo. Esto corresponde a comprobar el *subdesbordamiento* y el *desbordamiento del exponente* respectivamente. Cada uno de estos casos representa condiciones en las cuales el número está fuera del rango de números que puede manejar el computador. El subdesbordamiento generalmente conduciría a que el número se haga igual a cero, mientras que el desbordamiento resultaría en una condición de error que debe enviarse para su información al sistema operativo.

4) Los símbolos >> y << especifican un desplazamiento del dato que está a su izquierda hacia la derecha e izquierda respectivamente. A la derecha del símbolo se indica el número de bits de desplazamiento.

4.8.9 Redondeo y truncamiento

En el apartado anterior se acaba de ver que algunas de las operaciones en aritmética en coma flotante conducen a un incremento en el número de bits de la mantisa y que se debe utilizar alguna técnica para mantener constante el número de bits de la mantisa. La técnica más simple se llama *truncamiento* y consiste nada más que en eliminar los bits que no se necesitan. Por ejemplo, 0,1101101 cuando se trunca a cuatro dígitos significativos se convierte en 0,1101. Una técnica mucho mejor es el *redondeo*. Si el valor de los dígitos que se pierden es mayor que la mitad del bit menos significativo de los bits retenidos, entonces se añade un 1 al bit menos significativo de los que quedan. Por ejemplo, considérese el redondeo a 4 bits significativos de los números siguientes:

$$0{,}1\,1\,0\,1\,1\,0\,1 \rightarrow 0{,}1\,1\,0\,1 + 1 = 0{,}1\,1\,1\,0$$
$$0{,}1\,1\,0\,1\,0\,0\,1 \rightarrow 0{,}1\,1\,0\,1$$

El redondeo se prefiere siempre al truncamiento de una parte debido a que es más preciso y de otra porque da lugar a un error no polarizado. El truncamiento siempre disminuye el resultado lo que lleva a la

aparición de un error sistemático, mientras que el redondeo algunas veces lo reduce y otras lo aumenta. La desventaja del redondeo es que requiere que se efectúe una operación aritmética adicional sobre el resultado.

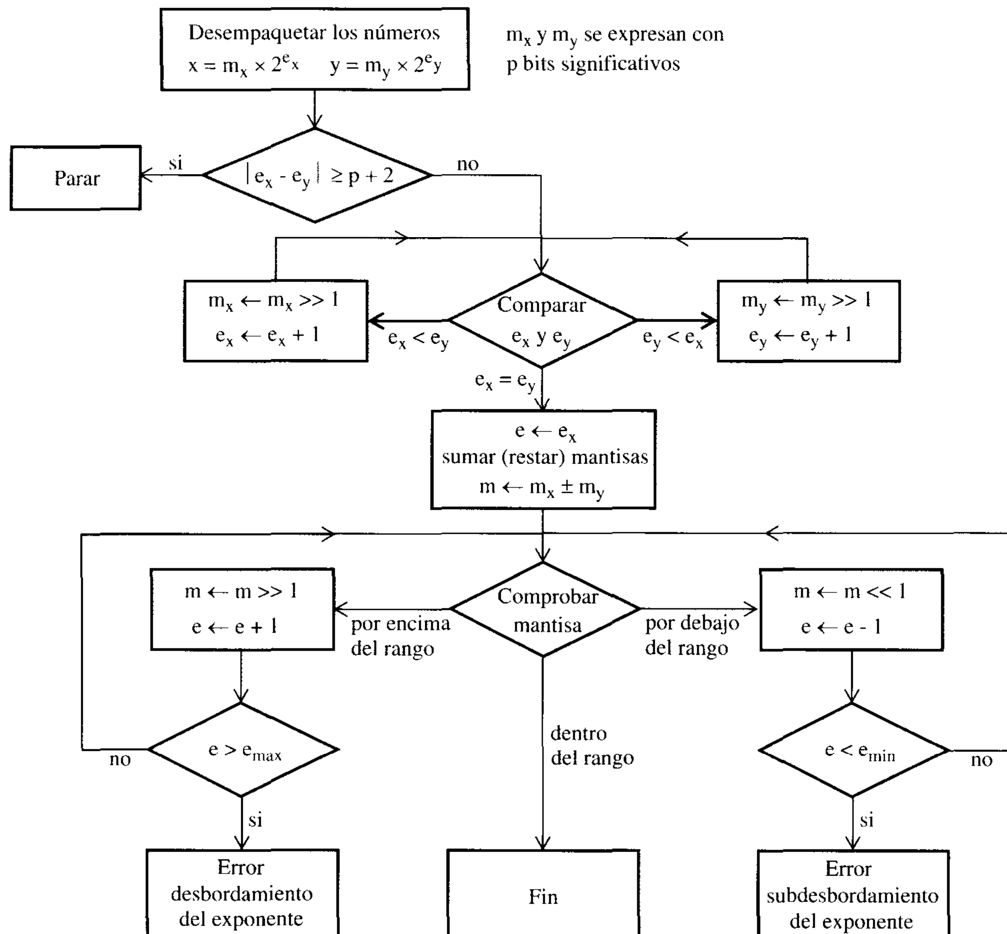


Figura 4.63: Diagrama de flujo del algoritmo de suma (resta) en coma flotante

4.8.10 Algoritmo de multiplicación y división en coma flotante

Se consideran un par de números en coma flotante representados por:

$$x = m_x\, 2^{e_x} \qquad y = m_y\, 2^{e_y}$$

la operación de multiplicación en coma flotante se puede definir como:

$$x \times y = (m_x \times m_y)\, 2^{e_x + e_y}$$

En el caso de la división, la multiplicación de las mantisas se sustituye por su división y la suma de los exponentes por su resta. Por ejemplo, cuando $x = 1{,}000 \times 2^{-2}$ e $y = -1{,}010 \times 2^{-1}$, el producto $x \times y$ se puede calcular como sigue:

1) *Sumar los exponentes*: -2 + (-1) = -3

2) *Multiplicar las mantisas*

```
           1 , 0 0 0
         - 1 , 0 1 0
         -----------
           0 0 0 0
         1 0 0 0
       0 0 0 0
     1 0 0 0
   -----------------
   - 1 , 0 1 0 0 0 0
```

Así el producto es $-1{,}0100 \times 2^{-3}$. En general el algoritmo para la multiplicación (división) es más simple que el de la suma (resta) ya que no precisa ni de la selección del operando con menor exponente, ni del alineamiento de las mantisas en coma flotante. Se distinguen cuatro pasos principales:

1) Realizar la suma (resta) de los exponentes de los operandos.

2) Realizar la multiplicación (división) de las mantisas y determinar el signo del resultado.

3) Normalizar y redondear el resultado si es necesario.

4) Comprobar condiciones de rebose.

La Figura 4.64 muestra la organización típica de un multiplicador (divisor) en coma flotante.

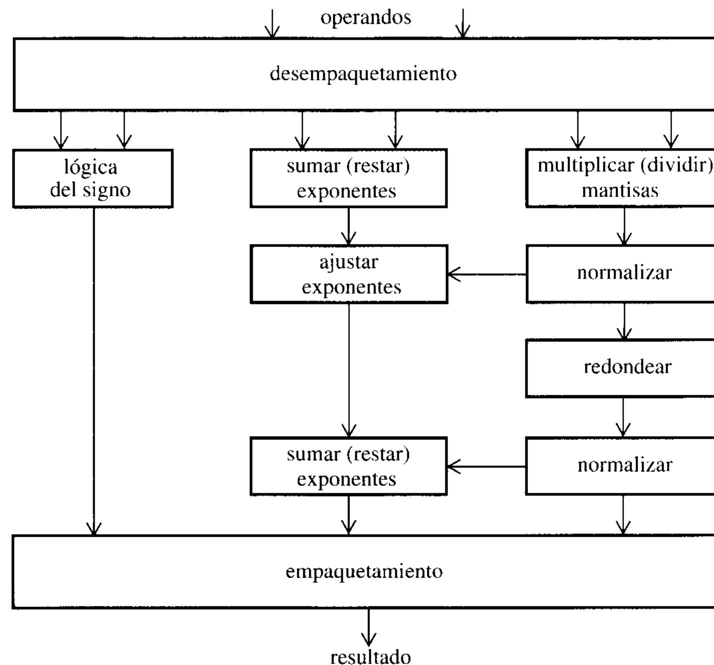


Figura 4.64: Organización de un multiplicador (divisor) en coma flotante

La primera etapa consiste en desempaquetar los operandos: se separan el bit de signo y los campos del exponente y de la mantisa, se reinsertan los bits escondidos de los operandos (en el caso de que el formato utilizado los posea) y se comprueba si hay condiciones de excepción en los operandos (por ejemplo mantisa igual a cero, necesidad de un desplazamiento de una de las mantisas para su alineamiento mayor que el número p de cifras significativas que se emplean en su representación, etc). La comprobación en esta etapa de que una mantisa es igual a cero puede ser útil porque si la operación es la de multiplicación queda ya determinado que el resultado de la operación aritmética es cero, también es necesaria para evitar una inserción impropia de un bit escondido. En la segunda etapa se realiza el cálculo del signo del resultado, la suma (resta) de los exponentes y la multiplicación (división) de las mantisas. A continuación el resultado se normaliza, redondea y renormaliza (si ocurre una condición de rebose durante el redondeo). La última etapa combina los diferentes campos, elimina el bit escondido y comprueba si se han producido condiciones de excepción tales como resultado igual a cero, desbordamiento y subdesbordamiento En la Figura 4.65 se muestra un diagrama de flujo del algoritmo de multiplicación (división) en coma flotante.

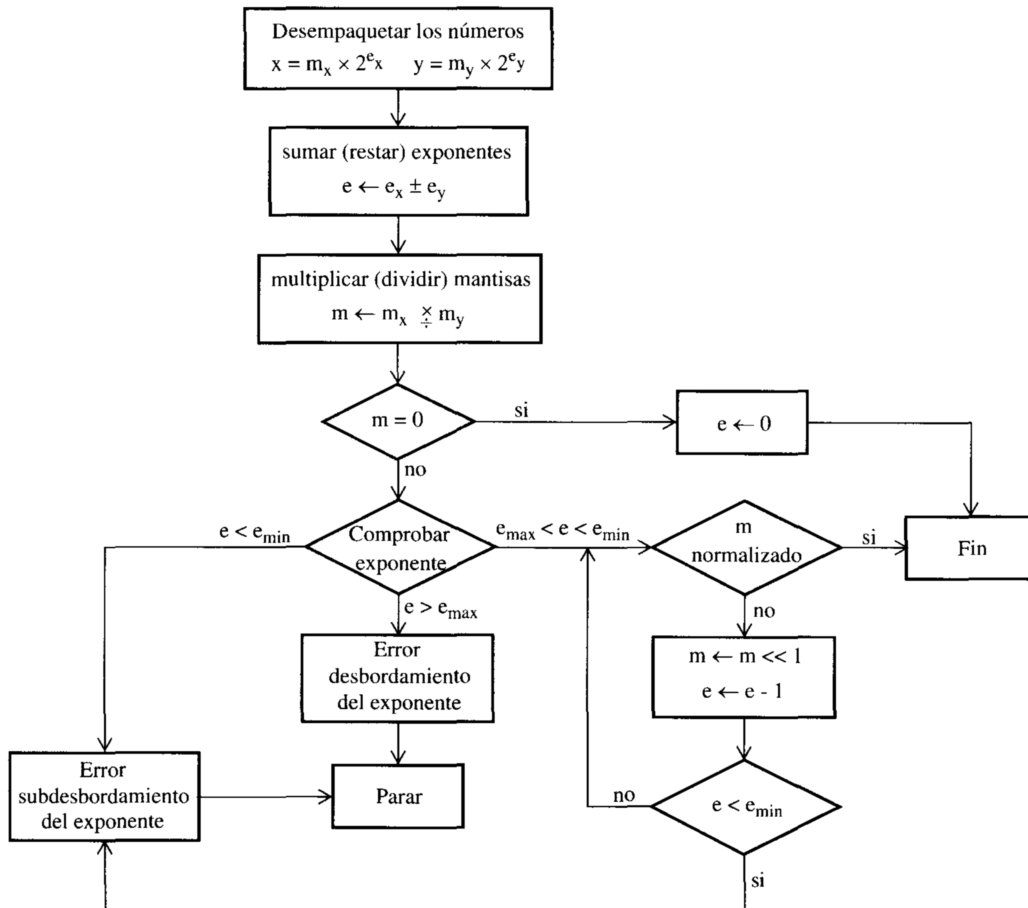


Figura 4.65: Diagrama de flujo del algoritmo de multiplicación (división) en coma flotante

4.8.11 Estructura básica de una unidad aritmética en coma flotante

Una unidad aritmética en coma flotante se puede realizar conectando dos ALU´s en coma fija tal como se muestra en la Figura 4.66. La unidad de tratamiento de mantisa se necesita para realizar las cuatro operaciones básicas sobre las mantisas, y se puede utilizar una ALU en coma fija de propósito general del tipo descrito en la Figura 4.55. Para la unidad de tratamiento de exponente es suficiente con un circuito más simple que sea capaz de sumar, restar y comparar exponentes. La comparación de los exponentes, como se ha comentado ya, se puede hacer con un comparador o simplemente restando los exponentes (ver sección 4-10 para una explicación más amplia de la operación de comparación).

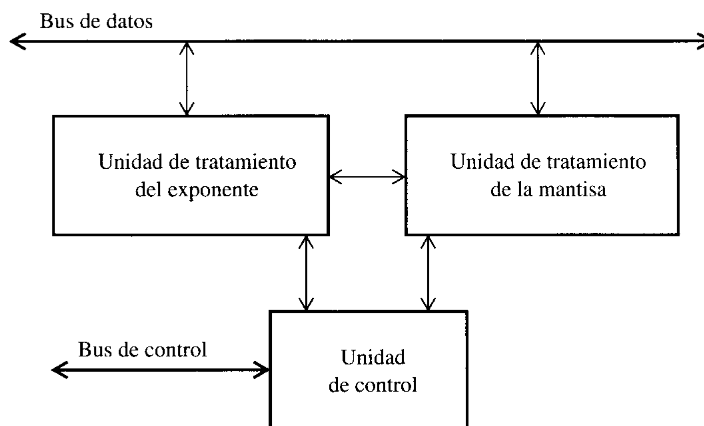


Figura 4.66: Estructura básica de una ALU en coma flotante

En la Figura 4.67 se presenta la estructura general de una ALU en coma flotante que utiliza como comparación la resta de exponentes.

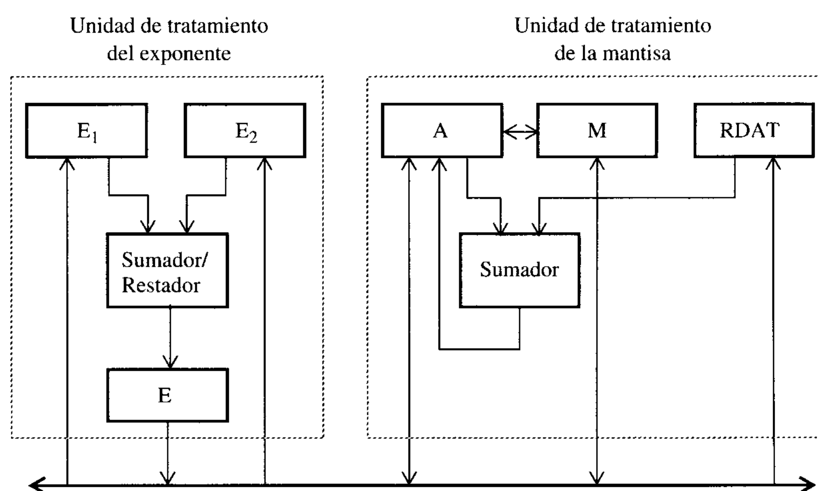


Figura 4.67: ALU en coma flotante

Los exponentes de los operandos de entrada se almacenan en los registros E_1 y E_2, que están conectados a un sumador paralelo que permite calcular $E_1 \pm E_2$. La comparación de los exponentes, que se necesita para la suma y la resta en coma flotante, se efectúa calculando $E_1 - E_2$ y almacenando el resultado en el registro E. El mayor de los exponentes queda determinado por el signo de E. El registro E puede además controlar el desplazamiento que se necesita de una de las mantisas para conseguir sus alineamientos (antes de que pueda tener lugar su suma o su resta). El contenido del registro E se va decrementado de forma secuencial hasta 0. Después de cada decremento se desplaza un dígito la mantisa apropiada. Una vez conseguido el alineamiento de las mantisas, se procesan de la forma normal. Finalmente se calcula el exponente del resultado y se almacena en el registro E.

Apéndice A
Componentes combinacionales

Un *circuito combinacional* es un circuito lógico cuyos valores de salida están determinados en cualquier instante de tiempo únicamente por los valores aplicados a sus entradas, y por tanto son independientes de los estados anteriores de las mismas. Los circuitos combinacionales no nos permiten pues almacenar el estado de las entradas y utilizarlas posteriormente para tomar decisiones, es decir, son circuitos que no tienen memoria.

La característica principal de un circuito combinacional es la función lógica que realiza. Esta función lógica describe el comportamiento del circuito pero no tiene por qué describir su estructura interna, por lo que es posible que existan diferentes circuitos lógicos que realizan la misma función. Los circuitos combinacionales tienen dos usos principales en los sistemas digitales:

a) *Transferencia de datos.* Controlan el flujo de señales lógicas de una parte del sistema a otra.

b) *Procesamiento de datos.* Procesan o transforman los datos realizando los cálculos necesarios.

En este apéndice se consideran los principales componentes combinacionales que se emplean en la descripción a nivel de registro de un sistema digital.

1) Puertas de palabras
2) Codificadores
3) Decodificadores
4) Multiplexores
5) Demultiplexores
6) Dispositivos lógicos programables

Aunque algunos de estos componentes ya se han introducido en los propios temas (por ejemplo los decodificadores) se expone a continuación brevemente, y con la finalidad de hacer el texto lo más autocontenido posible, la función lógica que realiza cada uno de ellos. No se consideran los circuitos sumadores, multiplicadores, comparadores y unidades aritmético-lógicas ya que a su estudio exclusivo se dedicó el tema 4.

Sección A-1

Puertas de palabras

Se utilizan para representar la realización de una función lógica, bit a bit, entre dos palabras o bien entre una palabra y un bit constante.

El símbolo de la puerta de palabra expresa la operación que se efectúa entre sus datos de entrada y la correspondencia entre símbolos y funciones lógicas es la misma que se utiliza con las puertas lógicas básicas, con la única diferencia de explicitar en las entradas y salidas el número de bits que contiene.

Así, dados $X = (x_{n-1},\dots, x_1, x_0)$ e $Y = (y_{n-1},\dots, y_1, y_0)$, en la Figura A.1 se muestran respectivamente las puertas AND, OR y OR exclusiva de palabras de *n* bits.

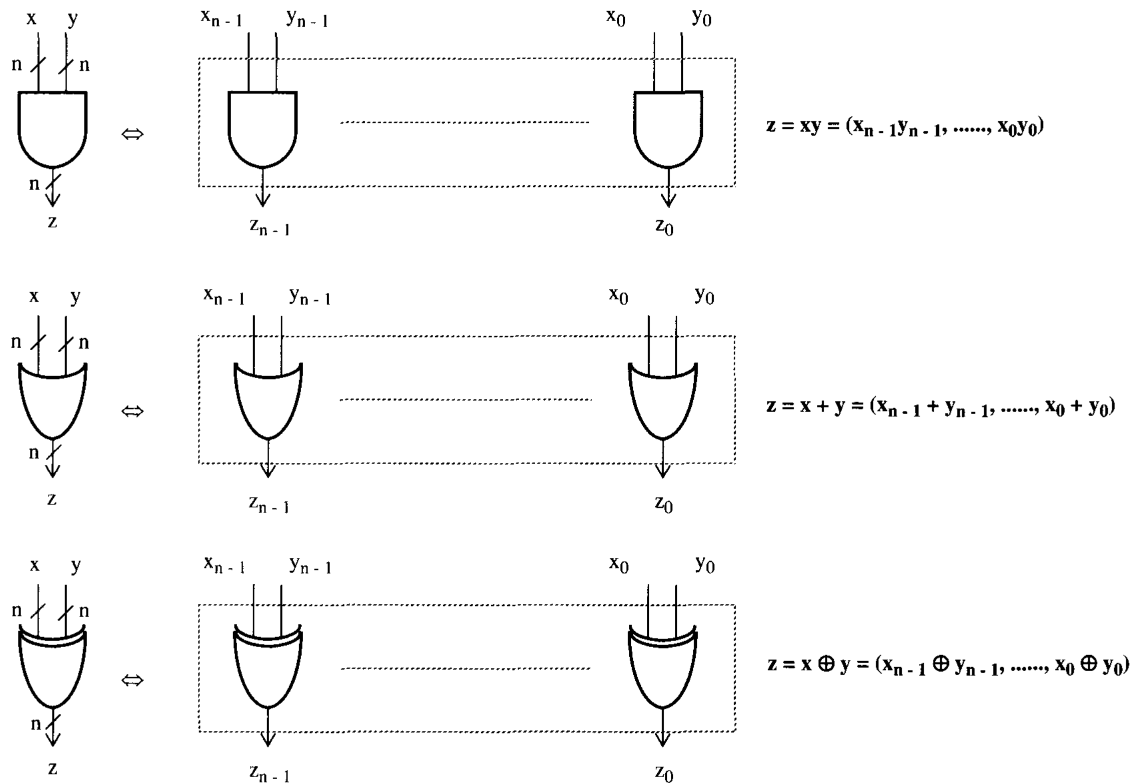


Figura A.1: Puertas AND, OR y OR exclusiva de palabras de *n* bits

Sección A-2

Codificadores

Los codificadores binarios son circuitos combinacionales que poseen 2^n entradas $y = (y_{2^n-1}, ..., y_1, y_0)$ y n salidas $z = (z_{n-1}, ..., z_1, z_0)$ tal como se muestra en la Figura A.2. Realizan la función inversa de los decodificadores y se pueden considerar como un conversor de código que transforma un código del tipo 1 entre 2^n a un código binario. Es decir, en cualquier instante de tiempo sólo una de las variables de entrada tiene valor 1 y las salidas representan, en un código binario, el índice de la entrada con valor 1. Si hay más de una entrada con el valor 1, la salida está indefinida. Un ejemplo de utilización es en los teclados donde al pulsar una tecla se activa una de las entradas de un codificador que se encarga de codificar la tecla pulsada.

El módulo también tiene una entrada H de habilitación, de forma que cuando $H = 0$ todas las salidas del decodificador valen 0 y una salida A que si vale 1 quiere decir que el codificador está activo. Esta salida indica que hay una entrada con valor 1 ("activa"). La entrada H y la salida A se emplean cuando se desea diseñar una red de módulos codificadores con un mayor número de entradas o generar una interrupción.

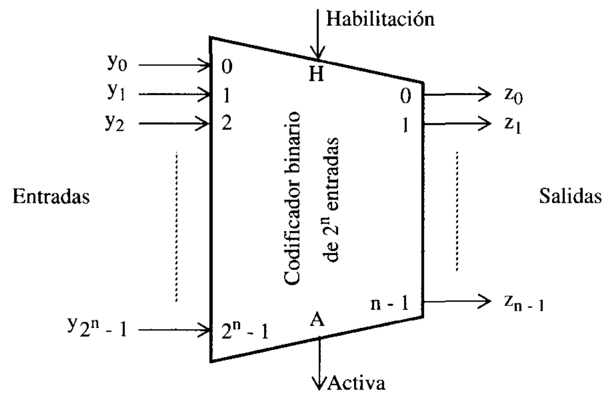


Figura A.2: Diagrama funcional de un codificador binario de 2^n entradas

Una descripción de alto nivel de un codificador binario de 2^n entradas es:

$z = i$ si $y_i = 1$ e $y_k = 0 \ \forall k \neq i$ y $H = 1$

$z = 0$ en cualquier otro caso

$A = 1$ si algún $y_i = 1$ y $H = 1$

$A = 0$ en cualquier otro caso

donde $z = \sum_{j=0}^{n-1} z_j 2^j$, y tal que $0 \leq i,\ k \leq 2^n - 1$.

Las expresiones lógicas que representan un codificador binario son:

$$z_i = H \sum y_k$$

donde y_k se incluye en la suma si el bit i-ésimo de la representación binaria de k es 1, y

$$A = H \sum_{i=0}^{2^n-1} y_i$$

Por ejemplo, las expresiones lógicas que representan un codificador binario de 4 entradas son:

$$z_0 = H\,(y_1 + y_3)$$
$$z_1 = H\,(y_2 + y_3)$$
$$A = H\,(y_0 + y_1 + y_2 + y_3)$$

En la Figura A.3 se muestra la realización con puertas lógicas de un codificador binario de 4 entradas.

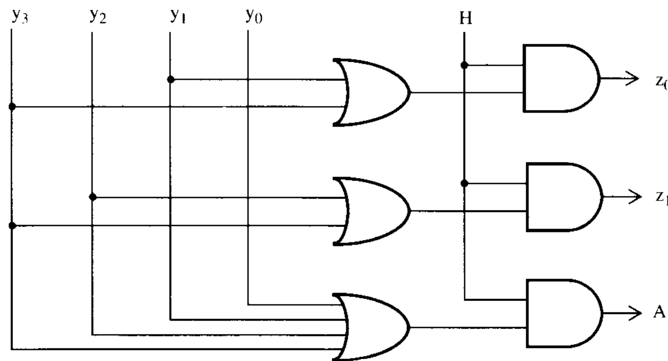


Figura A.3: Realización de un codificador binario de 4 entradas

Los codificadores se utilizan siempre que se desea identificar la ocurrencia de un suceso de un conjunto de posibles sucesos disjuntos, es decir, que no se pueden presentar simultáneamente. Cada suceso se identifica unívocamente mediante un número entero. Hay codificadores para otros códigos distintos del binario (BCD, Gray, etc).

A.2.1 Codificadores con prioridad

Los codificadores que se han considerado anteriormente presentan la limitación de que en cualquier instante de tiempo solamente una entrada puede tener el valor *1*. Como no hay nada que impida el que se activen

simultáneamente varias líneas de entrada de un codificador, los circuitos codificadores se diseñan normalmente para que respondan a una sola señal de entrada activa. Esto se hace asignando prioridades a las entradas, de forma que en cada instante sólo se genera el código correspondiente a la entrada activa que tenga la mayor prioridad, se habla entonces de *codificadores con prioridad.* El orden de la prioridad está fijado en el codificador y para un codificador de 2^n entradas se suele asignar a y_0 la menor prioridad y a y_{2^n-1} la mayor.

Para diseñar codificadores con prioridad de un número de entradas elevado se necesitan una entrada de habilitación H_e y dos salidas H_s y A. Una descripción de alto nivel de un codificador con prioridad de 2^n entradas es:

$$z = i \quad \text{si} \quad y_i = 1 \quad \text{e} \quad y_k = 0 \ \forall\, k \neq i \quad \text{y} \quad H_e = 1$$
$$z = 0 \quad \text{en cualquier otro caso}$$

donde $z = \sum_{j=0}^{n-1} z_j 2^j$, y tal que $0 \leq i,\ k \leq 2^n - 1$.

$$H_s = i \quad \text{si} \ \forall\, i \quad y_i = 1 \quad \text{y} \quad H_e = 1$$
$$H_s = 0 \quad \text{en cualquier otro caso}$$

y

$$A = 1 \quad \text{si} \ \exists \ \text{al menos un} \ y_i = 1 \quad \text{y} \quad H_e = 1$$
$$A = 0 \quad \text{en cualquier otro caso}$$

Los codificadores con prioridad se utilizan para seleccionar un suceso de un conjunto que pueden ocurrir simultáneamente. A este suceso lo representa por un número entero que lo codifica de forma unívoca. En el apartado 3.4.5 del tema 3 en el que se trata la *identificación de la fuente de interrupción mediante hardware paralelo* se presentó una aplicación del codificador con prioridad (ver Figura 3.21). Tal como se muestra en la Figura A.4, un codificador con prioridad se puede realizar empleando dos subsistemas. El primer subsistema corresponde a un circuito de resolución de la prioridad, cuyo objetivo es eliminar todos los 1's excepto el de mayor prioridad; el segundo es un codificador binario del tipo visto en el apartado anterior.

El circuito de resolución de prioridad tiene 2^n entradas (x_{2^n-1} , ..., x_0) y 2^n salidas (y_{2^n-1} , ..., y_0). Su descripción de alto nivel es:

$$y_i = i \quad \text{si} \quad x_i = 1 \quad \text{y} \quad x_k = 0,\ \forall\, k > i$$
$$y_i = 0 \quad \text{en cualquier otro caso}$$

Esto corresponde a la siguiente expresión lógica:

$$y_i = \overline{x}_{2^n-1} \ \ldots\ldots. \ \overline{x}_{i+1}\, x_i$$

donde $\overline{x}_j$ representa el complemento de x_j. En la Figura A.5 se muestra la realización mediante puertas AND de un circuito de 4 entradas para la resolución de prioridad.

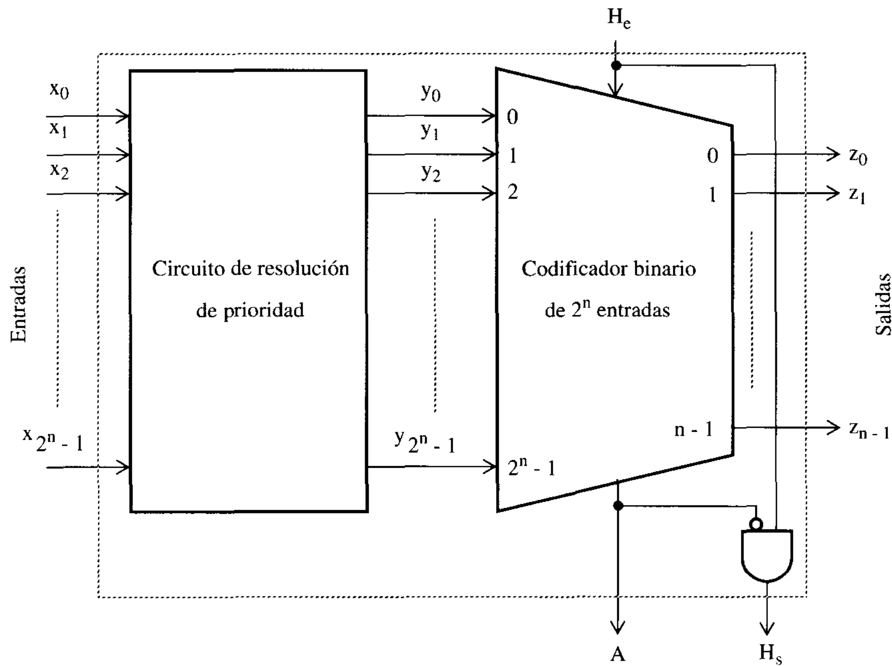


Figura A.4: Diagrama de bloques de un codificador con prioridad

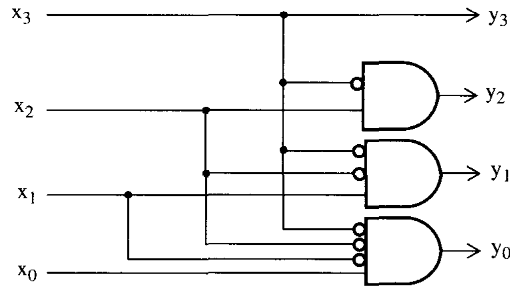
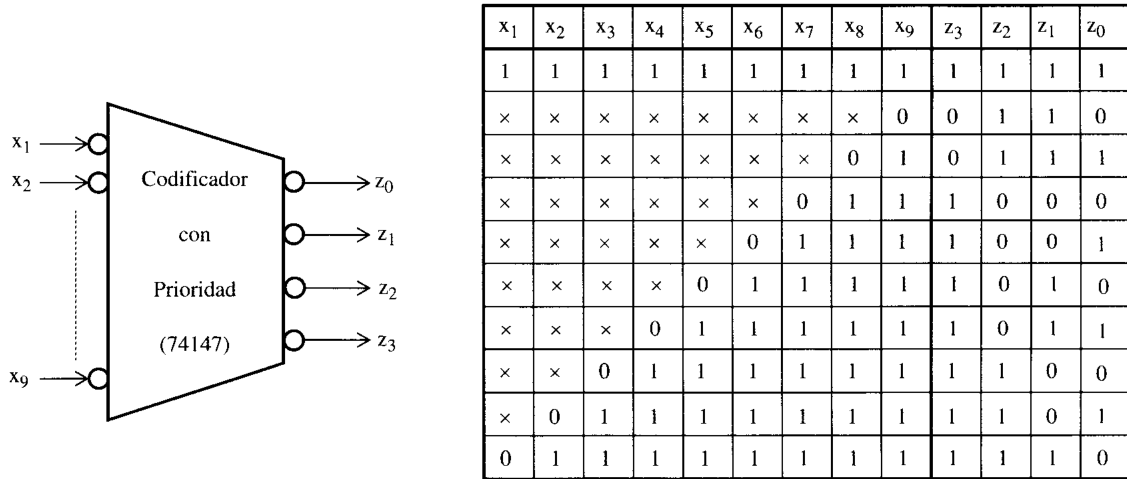


Figura A.5: Circuito paralelo de 4 entradas de resolución de prioridad

En la Figura A.6 se muestra el circuito integrado *74147* y su tabla de verdad, se trata de un codificador con prioridad, con 9 entradas (x_1, x_2, ..., x_9) y 4 salidas (z_0, z_1, z_2, z_3). El símbolo × corresponde a que el valor de la entrada puede ser 0 ó 1.

Las entradas son activas en baja y con la prioridad creciente con el número de entrada, es decir la entrada x_9 tiene la mayor prioridad. El código de salida corresponde al BCD con complementación en todas las salidas, se trata por tanto del circuito integrado inverso al *7445* (decodificador de BCD a decimal).



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	z_3	z_2	z_1	z_0
1	1	1	1	1	1	1	1	1	1	1	1	1
×	×	×	×	×	×	×	×	0	0	1	1	0
×	×	×	×	×	×	×	0	1	0	1	1	1
×	×	×	×	×	×	0	1	1	1	0	0	0
×	×	×	×	×	0	1	1	1	1	0	0	1
×	×	×	×	0	1	1	1	1	1	0	1	0
×	×	×	0	1	1	1	1	1	1	0	1	1
×	×	0	1	1	1	1	1	1	1	1	0	0
×	0	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0

Figura A.6: Diagrama de bloques y tabla de verdad del codificador 74147

Cuando ninguna entrada está activada el codificador *74147* lo interpreta como que la décima entrada, la entrada x_0 de menor prioridad, está activada y todas las salidas se ponen en alta, que corresponde al complemento del código BCD "0000". En la Figura A.7 se muestra la realización mediante puertas lógicas del circuito integrado *74147*.

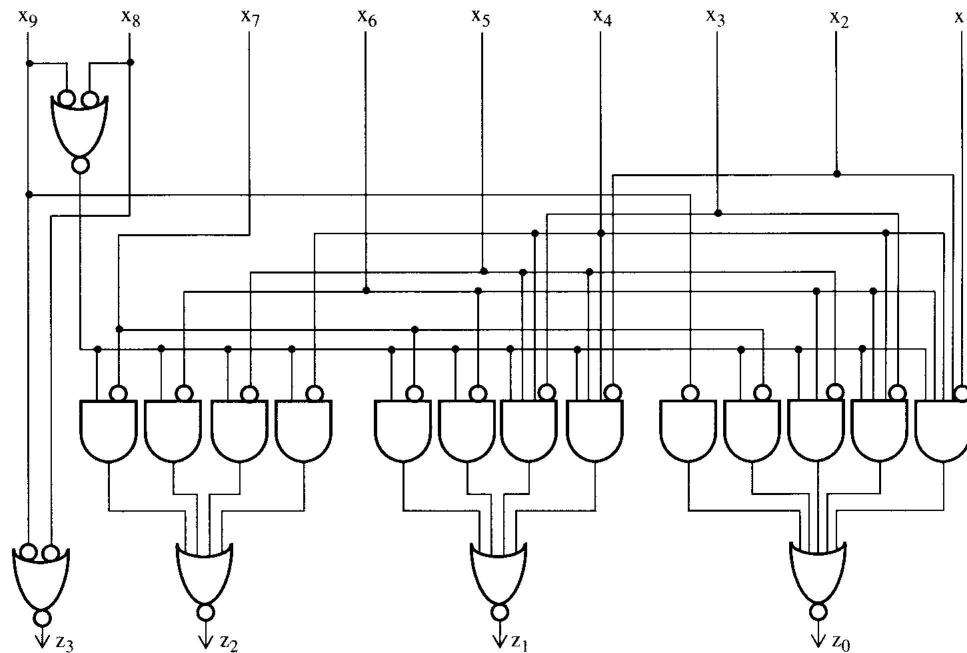


Figura A.7: Realización con puertas lógicas del codificador 74147

Sección A-3

Decodificadores

Los circuitos decodificadores son módulos combinacionales que *transforman los datos de un código numérico determinado (código de n bits) al código con un formato conocido como* 1 *entre* 2^n.

Estos módulos se utilizan fundamentalmente en el direccionamiento de los circuitos de memorias y de dispositivos de E/S. En esta aplicación el objetivo es seleccionar una palabra de entre 2^n palabras de memoria, o un dispositivo de entre 2^n dispositivos de E/S en operaciones de lectura o escritura. Para ello cada salida del decodificador se debe conectar a la entrada de control correspondiente de cada dispositivo o posición de memoria. En general un *decodificador binario de n entradas* (ver Figura A.8) es un sistema combinacional que posee n entradas $y = (y_{n-1}, ..., y_1, y_0)$ y 2^n salidas $z = (z_{2^n-1}, ..., z_1, z_0)$.

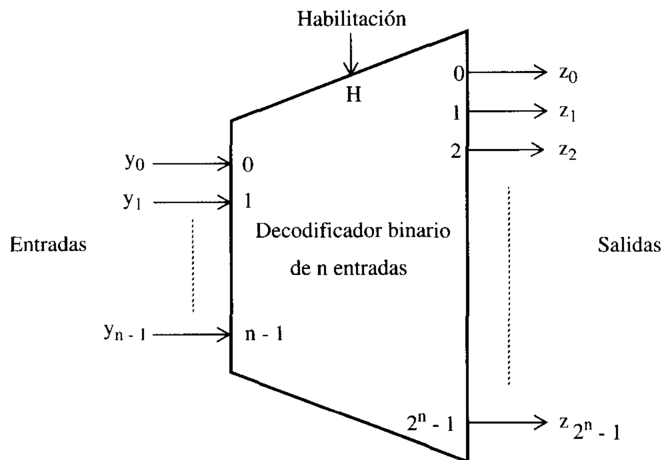


Figura A.8: Diagrama funcional de un decodificador binario de n entradas

El vector de entrada y se puede considerar que representa un nº entero comprendido entre 0 y 2^n - 1. Cuando la entrada es el número i, la salida y_i vale 1 y todas las otras salidas están a 0.

Para facilitar su uso en el diseño de circuitos el decodificador incorpora una entrada de habilitación del módulo H. Cuando H = 0 todas las salidas del decodificador valen 0. Una descripción de alto nivel de un decodificador binario de n entradas es:

$z_i = 1$ si $y = i$ y $H = 1$

$z_i = 0$ en cualquier otro caso

donde $y = \sum_{j=0}^{n-1} y_j 2^j$.

La expresión lógica que representa un decodificador binario es:

$$z_i = H\, m_i(y) \quad i = 0, 1, \ldots, 2^n - 1$$

donde $m_i(y)$ es el minterm i-ésimo de las n variables y.

En la Figura A.9 se da la realización con puertas lógicas de un decodificador binario de 2 entradas.

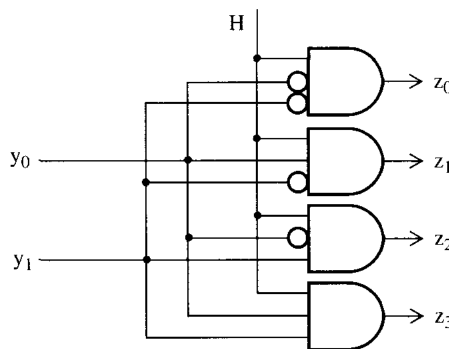


Figura A.9: Realización de un decodificador binario de 2 entradas

Los decodificadores se utilizan siempre que un conjunto de valores que se han codificado previamente tienen posteriormente que separarse (es decir decodificarse). Un ejemplo es el código de operación de un computador. Este código, que forma parte de cada instrucción, se tiene que decodificar para determinar la operación que el computador tiene que ejecutar (ver Figura A.10).

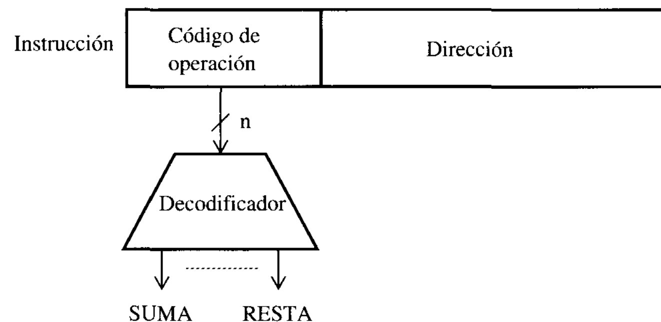


Figura A.10: Decodificación de instrucciones

En la Figura A.11 se muestra un decodificador de 1 entre 10 (se trata del circuito integrado *7445* que es un decodificador de código BCD a decimal). Tal como se indica mediante los círculos inversores, las señales de salida de este decodificador son activas a nivel bajo (lógica negativa). Así, si la entrada al decodificador es $y_3y_2y_1y_0 = 0011$ (que es en binario el número 3) su salida será $z_9z_8z_7z_6z_5z_4z_3z_2z_1z_0 = 1111110111$. El 0 en la línea de salida z_3 indica que se ha decodificado la entrada que corresponde al número 3.

Este circuito no tiene línea de habilitación, sin embargo resulta inhabilitado para cualquiera de las seis combinaciones de entradas que no representan un dígito BCD válido.

En la Tabla A.1 se da la tabla de verdad del circuito *7445*. Se observa que para las configuraciones de entrada que corresponden a los números decimales que van del 10 al 15 todas las salidas del decodificador están a 0 indicando así que no está habilitado.

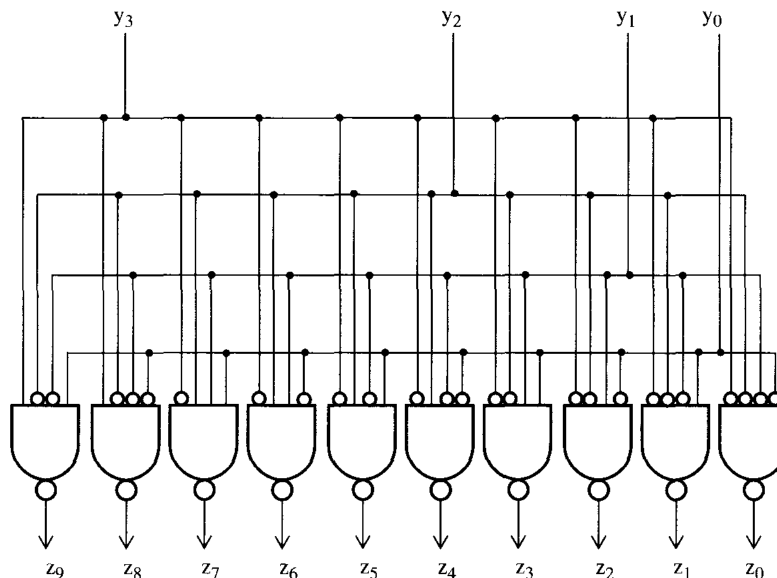


Figura A.11: Circuito lógico del decodificador 7445

Un decodificador de *n* entradas y una puerta OR realiza cualquier función de *n* variables. La salida *i*-ésima del decodificador corresponde al minterm $m_i(y)$, esto permite obtener la expresión canónica de la función como suma de minterms, conectando directamente a una puerta OR las salidas del decodificador que corresponden a los minterms de la función que se desea sintetizar. Con el mismo decodificador se pueden generar más de una función de las mismas variables.

Sean por ejemplo las siguientes funciones lógicas de tres variables expresadas en forma canónica como suma de sus minterms:

$$f_1(y_3, y_2, y_1) = \sum m(0, 3, 4)$$

$$f_2(y_3, y_2, y_1) = \sum m(1, 2)$$

$$f_3(y_3, y_2, y_1) = \sum m(3, 5, 7)$$

y_3	y_2	y_1	y_0	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Tabla A.1: Tabla de verdad del decodificador 7445

En la Figura A.12 se muestra la síntesis de estas funciones utilizando un decodificador binario de 3 entradas y 3 puertas OR.

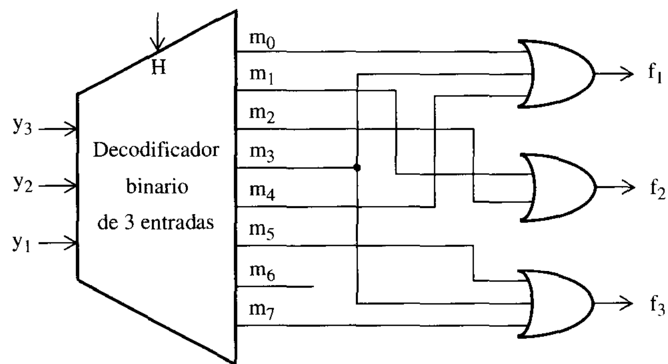


Figura A.12: Síntesis de funciones lógicas utilizando un decodificador

Los decodificadores se pueden interconectar entre sí para obtener módulos de mayor número de líneas de entrada y de salida (propiedad de ampliación de los módulos). La manera de realizarlo es mediante una estructura arborescente de decodificadores, tal como se muestra en la Figura A-11 en el caso de un decodificador de 4 entradas construido a partir de decodificadores de 2 entradas.

El árbol, de dos niveles, tiene un decodificador en el primer nivel y cuatro en el segundo. El vector de entrada $y = (y_3, y_2, y_1, y_0)$ se divide en dos subvectores, con (y_3, y_2) decodificados en el primer nivel e (y_1, y_0) en el segundo. Las 16 salidas se particionan en 4 grupos con 4 salidas en cada uno. La decodificación del subvector (y_3, y_2) habilita a uno de los grupos y la decodificación de (y_1, y_0) produce la salida correspondiente en el grupo que está activado.

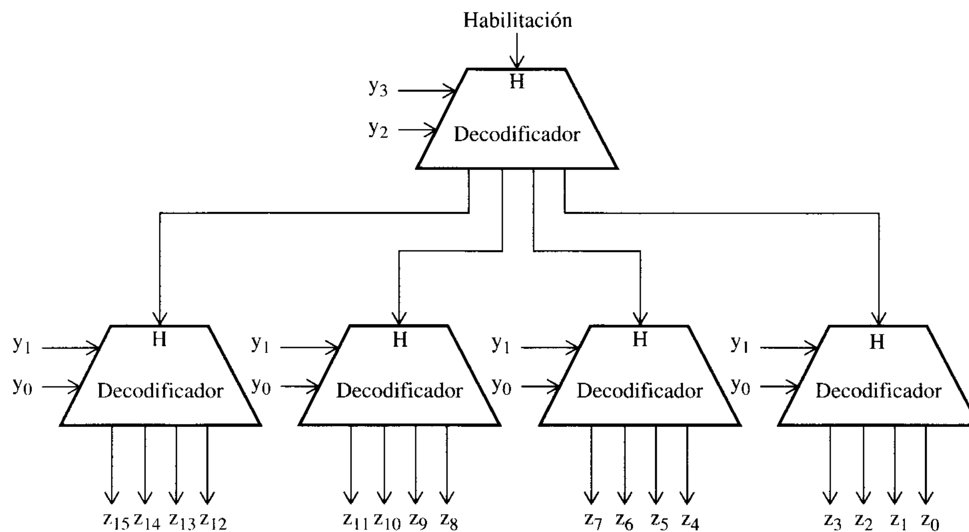


Figura A.13: Decodificador de 4 entradas obtenido mediante decodificadores de 2 entradas

Sección A-4

Multiplexores

La función primaria de un multiplexor (que en ocasiones se abrevia MUX) como componente combinacional en un computador es *hacer que palabras con orígenes diferentes se puedan transferir a un mismo destino.* Permite seleccionar una de entre varias señales de entrada y enviarla a la salida, por esta razón a un multiplexor también se le conoce como *circuito selector de datos.*

En un instante dado, sólo se transfiere una palabra a través del multiplexor, es decir, el bus de salida se comparte por todas las fuentes de datos. A esta manera de compartir se la conoce como *multiplexado en el tiempo.* El número de *líneas de selección o control* es tal que sus posibles combinaciones igualan al número de *entradas de datos*, es decir, si el circuito multiplexor admite 8 líneas de entrada, debe tener 3 líneas de selección, pues $2^3 = 8$ son todas sus posibles combinaciones. En general, para n líneas de selección o control (c_i, $i = 0, 1, ..., n-1$) hay disponibles hasta 2^n líneas de entradas de datos (y_i, $i = 0, 1, ..., 2^n - 1$).

El multiplexor dispone de una *línea de habilitación* (*H*) que cuando está desactivada (a 0 ó a 1 según se trate), la salida del multiplexor *z* toma un valor fijo independientemente del valor de las entradas de selección y de datos (ver Figura A.14). En algunos casos si el multiplexor no está habilitado su salida se encuentra en un estado de alta impedancia (ver apartado 1.4.5).

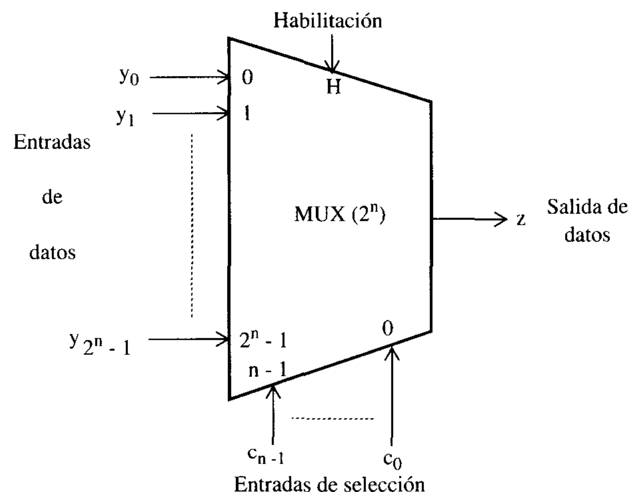


Figura A.14: Diagrama funcional de un multiplexor de 2^n entradas de datos

Los valores de las variables de control c se interpretan como la representación binaria de un número entero comprendido entre 0 y 2^n -1. La salida z del multiplexor corresponde al dato de entrada y_i si c representa el entero i. Una descripción de alto nivel del multiplexor es:

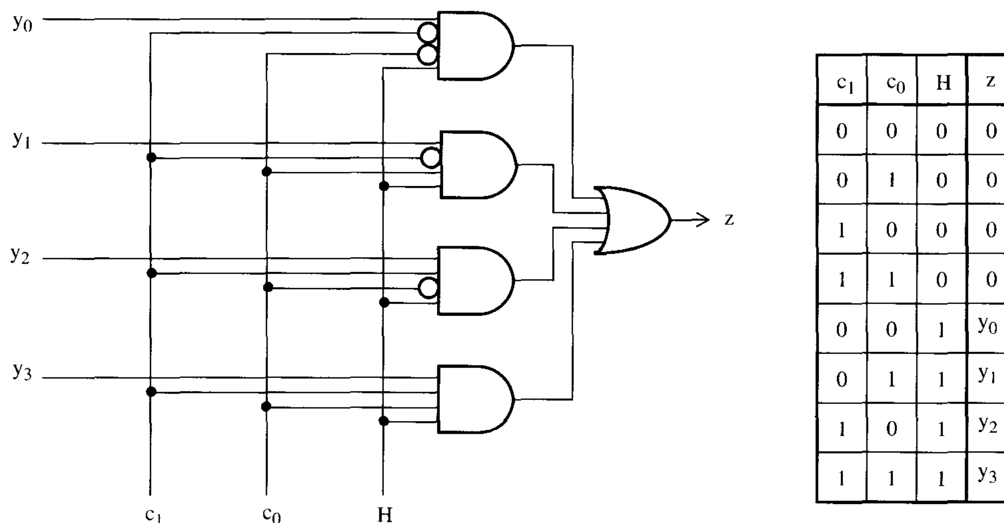
$z = y_c$	si el MUX está habilitado	H = 1
$z = 0$	si el MUX está deshabilitado	H = 0

donde

$$c = \sum_{j=0}^{n-1} c_j 2^j$$

En la Figura A.15 se muestra la realización mediante inversores, puertas AND y una puerta OR de un multiplexor con 4 entradas de datos de un bit cada una (y_3, y_2, y_1 e y_0) y que por tanto requiere un bus de control de 2 bits (c_1, c_0). También se indica la tabla de verdad del multiplexor.

Así si el multiplexor está habilitado (H = 1) y las entradas de selección o control valen $c_1\, c_0$ = 0 1 (que es la representación binaria del número entero c = 1), de acuerdo con la descripción de alto nivel del multiplexor su salida será $z = y_1$.



c1	c0	H	z
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	y0
0	1	1	y1
1	0	1	y2
1	1	1	y3

Figura A.15: Realización de un multiplexor de 4 entradas de datos

El multiplexor, si no se tienen en cuenta los inversores, es un ejemplo típico de circuito combinacional de dos niveles (AND-OR). Siempre se puede obtener un multiplexor más grande (mayor número de entradas de datos) interconectando adecuadamente varios multiplexores más pequeños (propiedad de ampliación de los módulos). Por ejemplo, en la Figura A.16 se presenta el diseño de un multiplexor de 4 entradas de datos (MUX(4)) utilizando 3 multiplexores de 2 entradas de datos (MUX(2)).

Esta propiedad de construcción mediante una estructura arborescente de multiplexores de más entradas, a partir de multiplexores con menor número de entradas (análoga al caso de los decodificadores) es totalmente general.

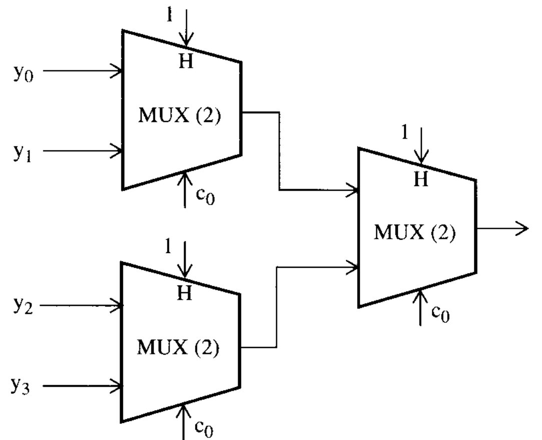


Figura A.16: MUX (4) obtenido mediante 3 MUX (2)

Además de su utilidad para seleccionar o transferir datos, otra característica muy interesante de los multiplexores es que pueden servir también como *módulos universales para la generación de funciones lógicas*. En el caso más simple, se elige un multiplexor de tantas líneas de selección como variables tenga la función lógica. Cada variable se conecta a una línea de selección y las entradas de datos del multiplexor se ponen a un valor fijo (0 ó 1) correspondiente a cada uno de los valores de la tabla de verdad de la función lógica que se desea generar. Por ejemplo, en la Figura A.17 se muestra la realización de la función OR-exclusiva de tres variables con un multiplexor de 8 entradas de datos.

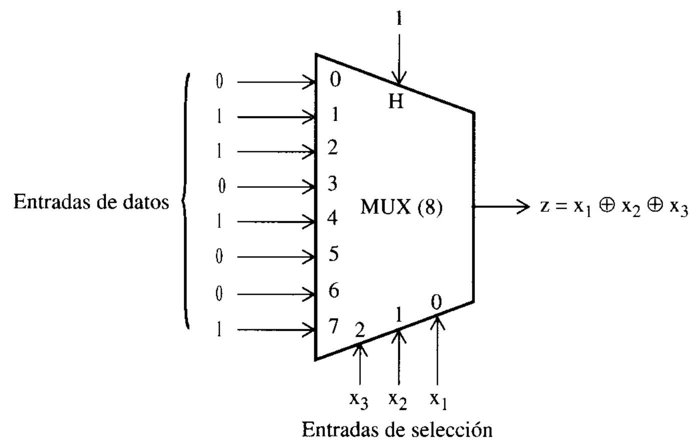


Figura A.17: Realización de la función $x_1 \oplus x_2 \oplus x_3$ con un MUX (8)

Sección A-5

Demultiplexores

Un demultiplexor es un circuito combinacional, con n entradas de selección o control $c = (c_{n-1}, ..., c_1, c_0)$, una entrada de datos y, una entrada de habilitación del módulo H y 2^n salidas de datos $z = (z_{2^n-1}, ..., z_1, z_0)$ (ver Figura A.18). La entrada se conecta con la salida seleccionada por las variables de control ó selección. Todas las otras salidas están a 0.

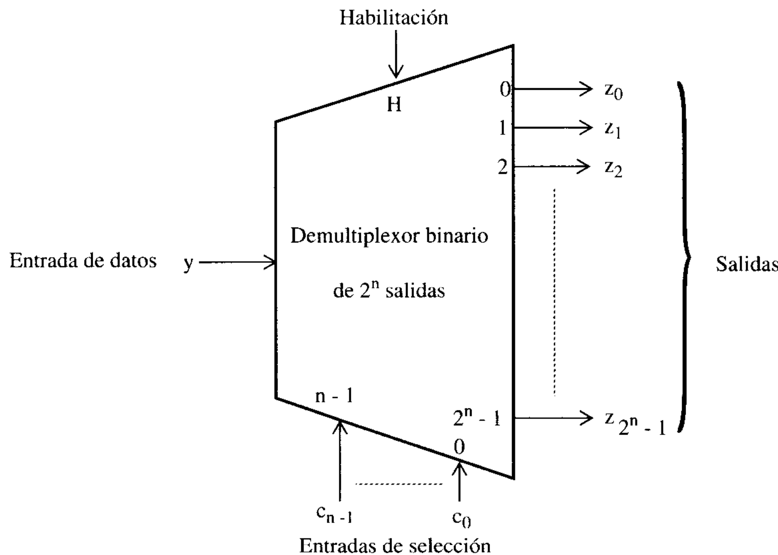


Figura A.18: Diagrama funcional de un demultiplexor de n entradas y 2^n salidas

Los demultiplexores se pueden emplear para transferir datos desde una fuente común a uno de entre varios posibles destinos bajo el control de las entradas de selección. Por este motivo se suele denominar también a los demultiplexores como distribuidores. Esta operación es la inversa de la de conducción de datos realizada por un multiplexor.

Una descripción de alto nivel de un demultiplexor de n entradas es:

$$z_i = y \quad \text{si} \quad i = c \quad \text{y} \quad H = 1$$

$$z_i = 0 \quad \text{si} \quad i \neq c \quad \text{o} \quad H = 0$$

donde $c = \sum_{j=0}^{n-1} c_j 2^j$, y tal que $0 \le i \le 2^n - 1$.

La expresión lógica que representa las salidas del demultiplexor es:

$$z_i = H \text{ y } m_i(c) \qquad i = 0, 1, \ldots, 2^n - 1$$

donde $m_i(c)$ es el minterm i-ésimo de las n variables de selección c. Estas expresiones son similares a las del decodificador con la única diferencia de la entrada de datos adicional y. En consecuencia, un demultiplexor se puede utilizar como un decodificador. Es posible también emplear de forma inversa un decodificador como un demultiplexor, a condición de que la entrada de habilitación H del decodificador se use como entrada de datos del demultiplexor. En este último caso el demultiplexor resultante no dispondría de entrada de habilitación.

Sección A-6

Dispositivos lógicos programables

Existen elementos lógicos de propósito general que han sido desarrollados para realizar cualquier función combinacional, como por ejemplo determinadas partes de la unidad de control de un computador no microprogramable. En general estos componentes han de cumplir los dos requisitos siguientes:

1) Cada componente debe tener una estructura subyacente uniforme, que permita una producción masiva utilizando las modernas técnicas de fabricación de circuitos integrados.

2) Es posible alterar la estructura básica del componente, o bien durante el propio proceso de fabricación o directamente por parte del usuario, para realizar la función lógica impuesta por el diseño.

Ejemplos de este tipo de circuitos son:

a) ROM (Read Only Memory).

b) PLA (Programmable Logic Array).

c) PAL (Programmable Array Logic).

La utilización de la ROM como elemento de propósito general para la síntesis de funciones lógicas ya se comentó en el tema 4 (ver apartado 4.2.6 y problema 4.19) y por ese motivo no se considera. Simplemente conviene recordar que con una ROM de 2^n palabras × m bits se pueden realizar simultáneamente m funciones lógicas de n variables.

A.6.1 Arrays lógicos programables (PLA)

Un *array lógico programable* o PLA (Programmable Logic Array) es un conjunto de puertas lógicas sin un propósito definido a priori, que se puede configurar para la realización de una determinada función. Esta particularización se lleva a cabo normalmente mediante la fusión de determinados fusibles que interconectan las puertas lógicas del PLA. Un PLA contiene los elementos siguientes:

a) Un conjunto de líneas de entrada (del orden de 8-16)

b) Un conjunto de líneas de salida (del orden de 6-8)

c) Un conjunto de puertas AND (del orden de 48-96)

d) Un conjunto de puertas OR (una por cada línea de salida)

e) Un conjunto de inversores sobre cada línea de entrada y salida

f) Dos matrices de conexiones alterables (matriz AND y matriz OR)

En la Figura A.19 se muestra un diagrama de bloques de la estructura de un PLA.

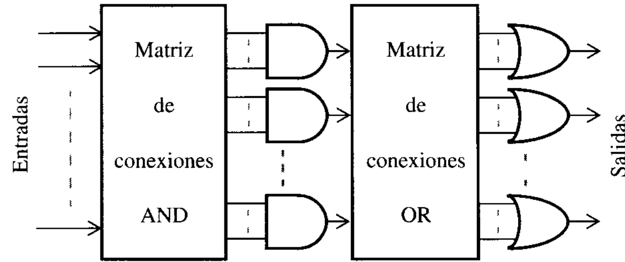


Figura A.19: Estructura funcional de un PLA

A nivel lógico un PLA permite una realización NOT-AND-OR de un conjunto de funciones. Específicamente un PLA $n \times m$ permite sintetizar m funciones de n variables. Consta de un conjunto de n *inversores* que proporcionan el valor complementario de las variables de entrada, un conjunto de k *puertas AND* que producen los términos productos y un conjunto de m *puertas OR* que generan los términos sumas.

Las puertas AND y OR se organizan en dos estructuras regulares denominadas arrays AND y OR respectivamente. Las conexiones de las líneas de entrada (o de sus complementos) con las entradas de las puertas AND se programan. Análogamente, también se programan las conexiones entre las salidas de las puertas AND y las entradas a las puertas OR. En la Figura A.20 se ilustra la estructura lógica de un PLA. Se observa que un PLA $n \times m$ que contiene k puertas AND puede realizar un conjunto de m funciones de conmutación de n variables, si el número de términos productos en su representación de suma de productos no es mayor que k.

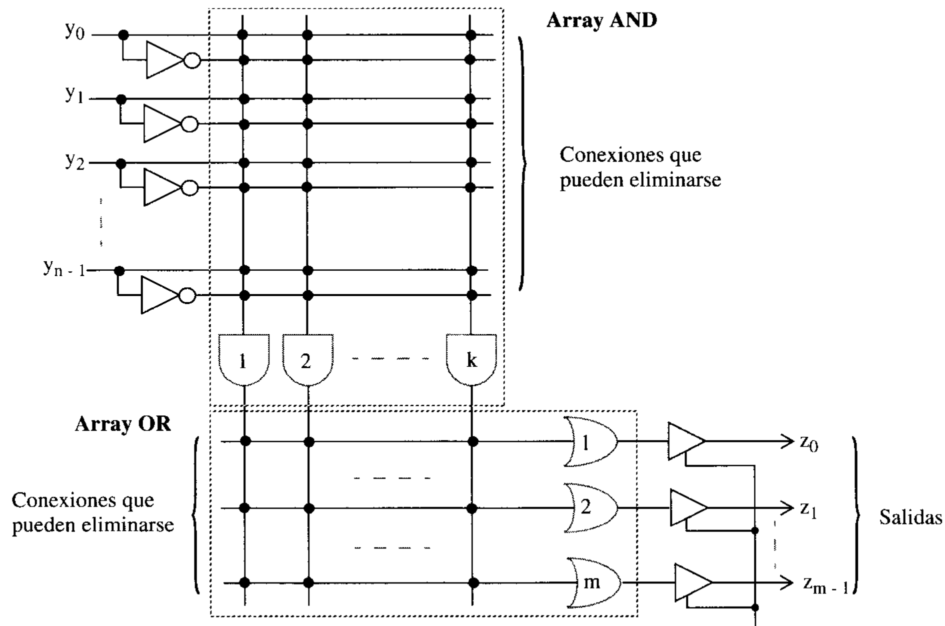


Figura A.20: Estructura lógica de un PLA

En la Figura A.21 se muestran las cuatro condiciones que hay para cada variable de entrada.

1) El estado inicial no programado con todas las conexiones intactas tanto de la variable como de su complemento.
2) La variable o su complemento se seleccionan eliminando la conexión correspondiente.
3) Si se eliminan ambas conexiones la variable no aparece en el término producto.
4) Si se eliminan todas las conexiones de una puerta AND, se genera a su salida un 1 lógico.

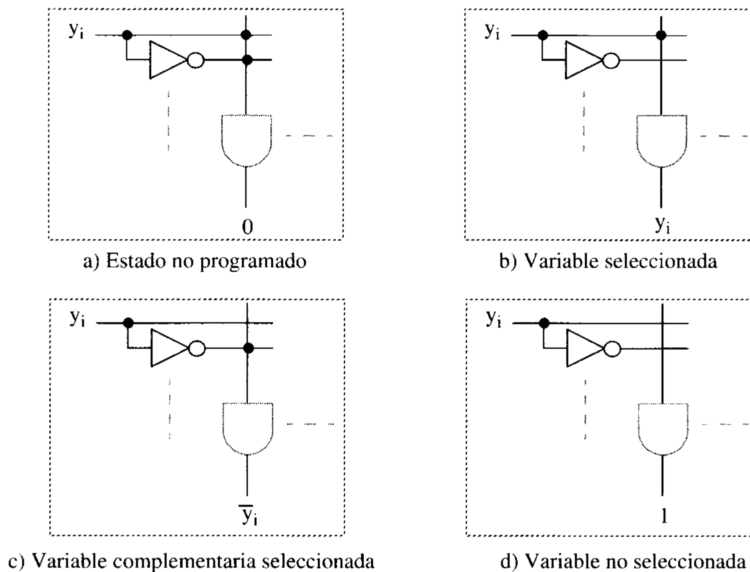


Figura A.21: Estados de conexión en el array AND de un PLA

En general no se debería utilizar la condición de estado no programado ya que esto llevaría a que de forma permanente el término producto sería 0 ($py_i\overline{y}_i = 0$, donde p representa las otras variables que forman el término producto). Una excepción a esta regla es cuando no se desea utilizar una puerta AND en cuyo caso hay que emplear el estado no programado que deja intacta todas las conexiones y hace que el término producto correspondiente sea 0.

Conviene señalar las diferencias, que desde el punto de vista de realización de funciones lógicas presentan una ROM y un PLA.

1) En general un PLA produce una síntesis más compacta que una ROM, ya que en lugar de almacenar la tabla de verdad completa de la función, realiza una suma de productos mínima. La reducción en tamaño resulta considerable si el número de productos que se necesitan es mucho menor que el máximo de 2^n.
2) La realización con un PLA está limitada a un conjunto de funciones, que pueden representarse por un conjunto de sumas de productos que no tiene mas de k términos productos. Esta restricción no existe en el caso de la ROM que incorpora todos minterms de la función.

3) La programación de un PLA es más difícil que el de una ROM. Esto se debe a que en un PLA hay que diseñar una estructura AND-OR de dos niveles.

A.6.2 Ejemplo: Síntesis de funciones lógicas con un PLA

Se desea implementar con un PLA el siguiente conjunto de funciones booleanas:

$$f_1(y_3, y_2, y_1, y_0) = y_0\overline{y}_1 y_2$$

$$f_2(y_3, y_2, y_1, y_0) = y_0 y_2 + y_1 y_2$$

$$f_3(y_3, y_2, y_1, y_0) = \overline{y}_0\overline{y}_1 + y_0\overline{y}_2\overline{y}_3 + y_0\overline{y}_1 y_2$$

En la Figura A.22 se muestran las conexiones de un PLA para estas funciones. Conviene observar que cuando existe un término producto que es común a varias funciones se puede compartir de la misma forma que en una implementación normal AND-OR en dos niveles (la salida de la puerta AND número 5 forma el término producto $y_0\overline{y}_1 y_2$ que es compartido por las funciones f_1 y f_3). La simplificación lógica puede ser útil también con una implementación PLA debido al número limitado de puertas que se dispone.

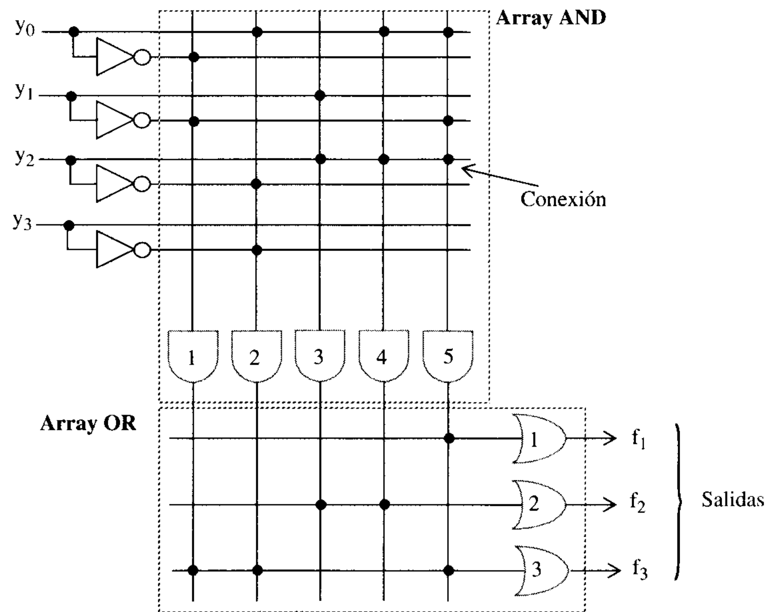


Figura A.22: Síntesis de las funciones f_1, f_2 y f_3 del ejemplo con un PLA

A.6.3 Arrays lógicos programables (PAL)

El PLA descrito en la sección anterior tiene programable tanto las conexiones del array AND como del array OR. La compañía Advanced Micro Devices (AMD) fabrica una familia de dispositivos lógicos programables que solo permita alterar las conexiones del array AND. Denominan a este tipo de módulo como PAL (Programmable Array Logic), término que es ahora ampliamente aceptado para nombrar a esta clase de dispositivos.

En un PAL la salida de cada puerta AND va directamente a la entrada de una puerta OR. Típicamente, se conectan 8 puertas AND a las entradas de cada puerta OR (de 8 entradas). Un motivo para esta restricción es aumentar la velocidad de operación ya que las conexiones programables incurren en un retardo mayor que una conexión cableada directamente. En la Figura A.23 se muestra un ejemplo de un circuito combinacional PAL, el *AMD PAL16L8*. La letra *L* (Low) en la identificación del módulo indica que las salidas se invierten .

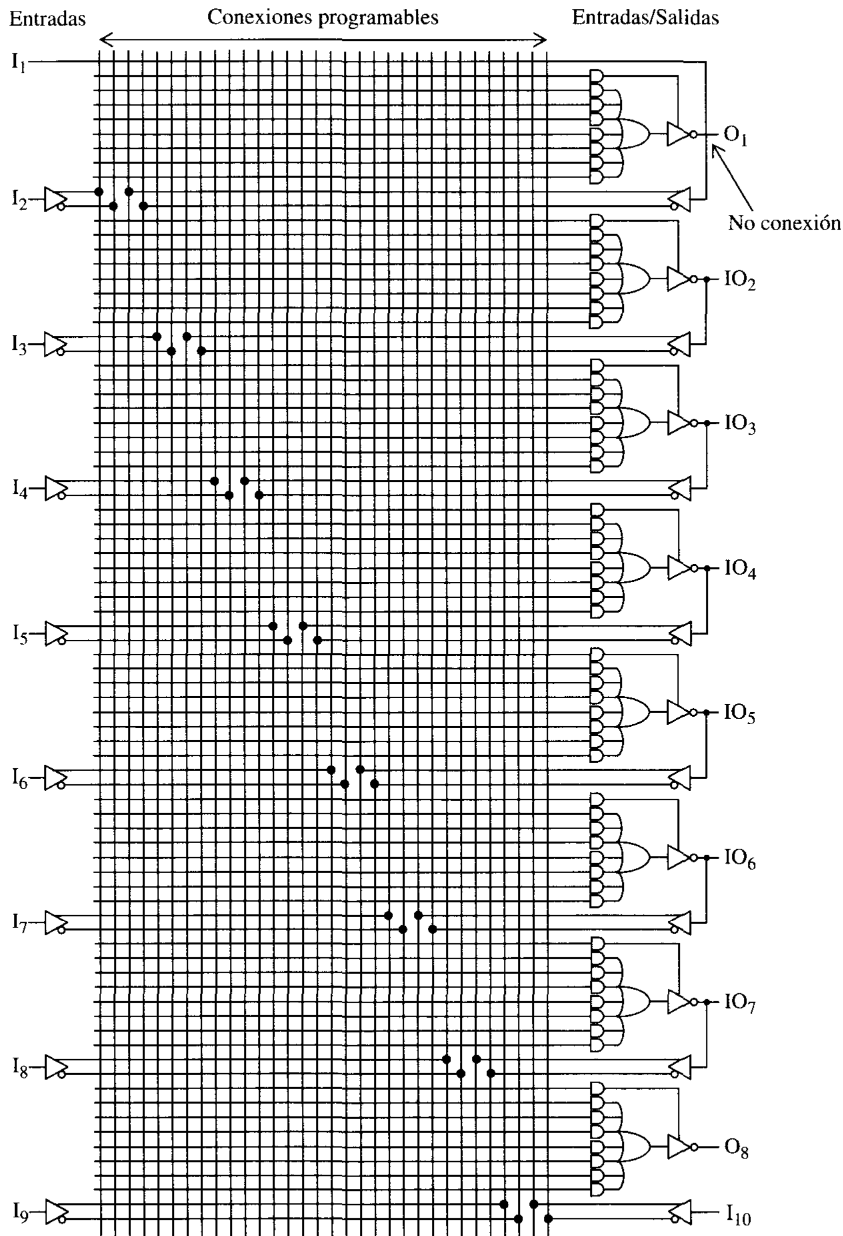


Figura A.23: Array lógico programable tipo PAL (AMD PAL16L8)

Como en el caso del PLA, inicialmente todas las conexiones están activas. Aunque en la Figura A.23 por claridad no se han dibujado. En este dispositivo hay un máximo de 16 entradas y 8 salidas, aunque algunas salidas se comparten con las entradas debido a limitaciones en el número de terminales del circuito integrado. Las conexiones de entrada/salida permiten establecer caminos de realimentación. Las salidas del módulo van dotadas de puertas triestado que se pueden activar mediante un término producto de las entradas y variables de realimentación. Las conexiones de entrada/salida compartidas se pueden emplear como entradas cuando se desactivan las salidas.

En la Figura A.23, hay 10 entradas independientes, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9 e I_{10}, cada una de ellas con posibilidad de selección de la variable o de su complemento. Hay 8 salidas, 6 de las cuales también sirven como entradas IO_2, IO_3, IO_4, IO_5, IO_6, IO_7 y 2 se dedican a salidas O_1 y O_8. Este módulo puede generar 8 sumas de funciones producto, cada una con 7 términos producto y cada término producto con 10 variables. Alternativamente se pueden generar 2 sumas de términos productos cada una con 7 términos productos y hasta 16 variables (6 de las cuales provienen de los terminales compartidos de entrada/salida). Son posibles otras configuraciones aunque siempre sujetas a las limitaciones que imponen la compartición de líneas de entrada/salida. En este caso no hay ninguna ventaja en tener términos productos compartidos entre las expresiones de suma de productos ya que cada término se genera separadamente para cada puerta OR. Como en el *PAL16L8* se invierten las salidas, las funciones lógicas que implementan son en realidad del tipo AND-OR-NOT.

A.6.4 Ejemplo: Síntesis de funciones lógicas con un PAL

La Figura A.24 muestra como se implementaría la función $\bar{f} = \bar{a}bc + a\bar{b}c + \bar{a}\bar{c}$ en una salida de un *PAL16L8*. Se observa que las puertas AND que no se utilizan tienen intactas todas sus conexiones lo que fuerza un 0 a las entradas de estas puertas. Las salidas de las puertas AND estarán así en un 0 permanente y no afectan a la puerta OR correspondiente. La salida $\bar{f}$ se genera eliminando todas las conexiones a la entrada de la puerta AND que controla la puerta triestado de salida ya que se precisa un 1 para activar dicha puerta.

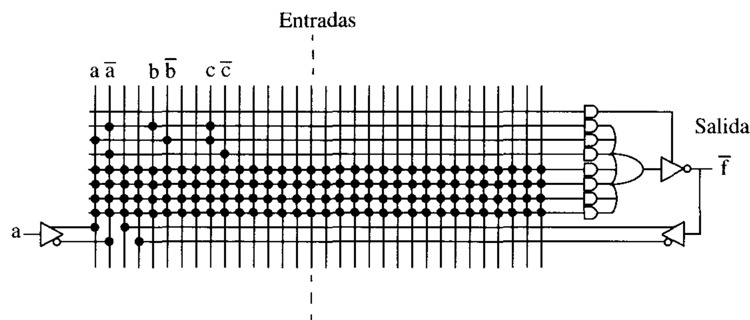


Figura A.24: Implementación de la función $\bar{f} = \bar{a}bc + a\bar{b}c + \bar{a}\bar{c}$ utilizando un PAL

Para permitir que la síntesis de una función lógica se pueda realizar en un PAL específico, una expresión suma de productos se puede implementar empleando más de dos niveles. Para hacerlo, tal como se muestra en la Figura A.25, se conecta una salida a una entrada.

Por ejemplo para obtener $a + b + c + d + e + f + g + h + i + j$, la primera etapa podría generar $x = a + b + c + d + e + f + g$ y la segunda etapa $x + h + i + j$. En la Figura A.25 no se incluyen, para una mayor claridad, las conexiones a las puertas AND no utilizadas.

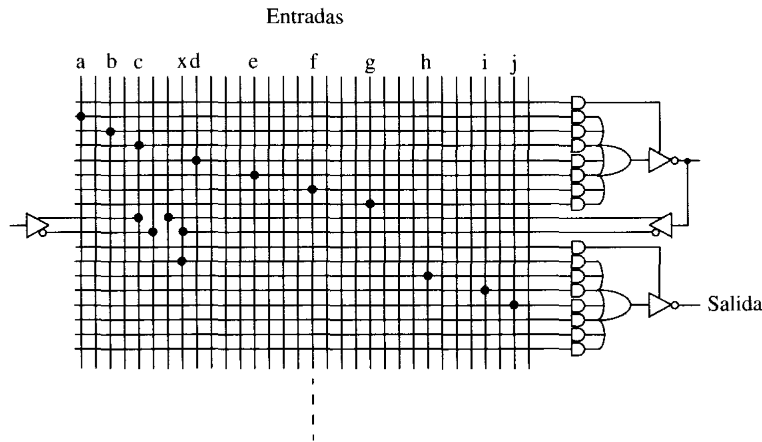


Figura A.25: Expresión multinivel de una suma de productos

El *AMD PAL16L8* descrito en la Figura A.23 proporciona la salida complementada (la *L* en el nombre indica que la salida es la complementaria). Existen PAL´s con salida sin complementar (emplean en el nombre la letra *H*). Si se dispone de un PAL con salida complementada y lo que se desea es implementar la función sin complementar la forma más sencilla de deducir la inversa de la función como una suma de términos productos es agrupar los 0´s de la función en un diagrama de Karnaugh en lugar de los 1´s.

En algunos dispositivos es posible programarlos de forma que pueden generar tanto la función sin complementar como complementada. Tal como se muestra en la Figura A.26, estos dispositivos incorporan un circuito programable que permite realizar ambas posibilidades.

Así, si se deja intacta la conexión programable se obtiene que *salida* = *entrada*, mientras que si se elimina dicha conexión se genera $salida = \overline{entrada}$. El circuito de la Figura A.26 se colocaría después de la puerta OR y antes del buffer triestado.

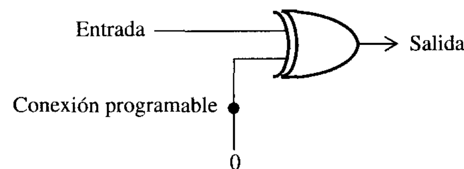


Figura A.26: Circuito de salida verdadera/inversa