TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores

CE 3201 — Taller de Diseño Digital

Documento de diseño

Autores:

Fabricio González Cerdas

Jian Zheng Wu

Profesor:

Luis Barboza Artavia

1. Propuestas de diseño Sumador

En esta sección se presentan dos métodos para implementar un sumador parametrizable de n bits. Ambas propuestas cumplen la misma función, pero difieren en la forma de abarcar los acarreos las ecuaciones lógicas y en la cantidad de compuertas utilizadas.

1.1. Ripple carry Adder

Un sumador es un bloque fundamental en el diseño digital, utilizado para realizar operaciones aritméticas. El objetivo de esta propuesta es presentar un diseño parametrizable que permita implementar un sumador de n bits a partir de la repetición de un sumador completo de 1 bit.

1.1.1. Bloque base: Sumador Completo (Full Adder)

El sumador completo recibe como entradas dos bits (A, B) y un acarreo de entrada (C_{in}), y produce como salidas el bit de suma (S) y un acarreo de salida (C_{out}). Donde sus ecuaciones booleanas son las siguientes:

$$\mathrm{S} = \mathrm{A} \oplus B \oplus C_{in}$$

$$\mathrm{C}_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

Donde su tabla de verdad es esta:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ahora si, para el sumador de n bits se propone interconectar n sumadores completos de 1 bit en una arquitectura tipo *ripple-carry adder*.

1.1.2. Ecuaciones generales

Sea $A = (A_{n-1}, A_{n-2}, \ldots, A_0)$ y $B = (B_{n-1}, B_{n-2}, \ldots, B_0)$. El cálculo se realiza de manera iterativa:

$$S_i = A_i \oplus B_i \oplus C_i \tag{1}$$
$$C_{i+1} = (A_i \cdot B_i) + (C_i \cdot (A_i \oplus B_i)) \tag{2}$$

donde el C inicial:

$$C_0 = C_{in}, \quad C_n = C_{out}.$$

1.1.3. Parametrización

El número de bits n se define como un *parámetro* o *genérico* en la descripción hardware, de modo que el mismo código puede instanciar un sumador de 4, 8, 16 o cualquier cantidad de bits sin cambios en la lógica básica.

1.2. Carry Lookahead Adder (CLA)

Un sumador también puede implementarse mediante una arquitectura de anticipación de acarreo (*carry lookahead adder*), la cual reduce el retardo total en comparación con el método *ripple-carry*. El objetivo es calcular los acarreos de forma paralela mediante expresiones booleanas, en lugar de depender de la propagación secuencial bit a bit.

1.2.1. Bloque base: Sumador Completo con señales de propagación y generación

El sumador completo puede describirse introduciendo dos señales auxiliares: la señal de propagación P_i y la señal de generación G_i, definidas como:

$$P_i = A_i \oplus B_i \tag{3}$$
$$G_i = A_i \cdot B_i \tag{4}$$

Con estas señales, las ecuaciones del sumador se expresan como:

$$S_i = P_i \oplus C_i \tag{5}$$
$$C_{i+1} = G_i + (P_i \cdot C_i) \tag{6}$$

Donde su tabla de verdad (mostrando explícitamente P_i y G_i) es:

A	B	C_{in}	$P = A \oplus B$	$G = A \cdot B$	S	C_{out}
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	1	0	1	0
0	1	1	1	0	0	1
1	0	0	1	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	1	0	1	1	1

1.2.2. Ecuaciones generales de acarreo

A partir de P_i y G_i, los acarreos pueden expresarse de manera directa sin necesidad de recorrer bit a bit:

$$C_1 = G_0 + (P_0 \cdot C_0) \tag{7}$$

$$C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot C_0) \tag{8}$$

$$C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot C_0) \tag{9}$$

$$\vdots \tag{10}$$

$$C_n = G_{n-1} + (P_{n-1} \cdot G_{n-2}) + \cdots + (P_{n-1} \cdot P_{n-2} \cdot \cdots \cdot P_0 \cdot C_0) \tag{11}$$

1.2.3. Ecuaciones generales de suma

Sea $A = (A_{n-1}, A_{n-2}, \ldots, A_0)$ y $B = (B_{n-1}, B_{n-2}, \ldots, B_0)$. El cálculo de las sumas se mantiene:

$$S_i = P_i \oplus C_i, \qquad \text{con } P_i = A_i \oplus B_i \tag{12}$$

donde el C inicial:

$$C_0 = C_{in}, \quad C_n = C_{out}.$$

1.2.4. Parametrización

El número de bits n se define como un *parámetro* en la descripción hardware. Para sumadores grandes, es común agrupar en bloques de k bits (p. ej., $k = 4$) y encadenar varias unidades CLA de k bits con una etapa adicional de anticipación entre bloques. Esto permite instanciar sumadores de 4, 8, 16 bits o más, manteniendo una latencia de acarreo considerablemente menor que en el *ripple-carry*.

1.3. Selección de método

Se elige la arquitectura *ripple-carry adder* (RCA) por su sencillez de implementación, ya que únicamente requiere encadenar sumadores completos de 1 bit de manera secuencial, lo que facilita su descripción parametrizable y su comprensión académica. Aunque el *carry lookahead adder* (CLA) ofrece menor retardo en la propagación del acarreo, su complejidad lógica y de interconexión lo hace menos práctico en este contexto, por lo que el RCA resulta más adecuado como propuesta de diseño.

2. Propuestas de diseño restador

2.1. Ripple Borrow Subtractor

Un restador completo de 1 bit recibe como entradas dos bits A, B y un préstamo de entrada (B_{in}), y produce como salidas la diferencia (D) y un préstamo de salida (B_{out}). El restador de n bits se construye encadenando n restadores completos, de forma que el préstamo se propaga de manera secuencial, en una arquitectura tipo *ripple*.

2.1.1. Ecuaciones del bloque base

Para cada bit i:

$$D_i = A_i \oplus B_i \oplus B_i^{in} \tag{13}$$

$$B_i^{out} = (\overline{A_i} \cdot B_i) + (\overline{A_i} \cdot B_i^{in}) + (B_i \cdot B_i^{in}) \tag{14}$$

2.1.2. Tabla de verdad del restador completo

A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

2.1.3. Ecuaciones generales

Sea $A = (A_{n-1}, A_{n-2}, \ldots, A_0)$ y $B = (B_{n-1}, B_{n-2}, \ldots, B_0)$. El cálculo iterativo es:

$$D_i = A_i \oplus B_i \oplus B_i^{in} \tag{15}$$

$$B_{i+1}^{in} = (\overline{A_i} \cdot B_i) + (\overline{A_i} \cdot B_i^{in}) + (B_i \cdot B_i^{in}) \tag{16}$$

donde B_0^{in} es el préstamo inicial (por defecto 0) y B_n^{out} indica el préstamo final.

2.1.4. Parametrización

El número de bits n se define como parámetro, lo que permite instanciar restadores de cualquier tamaño con la misma lógica.

2.2. Borrow Lookahead substractor

Al igual que en los sumadores, el retardo del ripple puede ser elevado para n grande. Se puede mejorar definiendo señales de **propagación** y **generación de préstamo**:

$$P_i = (A_i \oplus B_i) \tag{17}$$

$$G_i = (\overline{A_i} \cdot B_i) \tag{18}$$

De este modo, los préstamos pueden calcularse de manera anticipada (lookahead).

2.2.1. Ecuaciones generales

$$B_1 = G_0 + (P_0 \cdot B_0) \tag{19}$$

$$B_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot B_0) \tag{20}$$

$$B_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot B_0) \tag{21}$$

$$\vdots \tag{22}$$

$$B_n = G_{n-1} + (P_{n-1} \cdot G_{n-2}) + \cdots + (P_{n-1} \cdot \cdots \cdot P_0 \cdot B_0) \tag{23}$$

Las diferencias se calculan como:

$$D_i = P_i \oplus B_i$$

2.2.2. Tabla de verdad del restador completo

A	B	B_{in}	D	B_{out}	P	G
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	1	1	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	1	1	0	0

Donde:

$$P_i = A_i \oplus B_i, \quad G_i = \overline{A_i} \cdot B_i$$

2.2.3. Parametrización

El diseño permite definir n como parámetro, y opcionalmente agrupar en bloques (p. ej. de 4 bits) con lógica de lookahead entre bloques para optimizar aún más el retardo.

2.3. Selección de método

La Propuesta 1 (Ripple) resulta más simple y clara para fines académicos, al construirse directamente con restadores completos en cascada. La Propuesta 2 (Borrow Lookahead) ofrece mejor desempeño en retardos, pero su implementación es más compleja. Por lo tanto, se elige la Propuesta 1 como diseño base, por su sencillez y facilidad de parametrización.

3. Propuestas de diseño multiplicador

En esta sección se presenta la idea para implementar el bloque de multiplicación de la ALU parametrizable sin utilizar el operador del HDL, partiendo de circuitos básicos:

- Generación de productos parciales con compuertas `AND`.
- Reducción/suma de productos parciales con sumadores (HA/FA) y/o compresores (CSA).
- Suma final con un sumador N–bit (ripple, carry lookahead, etc.).

3.0.1. Preliminares y tablas de verdad

Célula elemental de producto parcial. Para bits $a_i, b_j \in \{0, 1\}$:

$$p_{i,j} = a_i \cdot b_j = \text{AND}(a_i, b_j).$$

Tabla de verdad de la compuerta AND (1–bit):

a_i	b_j	$p_{i,j}$
0	0	0
0	1	0
1	0	0
1	1	1

Sumador medio (HA) y sumador completo (FA).

$$\text{HA:} \quad \begin{cases} S = x \oplus y \\ C = x \cdot y \end{cases} \qquad \text{FA:} \quad \begin{cases} S = x \oplus y \oplus c_{in} \\ C = xy + xc_{in} + yc_{in} \end{cases}$$

Tablas de verdad (resumen):

HA		Salida	
x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

FA			Salida	
x	y	c_{in}	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3.1. Multiplicador en *matriz/array* (Baugh–Wooley)

El multiplicador por array genera todos los productos parciales con compuertas AND y los dispone en una malla regular alineada según su peso. Cada columna se reduce utilizando sumadores completos (FA) y medios (HA). Para manejar números en complemento a 2 se utiliza el método **Baugh–Wooley**, que transforma los términos asociados a los bits de signo mediante complementos e inyección de unos, manteniendo la estructura uniforme del arreglo. Finalmente, la suma de las dos últimas filas se realiza con un sumador de $2N$ bits.

3.1.1. Ecuaciones generales

Dados los operandos:

$$A = \sum_{i=0}^{N-1} a_i 2^i, \qquad B = \sum_{j=0}^{N-1} b_j 2^j$$

los productos parciales son:

$$p_{i,j} = a_i \cdot b_j$$

y el producto final:

$$P = A \times B = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} p_{i,j} 2^{i+j}.$$

3.1.2. Ejemplo para 4 bits

Para $A = (a_3 a_2 a_1 a_0)$ y $B = (b_3 b_2 b_1 b_0)$, se obtienen 16 productos parciales. Estos se colocan en columnas según su peso $i + j$ y se reducen con HA/FA. En las columnas que incluyen a_3 o b_3 se aplican las reglas de Baugh–Wooley para representar correctamente el signo. La etapa final utiliza un sumador de 8 bits.

3.2. Multiplicador en árbol (Wallace/Dadda con CSA)

El multiplicador por árbol también genera todos los productos parciales, pero en lugar de reducirlos de forma secuencial, emplea compresores 3:2 (basados en sumadores completos) en una estructura de árbol. Este esquema, conocido como **Wallace** o **Dadda**, reduce la altura de cada columna de manera logarítmica, disminuyendo así la ruta crítica. Cuando quedan dos filas de sumandos y acarreos, se suman con un sumador rápido (CLA o similar).

3.2.1. Ecuaciones generales

Un compresor 3:2 recibe tres bits de igual peso (x, y, z) y genera:

$$s = x \oplus y \oplus z, \qquad c = xy + xz + yz$$

donde s se mantiene en la misma columna y c se propaga a la columna de peso $+1$.

Los productos parciales $p_{i,j} = a_i b_j$ se agrupan por columnas y se reducen iterativamente hasta quedar en dos filas.

3.2.2. Soporte de signo

El manejo de números en complemento a 2 puede realizarse de dos formas:

1. Aplicando la técnica de Baugh–Wooley sobre los productos parciales antes de la compresión.

2. Extendiendo los bits de signo y ajustando constantes para mantener la equivalencia algebraica.

3.3. Selección de método

La Propuesta 1 (Array Baugh–Wooley) ofrece simplicidad, regularidad estructural y facilidad de implementación en FPGA, aunque con un retardo proporcional al tamaño de los operandos. La Propuesta 2 (Árbol Wallace/Dadda) es más rápida, pero requiere un diseño más complejo. Dado que el objetivo es una implementación académica y de demostración, se selecciona la Propuesta 1, por su claridad y facilidad de documentación, siendo suficiente en desempeño para anchos de palabra pequeños y medianos.