
long-short

Universidad Nacional de Córdoba
Facultad de Ciencias Exáctas, Físicas y Naturales



**Proyecto Integrador de la Carrera
Ingeniería Electrónica**

*”Diseño de un Sumador Rápido en tecnología CMOS
submicrónica utilizando Herramientas de Software Libre”*

Noviembre 2014

Índice general

Índice general	I
Índice de figuras	V
Índice de cuadros	VII
1. INTRODUCCIÓN	IX
1.1. Estructura del Proyecto Integrador	IX
1.2. Planteamiento del problema y motivación	IX
1.3. Objetivo	IX
1.4. Plan de Trabajo	X
I Diseño Digital	1
2. ESPECIFICACIONES DE DISEÑO	3
2.1. Introducción	3
2.2. Métricas de calidad	3
2.2.1. Performance	3
2.2.2. Potencia promedio disipada	5
2.2.3. Área	5
2.2.4. Resumen	5
3. DISEÑO DIGITAL	7
3.1. Introducción	7
3.1.1. Semisumador y sumador completo	7
3.2. Selección de la arquitectura del sumador	8
3.2.1. Costo, Retardo y Área de los circuitos combinacionales	8
3.2.2. Clasificación de los sumadores	8
3.2.3. Carry Lookahead Adders	9
3.2.4. Desenrollando la recurrencia del acarreo	10
3.2.5. Sumadores de Prefijo Paralelos (<i>Parallel Prefix Adders</i>)	11
3.2.6. Selección de la arquitectura	13
3.3. Implementación en Lenguaje de Descripción de Hardware	15
3.3.1. Implementación del RCA en lava	16
3.3.2. Patrones de conexión	17

3.3.3.	Sumador de Brent-Kung	18
3.3.4.	Simulación	20
3.3.5.	Síntesis del Netlist VHDL	21
4.	VERIFICACIÓN FORMAL	23
4.1.	Modelo de referencia	23
4.2.	Verificación de las Propiedades	24
4.2.1.	Propiedades de la suma	24
4.2.2.	Descripción de las propiedades en el RCA	24
II	Diseño Físico	27
5.	Flujo de Diseño Físico	29
5.1.	Introducción	29
5.1.1.	Etapas del diseño físico	29
5.2.	Relevamiento, comparación y selección de las herramientas disponibles	31
5.2.1.	Relevamiento	32
5.2.2.	Comparación	32
5.2.3.	Selección	32
5.3.	Selección del proceso de fabricación	33
5.3.1.	Obleas multiproyectos	33
5.4.	Selección de las Celdas estándar	35
5.4.1.	Características	36
5.5.	Ubicación y Cableado (<i>Place & Route</i>)	36
5.5.1.	Modificación al código fuente de la herramienta de síntesis lógica	37
5.5.2.	Configuración de la herramienta de Place & Route (PnR)	37
5.5.3.	Distintas alternativas y resultados	37
5.6.	Comparación de las distintas arquitecturas	38
5.6.1.	Performance y Potencia	38
5.6.2.	Tablas comparativas de performance, potencia y área	40
6.	Sign Out y Tape Out	43
III	Comparación de resultados	45
7.	Comparación de resultados	47
7.1.	Tablas comparativas	47
IV	Conclusiones	49
8.	Conclusiones	51
A.	NETLIST VHDL	53
B.	SCRIPT PERL	57

Índice de figuras

2.1.	Retardo de propagación de un inversor	4
2.2.	Estimación de Potencia Promedio Disipada	5
3.1.	Bit adders	8
3.2.	Ripple Carry Adder	9
3.3.	CLA 4-bits	11
3.4.	Retardo respecto al tamaño de los operandos	13
3.5.	Área respecto al tamaño de los operandos	14
3.6.	Operator Punto de Brent-Kung	15
3.7.	Generación y Propagación del Acarreo	15
3.8.	Sumador de Brent-Kung	16
3.9.	Red de prefijos paralelos (ejemplo de 16 bits)	16
3.10.	Diferentes Patrones de Conexión de Circuitos	17
3.11.	Construcción de la red de prefijos paralelos	19
4.1.	circuito addZero	25
5.1.	Flujo de diseño Físico	30
5.2.	Celda estándar con 3 capas de metales en color arena, y una capa de silicio poli- cristalino en color ladrillo. Azul y rojo son dopado N^+ y P^+ respectivamente . .	31
5.3.	Mapeo de una función lógica a una celda estándar	36
5.4.	Grilla de interconexión y riel de alimentación de las celdas estándar de 128λ . Por encima de cada celda, pueden pasar 16 pistas horizontales que la herramien- ta de conexión tendrá a disposición, a partir del metal 3 para arriba. Notar la separación de 8λ para todas las pistas horizontales, y 8λ para las verticales también. Sólo en la intersección de las pistas puede ubicarse los pines de entra- da/salida de la celda, así como los contactos a <i>bulk</i>	37
5.5.	Conjunto de celdas estándar	38
5.6.	Configuración del Silicon Compiler	38
5.7.	Tres arquitecturas y tres tamaños de sumandos distintos. El gráfico está en escala, la unidad de los dos ejes es λ	41
6.1.	Flujo de diseño Físico	43

Índice de cuadros

2.1.	Especificaciones de diseño para el sumador binario	3
2.2.	Métricas de comparación	6
3.1.	Resumen Características de Sumadores	14
5.1.	Procesos disponibles por medio de MOSIS	34
5.2.	Procesadores fabricados en CMOS 180nm	34
5.3.	Ubicación y conexionado para Ripple carry en 3 tamaños: 8, 16 y 32 bits	39
5.4.	Ubicación y conexionado para Sklansky en 3 tamaños: 8, 16 y 32 bits	39
5.5.	Ubicación y conexionado para Brent-Kung en 3 tamaños: 8, 16 y 32 bits	40

Capítulo 1

INTRODUCCIÓN

En el presente capítulo se describe en rasgos generales el flujo para el diseño de Circuitos Integrados de Aplicación Específica (*ASIC* por su sigla en inglés), y la metodología utilizada para llevar adelante el diseño, implementación y tape out del mismo.

1.1. Estructura del Proyecto Integrador

Este proyecto cuenta con 4 partes: Las primeras tres partes:

?: Selección de la Arquitectura

?: Implementación Física

III: Comparación de resultados.

Estas tres partes forman un flujo de trabajo circular e iterativo.

Y finaliza con un resumen de las conclusiones, además de conjunto de anexos técnicos ubicados al final para su consulta.

1.2. Planteamiento del problema y motivación

En la actualidad los microprocesadores, los DSP, los microcontroladores, y otro hardware específico para cálculo computacional son desarrollados en tecnología CMOS submicrónica. El problema planteado es, ¿Cómo hacer para diseñar circuitos integrados en esta tecnología, con herramientas flexibles, libres¹ y accesibles para todo tipo de uso: academico y comercial?.

1.3. Objetivo

El objetivo del trabajo es diseñar un sumador de n-bits, por ser este el elemento central de cualquier tipo de circuito digital de cálculo: Los multiplicadores, los MAC (*multiply-accumulate*), los filtros FIR, etc, que pueda ser enviado a fabricar utilizando procesos de fabricación CMOS para circuitos integrados. Integrar y documentar un flujo de diseño de este sistema digital utilizando herramientas de Software Libre, será un subproducto de este diseño, para lograr la base de conocimiento necesaria en el diseño de circuitos integrados con tecnología CMOS. Este trabajo

¹En el sentido que no impongan restricciones de uso, estudio, mejora y distribución.

además de integrar todos los procesos de diseño de un Circuito Integrado, pretende facilitar el acceso a las herramientas de diseño de circuitos integrados a los estudiantes de grado.

1.4. Plan de Trabajo

Parte I

Diseño Digital

Capítulo 2

ESPECIFICACIONES DE DISEÑO

2.1. Introducción

Los sumadores binarios son utilizados en la adición, la resta, la multiplicación y la división. La velocidad de un sistema de procesamiento de señales, o un sistema de comunicación depende fuertemente de **estas unidades funcionales**(14). Para cada una de esas operaciones, son necesarios sumadores de distinta cantidad de bits en el mismo diseño. Por lo cual, no se trata solamente de encontrar la arquitectura que para una determinada cantidad de bits logre el mejor compromiso de área, potencia y velocidad. Sino que también esta relacion se mantenga óptima para diferentes tamaños del sumador.

Por estas razones, precisamos diseñar un sumador de N-dígitos que sea lo más rápido posible, manteniendo una relación de compromiso óptima entre la velocidad, consumo de energía y área del circuito. Estas características las resumimos en el cuadro 2.1.

Parámetro	Especificación
Sumandos	Dos ¹
Cantidad de Dígitos	Parametrizable
Proceso de fabricación	Disponible por medio de MOSIS ²
Retardo de propagación	Lo mas bajo posible
Potencia total disipada	Tan bajo como sea posible
Área del Circuito	La menor posible

Cuadro 2.1: Especificaciones de diseño para el sumador binario

2.2. Métricas de calidad

Definiremos las métricas que nos permitan dar cuenta de la calidad del diseño.

2.2.1. Performance

El término performance puede representar distintas métricas, según desde qué perspectiva se esté realizando el análisis. Pero si nos enfocamos puramente en el diseño, la performance se define usualmente(14) como la duración del período del clock (o su frecuencia). El valor mínimo

de período de clock que pueda ser usado para una tecnología y un diseño dado, está definido por múltiples factores, como el tiempo que le toma a las señales propagarse a través de la lógica (retardo de propagación), el tiempo que lleva entrar y salir los datos de los registros, la incertidumbre de llegada del reloj (*clock uncertainty*). Pero el núcleo de todo análisis de performance reside en la performance de una sola compuerta.

Retardo de propagación

El retardo de propagación t_p de una compuerta define cuán rápido responde un circuito a un cambio en su(s) entrada(s). Expresa el retardo experimentado por una señal cuando pasa a través de una compuerta. Medido entre el 50 % del punto de transición de entrada y salida, como mostramos en la figura 2.1, correspondiente al tiempo de propagación de una compuerta inversora. Ya que el tiempo de propagación es distinto según el flanco de entrada, se definen 2 tiempos de propagación. El t_{pLH} es el tiempo de respuesta de una compuerta para una transición de la salida desde bajo a alto, mientras que t_{pHL} se refiere a el tiempo para una transición de la salida desde alto a bajo. El retardo de propagación t_p se define como el promedio de estos dos.

$$t_p = \frac{t_{pLH} + t_{pHL}}{2}$$

Camino Crítico

En un circuito digital con varias entradas y salidas, pueden existir mas de un camino desde la entrada hasta la salida. Se suele denominar camino crítico a aquel camino que tenga el mayor retardo de propagación, ya sea por cantidad de lógica que atraviesa o por las capacidades parásitas de las conexiones. El retardo de propagación de este circuito será el retardo de propagación del camino crítico.

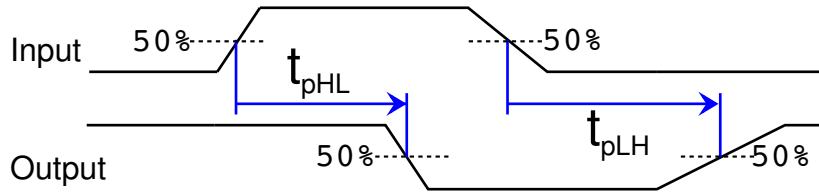


Figura 2.1: Retardo de propagación de un inversor

Mínimo retardo de propagación

Para poder comparar la performance de distintas tecnologías, se busca un circuito que no incluya parámetros como el fan-in o fan-out, que influyen en los tiempos t_f , t_r y t_f . Por ello, el circuito que es un estándar de facto para medir el tiempo de propagación, es el oscilador anillo (*ring oscillator*), que es un número impar de inversores conectados en serie, con la salida conectada a la entrada. Este circuito oscila espontáneamente, a una frecuencia de $T = 2 \times t_p \times N$, con N el número de inversores en la cadena.

Contar con esta métrica nos permitirá tener una referencia del límite inferior impuesto por la tecnología que se esté utilizando. Por ejemplo, tomemos la tecnología TSMC de 180 nm: La frecuencia de un oscilador anillo de 31 etapas es de 377,13 MHz. Es decir que el tiempo de propagación de una celda inversora en esta tecnología es $t_p = 47,8 \text{ ps}$

2.2.2. Potencia promedio disipada

Realizamos el análisis de potencia a lo largo de un período de tiempo T . La potencia promedio disipada total la podemos calcular si conocemos la corriente instantánea que brinda la fuente de tensión V_{DD} , como podemos ver en la ecuación 2.1.

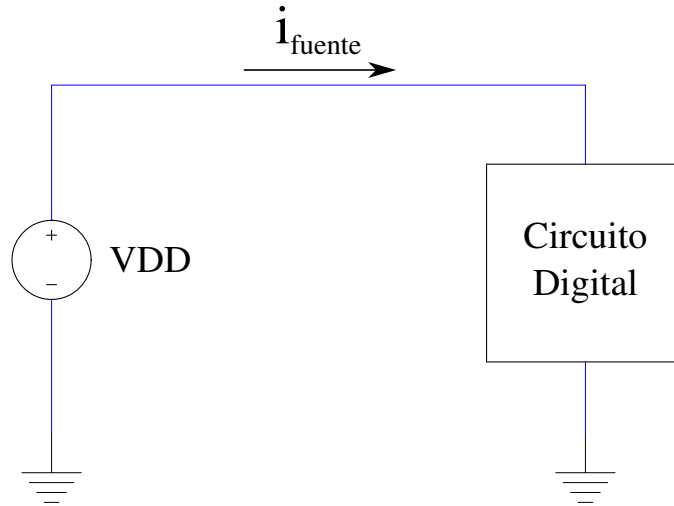


Figura 2.2: Estimación de Potencia Promedio Disipada

$$P_{av} = \frac{1}{T} \int_0^T p(t) dt = \frac{V_{DD}}{T} \int_0^T i_{fuente}(t) dt \quad (2.1)$$

El período de tiempo que tomaremos para la integral es el retardo de propagación del camino crítico.

2.2.3. Área

La importancia de minimizar el área de los circuitos radica principalmente en que esta impacta fuertemente en el costo de cada *die*(7), ya que el costo es una función que depende de la cuarta potencia del área del circuito(14). Además, los circuitos de menor área tienden a consumir menor energía.

2.2.4. Resumen

A continuación resumimos en la tabla 2.2 las métricas que utilizaremos para la comparación de las distintas arquitecturas de sumadores:

Capítulo 3

DISEÑO DIGITAL

3.1. Introducción

Es importante lograr sumadores binarios rápidos y eficientes según el uso de área y potencia. La suma es la operación elemental para lograr otras operaciones muy utilizadas en los circuitos aritméticos. Ejemplo de esto son los multiplicadores, la resta, división, los filtros FIR e IIR, por nombrar las más conocidas.

Para cada una de esas operaciones, son necesarios sumadores de distinta cantidad de bits en el mismo diseño. Por lo cuál, no se trata solamente de encontrar la arquitectura que para una determinada cantidad de bits logre el mejor compromiso de área, potencia y velocidad. Sino también lograr una relación de compromiso según crece la cantidad de bits del sumador.

3.1.1. Semisumador y sumador completo

Semisumador

El **Semisumador** (Half-adder) recibe 2 bits de entradas a y b y produce un bit de suma s y un bit de acarreo c .

$$s = a \oplus b \quad (3.1a)$$

$$c = ab \quad (3.1b)$$

Sumador Completo

Luego definimos un Sumador Completo de un bit, o Full Adder:

Entradas: Bits de operandos a , b y carry-in c_{in} (o a_i, b_i, c_i para la etapa i)

Salidas: Suma s y carry-out c_{out} (o s_i y c_{i+1} para la etapa i)

$$s = a \oplus b \oplus c_{in} \quad (3.2a)$$

$$c_{out} = ab + ac_{in} + bc_{in} \quad (3.2b)$$

Podemos construir un **sumador completo** (full-adder) combinando las ecuaciones del sumador y semisumador, como vemos en la figura 3.1b:

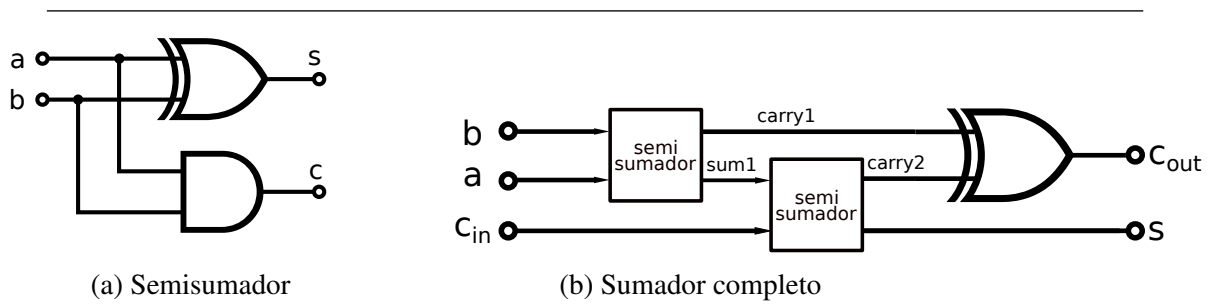


Figura 3.1: Bit adders

3.2. Selección de la arquitectura del sumador

Proponemos el uso de Celdas estándar CMOS (Complementary Metal Oxide Silicon) para la implementación¹. El carácter de nuestro flujo de diseño así lo requiere, ya que se utilizarán herramientas de síntesis de circuitos digitales basadas en celdas estándares. Quedan entonces descartadas las implementaciones utilizando transistion gates, lógica dinámica u otro tipo de implementacion lógica.

3.2.1. Costo, Retardo y Área de los circuitos combinacionales

Cada circuito combinacional G tiene un costo, área y un retardo. El costo de un circuito combinacional es la suma de los costos de las compuertas en un circuito. Le asignamos un costo unitario a cada compuerta, y el costo del circuito combinacional $c(G)$ es igual al número de compuertas en el circuito.

El retardo de un circuito combinacional $d(G)$ se define igual al del retardo de una compuerta. Es el menor tiempo requerido para que las salidas se estabilicen, asumiendo que todas las entradas están estables. Para simplificar el análisis, se le asigna un retardo unitario a cada compuerta.

El área de un combinacional se compone por el area total (*cell area*) de las compuertas utilizadas mas el área total de todas las conexiones (*net area*).

3.2.2. Clasificación de los sumadores

Dentro de los sumadores paralelos, se encuentran varias arquitecturas, cada una con sus ventajas y desventajas. Hacemos una lista de algunas de ellas:

¹Para ver otras posibilidades de implementación lógica, ver (FALTA CITA) RABAEY

Sumadores Binarios	
RCA	Ripple Carry Adder
CLA	Carry Look-Ahead Adder
CSkA	Carry Skeep Adder
CA	Canonical Adders
BBCLA	Block-based Look-Ahead Adders
CondSumA	Conditional Sum Adder
CSeA	Carry Select Adder
HybAd	Hybrid Adders
NPA	Network Prefixs Adders:
	Ladner and Fischer
	Kogge-Stone
	Brent-Kung
	Skalansky

Ripple Carry Adder

Definimos el sumador Ripple Carry Adder (RCA), utilizando n sumadores completos para sumar 2 operandos de n bits. El sumador de n bits produce una salida de n bits y una salida de acarreo c_{out}

Este sumador se implementa conectando como muestra la figura 3.2 el bloque `fullAdd` (Sumador Completo). El camino crítico de la señal se determina considerando el peor camino de propagación de la señal.

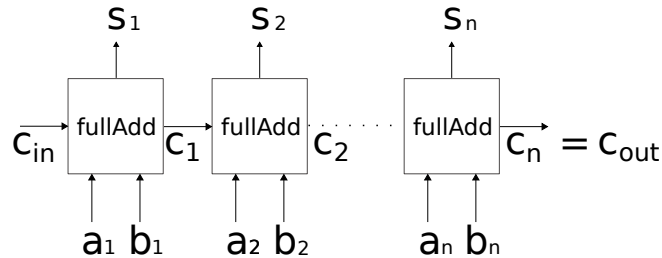


Figura 3.2: Ripple Carry Adder

El retardo del camino crítico de un sumador de n bits es:

$$T_{RCA} = (n - 1)T_m + T_{FA} \quad (3.3)$$

Siendo T_m el retardo del circuito de generación del acarreo de un sumador completo y T_{FA} el retardo de un sumador completo. Es decir, el retardo es proporcional al tamaño de los operandos.

3.2.3. Carry Lookahead Adders

La clave para sumar rápido es plantear el problema de la suma como el problema de generar las señales de acarreo en el menor tiempo posible; eso queda evidenciado al interpretar la ecua-

ción 3.4. Por lo tanto, el objetivo será lograr un bloque generador de las señales de acarreo de baja latencia(13).

Ya que una vez que el acarreo en la posición i es conocido, se puede calcular la suma como:

$$s_i = a_i \oplus b_i \oplus c_i \quad (3.4)$$

Con respecto al acarreo, lo importante es si en una posición dada el acarreo se *genera* ó se *propaga*. Con las siguientes ecuaciones lógicas podemos definir esas señales:

$$g_i = a_i b_i$$

$$p_i = a_i \oplus b_i$$

Asumiendo que estas señales se han calculado y están disponibles, podemos calcular recursivamente el acarreo de la siguiente forma:

$$c_{i+1} = g_i + c_i p_i \quad (3.5)$$

quiere decir que un acarreo entrará en una etapa $i + 1$ si este se genera en la etapa i ó entra en la etapa i y se propaga en esa etapa.

3.2.4. Desenrollando la recurrencia del acarreo

Uno puede desenrollar esta fórmula recursiva del acarreo hasta lograr una función que dependa directamente de los operandos (a y b) y del acarreo de entrada c_{in} :

$$\begin{aligned} c_i &= g_{i-1} + p_{i-1}c_{i-1} \\ &= g_{i-1} + p_{i-1}(g_{i-2} + p_{i-2}c_{i-2}) = g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}c_{i-2} \\ &= g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}g_{i-3} + p_{i-1}p_{i-2}p_{i-3}c_{i-3} \\ &= g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}g_{i-3} + p_{i-1}p_{i-2}p_{i-3}g_{i-4} + p_{i-1}p_{i-2}p_{i-3}p_{i-4}c_{i-4} \end{aligned}$$

El proceso se repite hasta que el último término contenga $c_0 = c_{in}$. Podemos computar todos los acarreos en un sumador de k -bit directamente con las señales auxiliares (g_i, p_i) y c_{in} , utilizando compuertas lógicas AND-OR con un fan-in máximo de $k + 1$. Para $k = 4$, tenemos:

$$\begin{aligned} c_4 &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 \\ c_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \\ c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\ c_1 &= g_0 + p_0c_0 \end{aligned} \quad (3.6)$$

Aquí, c_4 y c_0 son los c_{out} y c_{in} respectivamente de un sumador de 4-bits. Podemos usar un bloque de acarreo basado en estas ecuaciones, y usando compuertas AND de 2 entradas para g_i y compuertas XOR de 2 entradas para p_i y los bits de suma, construimos un sumador de 4-bits. Este sumador es conocido como *carry lookahead adder (CLA)*. Notar que como c_4 no se usa para calcular la suma, lo podemos obtener usando una ecuación mas simple:

$$c_4 = g_3 + c_3p_3$$

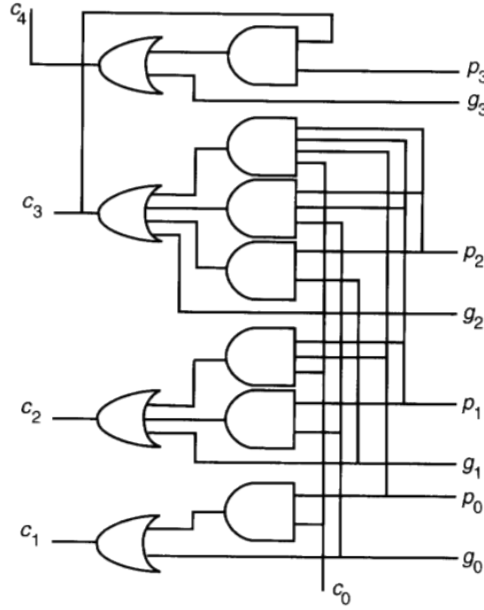


Figura 3.3: CLA 4-bits

Sin tener casi un deterioro en velocidad. La red de acarreo que resulta de estas ecuaciones la podemos ver en la figura 3.3.

Si observamos las ecuaciones 3.6, vemos que el retardo de esta red será el retardo T_{AND_n} de la mayor celda AND, mas el retardo T_{OR_n} de la operación OR de n entradas. Esto es un inconveniente, ya que según aumenta el fan-in también aumenta el retardo. El retardo de un sumador construido con esta red tendrá también el retardo T_p del cálculo de p mas el retardo de un sumador completo.

$$T_{CLA} = T_p + T_{AND_n} + T_{OR_n} + T_{FA} \quad (3.7)$$

Se pueden realizar por medio de árboles binarios una reducción a celdas con un fan-in de dos (por ejemplo), pero agregando una etapa por cada reducción, en ese caso el retardo en este circuito sería en función del $\log_2 n$.

3.2.5. Sumadores de Prefijo Paralelos (*Parallel Prefix Adders*)

En la sección anterior vimos como desarrollar ecuaciones que nos permiten obtener las señales de acarreo a partir de las señales auxiliares, para poder calcular la suma del bit n , sin esperar a que el acarreo del bit $n - 1$ sea computado. Aunque esta solución tal cuál como la presentamos deja de ser aplicable según aumenta n , nos permite abordar el problema del cálculo de los acarreos como un problema de prefijos paralelos.

Problema de Prefijos Paralelos (*Parallel Prefix Problem*)

El problema de prefijo paralelo es:

Dado:

Entradas: x_0, x_1, \dots, x_{k-1}

Un operador + asociativo

Computar : x_0

$x_0 + x_1$

$x_0 + x_1 + x_2 +$

\vdots

$x_0 + x_1 + x_2 + \dots + x_{k-1}$

Cómputo del acarreo como un problema de prefijo paralelo

Pensemos la ecuación 3.6 de la siguiente forma, asumiendo que $c_0 = c_{in}$ viene desde otro bloque:

$$g_{[i,i+3]} = g_{i+3} + g_{i+2}p_{i+3} + g_{i+1}p_{i+2}p_{i+3} + g_i p_{i+1}p_{i+2}p_{i+3}$$

$$p_{[i,i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$$

Podemos interpretar estas ecuaciones de la siguiente forma: las cuatro posiciones de bits propagan colectivamente un acarreo c_{in} si y solo si cada una de las posiciones propaga; y el bloque genera un acarreo si en la posición $i + 3$ se genera uno, o se produce en la posición $i + 2$ y es propagado por la posición $i + 3$, etc.

Con este procedimiento podemos llegar a expresar una generalización muy importante, para bloques adyacentes que se superponen $[i_1, j_1]$ y $[i_0, j_0]$, con $i_0 \leq i_1 - 1 \leq j_0 < j_1$:

$$g_{[i_0,j_1]} = g_{[i_1,j_1]} + g_{[i_0,j_0]}p_{[i_1,j_1]}$$

$$p_{[i_0,i_1]} = p_{[i_0,j_0]}p_{[i_1,j_1]}$$

Aquí, $g_{[i_0,j_1]}$ y $p_{[i_0,i_1]}$ son las señales que producimos de 2 bloques adyacentes (B'' y B' con sus señales asociadas (g'', p'') y (g', p')) que para simplificar la notación nos permite reescribir la anterior ecuación como:

$$g = g'' + g'p''$$

$$p = p'p''$$

Ahora entonces definimos un operador acarreo \circ para condensar estas operaciones:

$$(g, p) = (g'', p'') \circ (g', p') = (g'' + g'p', p'p'')$$

Este operador es un operador asociativo, y esto se puede demostrar utilizando la propiedad asociativa de los operadores OR y AND. Finalmente, ya tenemos un operador asociativo, y las entradas $(g''', p'''), (g'', p''), (g', p'), \dots$ que nos permiten plantear el problema de la construcción de la red

(o bloque) de acarreo, como un problema de *Prefijos Paralelos*:

Dados:

Entradas: $(g_0, p_0), (g_1, p_1), \dots, (g_{k-1}, p_{k-1})$

Un operador \circ asociativo

Computar : $(G_0, P_0) = (g_{[0,0]}, p_{[0,0]})$

$(G_1, P_1) = (g_{[0,0]}, p_{[0,0]}) \circ (g_{[0,1]}, p_{[0,1]})$

\vdots

$(G_{k-1}, P_{k-1}) = (g_{[0,0]}, p_{[0,0]}) \circ (g_{[0,1]}, p_{[0,1]}) \circ \dots \circ (g_{[0,k-2]}, p_{[0,k-2]}) \circ (g_{[0,k-1]}, p_{[0,k-1]})$

Retomando la ecuación 3.4 de la suma, y con estas ecuaciones que nos dan las señales propagadas o generadas del acarreo, podemos construir distintos sumadores, que varían en la red de cálculo del acarreo, particularmente en cómo se elija la asociación del operador punto. La implementación mas básica (y lenta) sería la de ir asociando en serie a este operador.

PORHACER: AGREGAR GRAFICO DE UNA RED SERIE

3.2.6. Selección de la arquitectura

Se puede afirmar que los llamados sumadores paralelos prefijo son mejores con respecto al producto Potencia - Retardo. Aunque no hay una estructura que pueda calificarse como globalmente la mejor. Estos sumadores reducen el problema de calcular de forma paralela las señales de acarreo como el problema de un cálculo de prefijo.

Brent Kung(4), Kogge Stone (9), Ladner Fisher [3], Hans Carlson(9) y Knowles(8) son implementaciones que se diferencian en que cada caso por minimizar alguna relación de compromiso, en el espacio de diseño para minimizar retardo, area y potencia. Por ejemplo citamos un estudio que presenta los siguientes resultados de la figuras 3.4 y 3.5 de un estudio comparativo (1) para tecnología CMOS 0,13 μm .

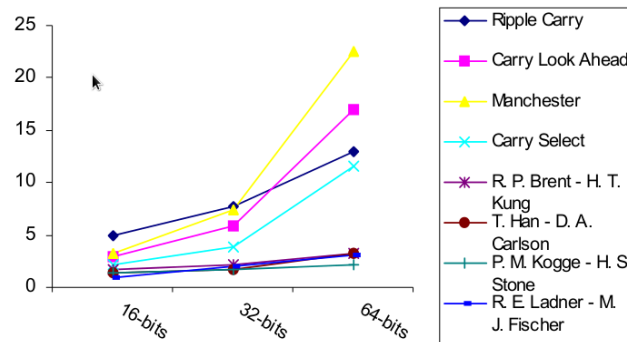


Figura 3.4: Retardo respecto al tamaño de los operandos

s

Resumimos en la tabla 3.1 las características y diferencias entre los distintos sumadores(2).

Sumador Rápido de Brent-Kung

Para tener en cuenta el problema de la interconexión entre las compuertas de forma tal que estas sean mínimas y que el área de celdas y de conexión se minimicen, se propone el sumador

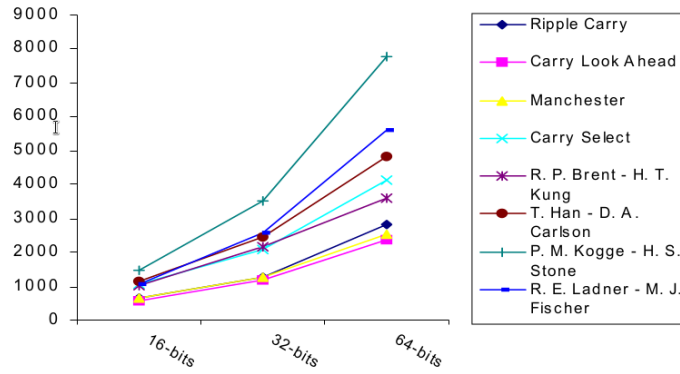


Figura 3.5: Área respecto al tamaño de los operandos
Cuadro 3.1: Resumen Características de Sumadores

Arquitectura	Retardo Máx.	Área
Ripple Carry Adder (RCA)	$O(n)$	$O(n)$
Carry Save Adder (CSaA)	$O(\log(n))$	$O(n)$
Carry Look-Ahead Adder (CLA)	$O(\log(n))$	$O(n \log(n))$
Carry Skip Adder (CSA)	$O(n^{l+2/l+1})$	$O(n)$
Carry Increment Adder (CIA)	$O(n^{l+2/l+1})$	$O(n)$
Carry Select Adder (CselA)	$O(n^{l+2/l+1})$	$O(n)$
Ladner-Fisher	$O(\log_2(n))$	$O(n \log(n))$
Skalansky	$O(\log_2(n))$	$O(\log^2(n))$
Kogge-Stone	$O(\log_2(n))$	
Han-Carlson	Falta	Falta
Brent-Kung	$O(\log_2(n))$	$O(n \log_2(n))$

de Brent-Kung. Este sumador (4) es una versión que considera el problema de la interconexión entre las compuertas, de una forma que minimice el área, a costa de un aumento en el retardo. Esto se expresa en la función de retardo que es $2 \log_2(n) - 2$, a diferencia de los sumadores de Ladner-Fisher(10) y Kugge-Stone(9) que en $\log_2(n)$ etapas calculan todos las señales de acarreo.

Operador de Brent-Kung

El operador \circ se define¹ como:

$$(g, p) \circ (\hat{g}, \hat{p}) = (g \vee (p \wedge \hat{g}), p \wedge \hat{g}) \quad (3.8)$$

El operador Punto de Brent-Kung es asociativo, es decir:

$$((a, b) \circ (c, d)) \circ (e, f) = (a, b) \circ ((c, d) \circ (e, f))$$

Y por lo tanto podemos ahorrarnos los paréntesis y escribimos:

$$(a, b) \circ (c, d) \circ (e, f) \circ \dots$$

¹ Para respetar la notación de la bibliografía original comenzamos a utilizar la notación lógica con \vee , \wedge y \oplus como los operadores booleanos AND, OR y XOR respectivamente

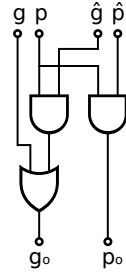


Figura 3.6: Operador Punto de Brent-Kung

Circuito de Generación y Propagación de acarreo

Ahora necesitamos un circuito que con cada bit de entrada de los operandos a y b calcule la señal de acarreo y la de propagación:

$$g_i = a_i \wedge b_i, p_i = a_i \oplus b_i$$

Esas señales se generan en paralelo, dado dos números binarios $a[n]$ and $b[n]$ de longitud n .

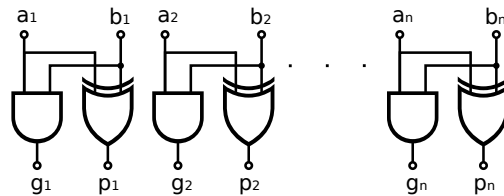


Figura 3.7: Generación y Propagación del Acarreo

Circuito completo

Asumiendo que ya tenemos diseñado el bloque de cálculo de los acarreos en cadena, al cual le llamamos red de prefijos paralelos, el circuito propuesto por el paper de Brent-Kung(4) lo podemos ver como en la figura 3.8, que nos servirá a la hora de la implementación en HDL.

Red de Prefijos Paralelos

Con la figura 3.9, detallamos ahora la red de prefijos paralelos con un fan-out máximo de dos, lo cuál diferencia a el sumador de Brent-Kung de los otros sumadores de prefijos paralelos. La red se realiza con 2 elementos: Los puntos negros son los operadores punto de Brent-Kung de la figura 3.6 y con buffers (los puntos blancos) que realizan una copia de la señal. Cada cable representa un par de bit g_i, p_i de la figura 3.8.

3.3. Implementación en Lenguaje de Descripción de Hardware

Ya hemos presentado una descripción esquemática del sumador binario de n bits. El objetivo es implementar el circuito en un lenguaje de HDL parametrizado por el tamaño n de los números binarios a ser sumados. Para describir el circuito, elegimos Lava, un sistema para diseñar,

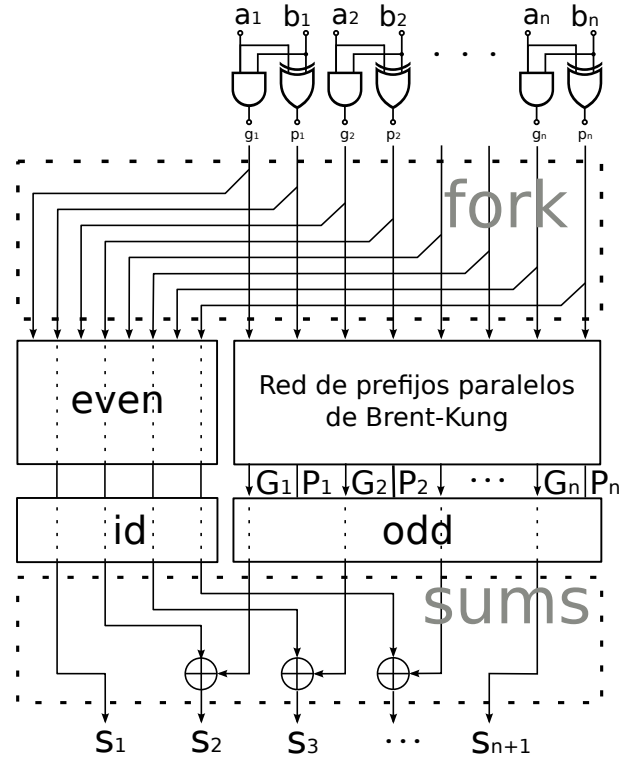


Figura 3.8: Sumador de Brent-Kung

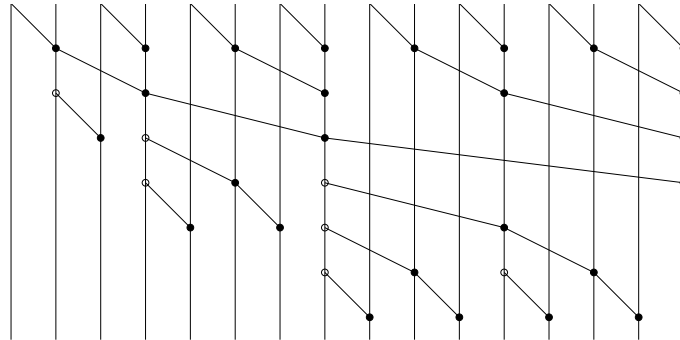


Figura 3.9: Red de prefijos paralelos (ejemplo de 16 bits)

especificar, verificar e implementar hardware. Lava está embebido en el lenguaje de programación funcional Haskell. En lava los circuitos son descritos como funciones que operan sobre listas, tuplas o sobre circuitos. Esto último se debe que el lenguaje Haskell permite la definición de funciones de alto orden, es decir podemos definir funciones que su dominio e imagen son funciones.

3.3.1. Implementación del RCA en lava

Siguiendo la figura 3.1a, definiremos el semisumador:

```
halfAdd (a, b) = (s, c)
  where
    s = xor2 (a, b)
    c = and2 (a, b)
```

Para escribir el circuito del sumador completo usamos la figura 3.1b, nombrando las señales internas y escribiendo los subcomponentes de la siguiente forma:

```
fullAdd (cin, (a, b)) = (s, cout)
  where
    (sum1, carry1) = halfAdd (a, b)
    (s, carry2) = halfAdd (cin, sum1)
    cout = xor2 (carry2, carry1)
```

Por último escribimos la descripción del sumador binario (RCA) de la figura 3.2 de la siguiente forma:

```
adder (carryIn, ([], [])) = ([], carryIn)
adder (carryIn, (a:as, b:bs)) = (sum:sums, carryOut)
  where
    (sum, carry) = fullAdd (carryIn, (a, b))
    (sums, carryOut) = adder (carry, (as, bs))
```

3.3.2. Patrones de conexión

Patrones de conexión estandar. Los patrones de conexión son funciones de alto orden¹ que pueden ser utilizadas para construir circuitos, les llamamos circuitos de alto orden o generadores de circuitos.

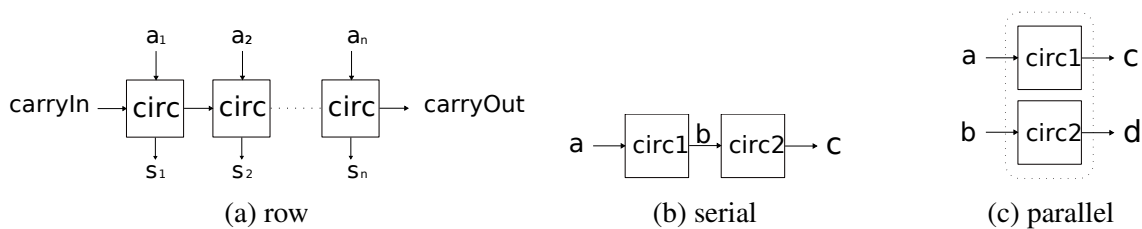


Figura 3.10: Diferentes Patrones de Conexión de Circuitos

Observando la definición de `adder` y su topología, podemos generalizar esa estructura de conexión reemplazando el circuito por un parámetro, que en la definición² del circuito será una entrada mas. A ese parámetro lo nombramos `circ`:

```
row circ (carryIn, ([])) = ([], carryIn)
row circ (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry) = circ (carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

La función `row` toma un circuito `circ`, un conjunto de entradas, y las conecta como se muestra en la figura 3.10a. Ahora, usando el generador de circuito `row`, el sumador binario lo podemos describir mas simplemente así:

¹Las funciones de alto orden (*high order functions*) son funciones que toman otras funciones como argumento y devuelven otra función como resultado.

²Esto es posible dado que Haskell implementa *pattern matching*.

```
adder' (carry, inps) = row fullAdd (carry, inps)
```

Inclusive para simplificar mas, podemos currificar¹ la definición:

```
adder'' = row fullAdd
```

Definir `adder'` y `adder''` de esa forma es bastante conveniente ya que podemos pensar en término de *generadores de circuitos* en vez de recursión sobre listas.

Ya que hemos visto la ventaja de definir los patrones de conexión, presentamos dos generadores de circuitos que vamos a usar mas tarde:

```
par cir1 cir2 (a, b) = (c, d)
  where
    c = cir1 a
    d = cir2 b
```

Es muy útil definir una versión mas gráfica de la función `par`, si definimos el operador infijo `-|-`:

```
cir1 -|- cir2 = par cir1 cir2
```

Y por último la conexión serie y su versión con el operador infijo:

```
serial cir1 cir2 a = c
  where
    b = cir1 a
    c = cir2 b

cir1 ->- cir2 = serial cir1 cir2
```

3.3.3. Sumador de Brent-Kung

Operador de Brent-Kung

Comencemos a describir el sumador de Brent-Kung. En Lava, podemos describir el circuito que implementa la función 3.8 siguiendo la figura 3.6:

```
dotOp ((g1, p1) , (g, p)) = (go, po)
  where
    go = or2 (g, and2 (p, g1))
    po = and2 (p, p1)
```

Generación y Propagación del Acarreo

En Lava escribimos asi lo que captamos de la figura 3.7:

```
gAndPs ([], []) = []
gAndPs (a:as, b:bs) = (g,p):gps
  where
    (g, p) = (and2 (a, b), xor2 (a, b))
    gps    = gAndPs (as, bs)
```

Para ver una explicación con mayor nivel de detalles de cómo construir el circuito, ver el manual de lava (5) en conjunto con el paper aqui citado (11)

¹Curricular, es una referencia al lógico Haskell Curry, y hace referencia a la técnica que consiste en transformar una función que utiliza una n-tupla como argumento, en una función que utiliza un único argumento.

Red de Prefijos Paralelos

Ahora para describir esta red que usamos en la figura 3.8 y mostramos un ejemplo de una red para 16 bits en la figura 3.9, nos basamos en un patrón recursivo que propone Sheeran (16) al que le llama *wrap*. En cada paso de la iteración tomamos el resultado anterior (el circuito *P*) y le aplicamos el operador punto antes y después de forma intercalada como se puede ver en la figura 3.11a. Esto nos lleva a construir redes como la de la figura 3.9.

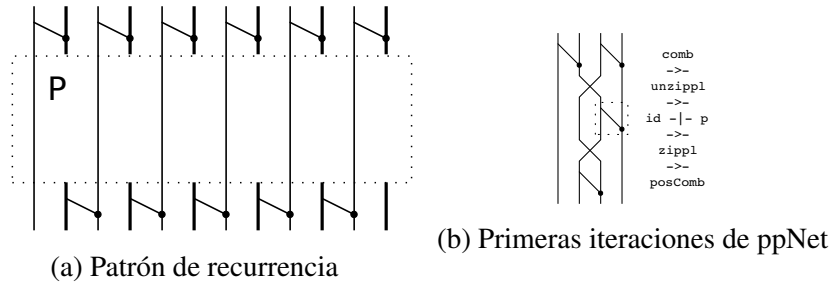


Figura 3.11: Construcción de la red de prefijos paralelos

La figura 3.11b representa las dos primeras iteraciones del circuito ppNet, en el cual la caja de líneas punteada es el caso base de la descripción, los puntos negros son la función `dotOp`. Lo que producimos con esta función recursiva son redes como la de la figura 3.9.

A continuación, describimos el circuito ppNet, pero antes escribimos las funciones auxiliares `dop`, `unzipl`, `zipl`, `comb`, `posComb`, `miti` y `wrap`, que nos servirán para escribir ppNet:

```
dop [a, b] = [a, dotOp(a, b)]
--
unzipl []      = ([], [])
unzipl [a]     = ([a], [])
unzipl (a:b:abss) = (a:as, b:bs)
  where
    (as, bs) = unzipl abss
--
zipl ([], [])      = []
zipl ([a], [])     = []
zipl (a:as, b:bs) = a:b:zipl(as, bs)
--
-- La forma en que hemos escrito las funciones zipl y unzipl
-- son la clave para lograr una descripción de un sumador
-- binario que acepte cualquier cantidad de entradas
--
comb []      = []
comb [a]     = []
comb (a:as) = dop [a, head as] ++ comb (tail as)
--
posComb (a:as) = a: (comb (init as)) ++ [last as]
--
miti p = unzipl ->- (id -|- p) ->- zipl
--
wrap p = comb ->- miti p ->- posComb
```

Luego finalmente, podemos describir ppNet:

```
ppNet [a]      = []
ppNet [a, b]   = dop [a, b]
ppNet as       = wrap ppNet as
```

Circuito top level

Ahora que ya tenemos construidas todas las partes del sumador, sólo resta juntarlas siguiendo el esquemático de la figura 3.8. Prestar atención a que el circuito `fork` realiza una copia de las señales, el `even` deja pasar los bits pares, `odd` los impares, `id` es la función identidad y `sums` mapea los bits de entrada con la función booleana XOR, salvo el primer y último bit:

```
fork as = (as, as)
--
even as = cs
  where
    (bs,cs) = unzip as
--
odd as = bs
  where
    (bs,cs) = unzip as
-- Unas definiciones mas cortas:
dropP = id -|- odds
dropG = even -|- ppNet
--
sums (a:as,bs) = (a:lastXor (as,init bs),cOut)
  where
    cOut = last bs
--
lastXor (as, bs) = map xor2 cs
  where
    cs = zipp (as, bs)
--
zipp ([],[]) = []

zipp (a:as, b:bs) = c:cs -- da lo mismo que poner (c:cs)
  where
    c = (a, b)
    cs = zipp (as, bs)
```

Y el circuito completo es:

```
fastAdd = gAndPs ->- fork ->- dropG ->- dropP ->- sums
```

3.3.4. Simulación

En Lava podemos simular el circuito usando la operación `simulate`, el circuito y el estado de las entradas, por ejemplo:

```
simulate fastAdd ([high,low],[low,high])
```

devuelve: `([high,high],low)`. También podemos simular secuencia de entradas con la operación `simulateSeq`:

```
simulateSeq halfAdd [(low,low),(high,low),(low,high)]
```

que devuelve `[(low,low),(high,low),(high,low)]`

Simulaciones con números decimales

Lava nos permite una interfase con números enteros, por si nos interesa simular usando como operandos números enteros. Esto lo logramos si definimos una función como la siguiente, que toma dos enteros y convierte el segundo en un número binario de la cantidad de bits que indica el primero:

```
int2bin 0 num = []

int2bin n num = (bit:bits)
  where
    (bit, num) = numBreak num
    bits      = int2bin (n-1) num
```

Método de validación del hardware

Para este diseño en particular, no utilizaremos la simulación como una forma de validar el correcto funcionamiento del circuito, por eso no avanzaremos en las distintas alternativas de simulación que nos permite el sistema, como puede ser la creación de un archivo VCD¹ a partir de vectores de entrada².

Justificamos descartar la simulación como método de validación por la simple razón de que sólo simulando todos los posibles estados de las entradas se garantiza el correcto diseño del circuito. Por ejemplo, para un sumador de 64 bits, es necesario simular 2^{128} estados.

Para este tipo de sistemas es aplicable la verificación formal automática, que desarrollaremos en el capítulo 4.

3.3.5. Síntesis del Netlist VHDL

Para continuar en nuestro flujo de diseño, precisamos generar el circuito en un lenguaje que nuestra herramienta de *Place and Route* pueda manejar. Para eso lava nos permite crear un netlist VHDL siguiendo dos pasos, el primero definiendo los nombres de los puertos y el bloque a ser creado:

```
fastAdder n = writeVhdlInputOutputNoClk
  "BrentKungFastAdder" fastAdd
  (varList n "a", varList n "b")
  (varList n "sum", var "cout")
```

¹VCD: Value Change Dump es un formato basado en ASCII para loguear señales, que es utilizado por herramientas de simulación lógica. Para visualizarlo podemos utilizar el software GTKWave, de licencia libre.

²Podemos usar una librería de Haskell llamada *casualmente* vcd que nos permite escribir y leer archivos con este formato

Y el segundo paso para crear el netlist, debemos especificar el valor real de sumador, por lo tanto valuamos el circuito con el número de bits del sumador y conseguiremos el archivo BrentKungFastAdder.vhl que mostramos en el apéndice [A](#):

```
Main> fastAdder 16  
Writing to file "BrentKungFastAdder.vhd" ... Done..
```

Si por alguna razón este netlist lo utilizáramos con una otra herramienta de síntesis, deberemos especificar que no modifique los cables para preservar la estructura de esta red.

Capítulo 4

VERIFICACIÓN FORMAL

Como aclaramos en el capítulo 3, el correcto funcionamiento del circuito se garantiza por medio de la verificación formal de las propiedades de la suma.

Nuestro flujo para esta etapa tiene que ver con la verificación de propiedades que se denominan *safety properties*. Estas son propiedades que se mantienen como verdaderas siempre (o lo que es equivalente, nunca son falsas). En Lava escribimos estas propiedades de la misma forma en que escribimos los circuitos, inclusive utilizando otros circuitos que nos sirvan para expresar una condición. Esto se verá con mas claridad cuando avancemos con la verificación. Entonces, la pregunta que estamos haciendo para verificar cualquier propiedad descrita de esta forma es: ¿Este circuito de verificación siempre tiene como salida el estado `True` sin importar cuales son las entradas? Para responder esta pregunta, en Lava usamos la operación `verify`.

Este proceso funciona así: Tal como podemos generar un netlist VHDL (o la simulación) a partir de la descripción del circuito, también podemos generar una fórmula lógica que representa al circuito. Esta fórmula lógica se la damos a un probador de teorema externo que nos probará (o desaprobará) la validez de la fórmula. El probador externo que usaremos es `miniSAT`¹.

4.1. Modelo de referencia

A los fines de la verificación, usaremos un sumador de referencia `adder` bien simple, en el cual podamos probar todas las propiedades de la suma, para luego hacer un chequeo de equivalencia lógica (LEC por sus siglas en inglés) entre el sumador de referencia y el sumador que queremos implementar. Esto es conveniente porque es una gran ventaja (desde el punto de vista de tiempo de cálculo) hacer todas las pruebas sobre circuitos mas simples (pequeños), para luego realizar una sola comprobación de equivalencia lógica entre este circuito simple y el circuito diseñado, garantizando así que si todas las propiedades se cumplen en uno, también se cumplen en el otro.

¹Minisat es un programa que resuelve problemas conocidos como *Boolean satisfiability problem (SAT)*, o directamente *SAT solver*

4.2. Verificación de las Propiedades

4.2.1. Propiedades de la suma

La suma tiene las siguientes propiedades:

- Asociativa
- Conmutativa
- Existencia del elemento neutro cero.

4.2.2. Descripción de las propiedades en el RCA

Debido a que nuestro sumador de Brent-Kung asume que el acarreo de entrada es cero, debemos modificar nuestro sumador de referencia (un Ripple Carry Adder) para que desprecie el acarreo de entrada. Por lo tanto describimos nuevamente una versión del RCA de la siguiente forma:

```
adder2 ([], []) = []
adder2 (a:as, b:bs) = sum:sums
  where
    (sum, carry)      = halfAdd (a, b)
    (sums, carryOut) = adder (carry, (as, bs))
```

Propiedad Conmutativa

Ahora declaramos la propiedad conmutativa de la suma de la siguiente forma:

```
prop_AdderCommutative (as, bs) = ok
  where
    out1 = adder2 (as, bs)
    out2 = adder2 (bs, as)
    ok    = out1 <==> out2
```

Notar que el operador <==> es la versión infija de una función que mapea dos listas a la compuerta `xnor2`, la cual es la operación de equivalencia lógica. Como es muy difícil verificar automáticamente para cualquier tamaño, definimos una nueva propiedad que incluye el tamaño del circuito a ser verificado:

```
prop_AdderCommutative_ForSize n =
  forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      prop_AdderCommutative (as, bs)
```

Luego hacemos la verificación corriendo Minisat desde Lava, dando el tamaño del sumador:

```
minisat (prop_AdderCommutative_ForSize 32)
```

Si nuestro circuito está correctamente diseñado, tenemos:

```
Minisat: ... (t=0.00system) Valid.
```

De otro modo, podemos tener uno de estos resultados:

```
Minisat: ... (t=0.00system) Falsifiable.
Minisat: ... (t=0.00system) Inderterminate.
```

Propiedad Asociativa

La propiedad Asociativa la declaramos como:

```
prop_AdderAssociative (as, bs, cs) = ok
  where
    out1 = adder2 (adder2 (as, bs), cs)
    out2 = adder2 (as, adder2 (bs, cs))
    ok    = out1 <==> out2
```

Existencia del Elemento Neutro

Para verificar que el cero es el elemento neutro de la adición, necesitamos escribir un poco mas de lógica al circuito para transformar uno de los operandos a cero:

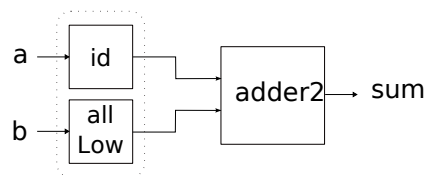


Figura 4.1: circuito addZero

```
alwaysLow :: [Signal Bool] -> [Signal Bool]
alwaysLow (as) = [low | n <- [1..n]]
  where
    n = length as
```

```
addZero = (id -|- alwaysLow) ->- adder2
-- id es la funcion identidad
```

Y la verificación de esta propiedad en el circuito:

```
prop_AdderZero (as,bs) = ok
  where
    out = addZero (as, bs)
    ok   = out <==> as
```

Equivalencia lógica entre el RCA y el sumador de Brent-Kung

Finalmente, hacemos la equivalencia lógica entre los dos circuitos: `fastAdd` (el BKA) y `adder2` (el RCA). Eso se declara en Lava de la siguiente forma:

```
prop_Equivalent adder2 fastadd a = ok
  where
    out1 = adder2 a
    out2 = fastadd a
    ok    = out1 <==> out2
```

Parte II

Diseño Físico

Capítulo 5

Flujo de Diseño Físico

5.1. Introducción

En este punto convertimos la representación de un circuito (con sus componentes e interconexiones) a una representación en formas geométricas, conocida como *layout*. Dicho en otras palabras, explicaremos (y realizaremos) el proceso que logra transformar una descripción de funciones lógicas a una representación de formas geométricas del circuito integrado, que luego de ser fabricado con las capas correspondientes, nos aseguran que obtendremos los transistores ubicados e interconectados dentro de un chip de silicio, de forma tal que implemente nuestro sistema digital.

El proceso que explicaremos en términos generales, y que podemos ver en contexto en la figura 5.1, se realiza iterativamente hasta lograr que el circuito cumpla las especificaciones con el menor costo en potencia disipada y área ocupada.

5.1.1. Etapas del diseño físico

Generalmente en un flujo de este tipo, partimos desde una descripción estructural del circuito. Esta descripción, comúnmente llamada *netlist*, contiene información sobre qué bloques están presentes, y cómo estos están interconectados.

Particionado Según el tamaño del circuito, será necesario definir particiones del mismo, dividiendo el circuito en dos o mas particiones con fines de acotar la magnitud o dificultad inicial del circuito original, en partes más pequeñas de menor dificultad, si las particiones se realizan correcta e inteligentemente.

Plano general Luego es necesario definir un plano general del circuito (mencionado como *floorplan* en la bibliografía en inglés), que impondrá condiciones físicas mínimas como el área utilizada y la disposición física de las entradas y salidas.

Ubicación A continuación, ubicamos en este plano todos los componentes del circuito (conocido como *placement* en la bibliografía en inglés), en una disposición tentativa que nos permita evaluar rápidamente la factibilidad del circuito con las condiciones impuestas por el *floorplan*, por ejemplo si todos los componentes y el conexionado caben dentro del *floorplan*. Dependiendo de las herramientas que utilicemos, también se puede tener una estimación sobre la velocidad de las señales.

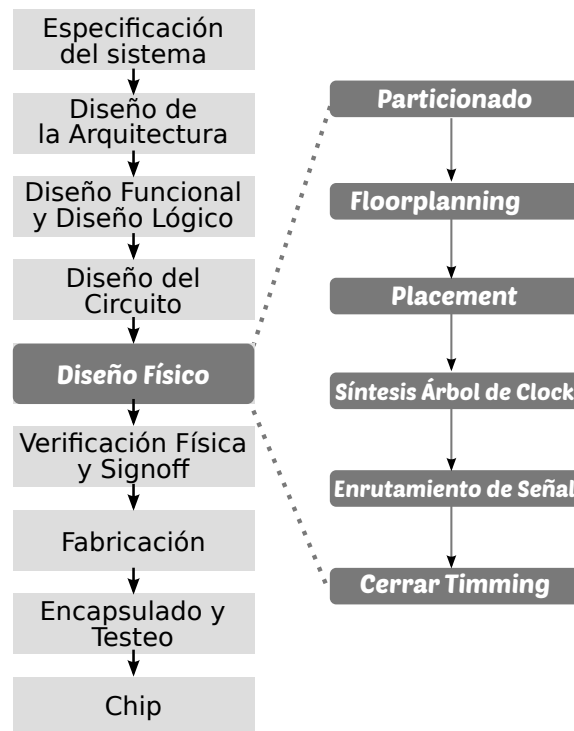


Figura 5.1: Flujo de diseño Físico

Síntesis del árbol de reloj Una vez que todos los elementos estén en el plano, si el circuito es secuencial, será necesario realizar una distribución de la señal del reloj para que llegue a todos los registros, de la forma más pareja (en tiempo) posible dentro de un margen de tolerancia determinado. Para ello se agregan *buffers* donde sea necesario. Este proceso se conoce como [Clock Tree Synthesis \(CTS\)](#) en la bibliografía en inglés.

Conexionado Por último, se realiza el conexionado de todos los puertos de cada componente, utilizando las capas de metal disponible en la tecnología que se esté utilizando, un ejemplo de este conexionado se puede ver en la figura 5.2. Este proceso se conoce como *routing* en la bibliografía en inglés.

En este punto, se puede realizar la mejor estimación sobre las capacidades y resistores parásitos que representan la interconexión de todo el circuito. Se encuentran los caminos críticos y se realizan las modificaciones necesarias para que el circuito cumpla con la especificaciones de retardo de propagación máximo. Siempre en cada etapa de este proceso se puede iterar para mejorar el resultado, pero si aún así no logramos la mejora necesaria, debemos volver a iterar sobre una etapa anterior y continuar este flujo, secuencialmente.

El procesos de ubicación de los componentes e interconexionado que acabamos de describir, es muy común que se mencione como [PnR](#), por sus siglas en inglés.



Figura 5.2: Celda estándar con 3 capas de metales en color arena, y una capa de silicio policristalino en color ladrillo. Azul y rojo son dopado N^+ y P^+ respectivamente

5.2. Relevamiento, comparación y selección de las herramientas disponibles

Para realizar las tareas que describimos en la sección 5.1.1, será necesario buscar una o varias herramientas de software que se ajusten a los requerimientos del diseño y la tecnología de fabricación¹ del circuito integrado.

Características esperadas de las herramientas

- Desarrollo activo y existencia de una comunidad de usuarios/as y desarrolladores/as que brinden soporte
- Mayor cantidad de herramientas integradas

¹Se utilizará una tecnología definida por Mead y Conway(12), conocida como **SCMOS** (Scalable CMOS). Esto es un conjunto de capas lógicas junto a sus reglas de diseño, que proveen un proceso casi independiente de la tecnología y dimensión, que sirve para muchos procesos CMOS disponibles a través de MOSIS.

-
- Flexibilidad para importar y exportar datos.
 - Disponibilidad de un Kit de diseño para el proceso de la tecnología seleccionada, conocido como [Process Design Kit \(PDK\)](#).

oo

5.2.1. Relevamiento

Luego de una inspección de esas características, las herramientas candidatas que cumplen con estas características son:

Open Circuit Design Proyecto de software libre que reúne en un único sitio varias herramientas independientes, mencionamos sólo algunas: **Magic**: Layout, [Design Rule Check \(DRC\)](#) y extracción de parásitos; **Xcircuit**: Entrada de circuitos esquemáticos; **netgen**: [Layout Vs. Schematic \(LVS\)](#); **IRSIM**: simulador digital a nivel de transistor como llaves ideales, con extracción de capacidad y resistores concentrados para hacer la simulación mas realista; **Qflow**: entorno para realizar la síntesis digital con celdas estándar, utiliza Yosys(18); **gray-wolf**: programa que realiza el *placement*; **Qrouter**: programa que realiza el conexionado.

Electric VLSI Design System(6) Es un sistema de automatización de diseño electrónico. Es un entorno integrado muy flexible que permite la descripción del circuito de varias formas (circuitos esquemáticos, *netlist* VHDL y *layout*). Cuenta también con herramientas para hacer [DRC](#), [LVS](#) y [PnR](#), simulación digital, visualización de formas de ondas, y un generador de *pad frame*¹, entre otras herramientas.

Alliance VLSI CAD System(17) Alliance es un conjunto de herramientas libres, y celdas estándar para el diseño de VLSI. Incluye un compilador vhdl y un simulador, herramientas de síntesis de lógica, y herramientas de [PnR](#) automáticas. Brinda un conjunto completo de celdas estándar CMOS escalables.

Es importante mencionar que existen mas herramientas disponibles (y muy útiles), pero al momento de realización de este trabajo, no forman parte de un flujo de diseño que las integre y por ello no son mencionadas aquí.

5.2.2. Comparación

Resaltamos las ventajas y desventajas de cada herramienta, que nos permitirá hacer una selección en función de las necesidades del proyecto.

5.2.3. Selección

La herramienta que seleccionamos es **Electric**, ya que nos brinda una serie de ventajas comparativas, teniendo en cuenta que nuestro circuito es puramente combinacional y no demanda gran esfuerzo de [PnR](#) a la herramienta:

¹El *pad frame* es un conjunto de celdas que se ubican en el marco del *die*, para conectar las señales del circuito con el exterior del chip.

-
- Fácil instalación
 - Curva de aprendizaje suave
 - Cuenta con todas las herramientas necesarias integradas

El hecho de que no cuente con un sintetizador lógico, como señalamos en la tabla ??, no tiene importancia para esta selección, ya que uno de los resultados del diseño digital del capítulo 3 es un *netlist* VHDL. Por la naturaleza de la solución propuesta, no deseamos que este resultado sea modificado por alguna optimización lógica, ya que rompería la interconexión original de nuestro circuito. Si quisieramos comparar nuestra implementación del circuito con la implementación automática de la operación suma, entonces tendríamos que pasar a una de las otras herramientas. Pero eso sería un trabajo de comparación de un diseño *custom* con uno automático, que no era el objetivo de este proyecto.

5.3. Selección del proceso de fabricación

En este punto, es importante mencionar un aspecto de la industria de los semiconductores. En los orígenes, la industria de semiconductores estaba verticalmente integrada. Esto significaba que la misma empresa que diseñaba el producto, también diseñaba las herramientas de software y fabricaba el chip. Pero hace poco mas de 20 años, surge la separación del proceso de diseño y fabricación. Hoy en día existen empresas que se dedican sólomente a desarrollar el producto, otras que se dedican únicamente a desarrollar herramientas de software para el diseño, otras que sólomente se dedican a fabricar los diseños de otras empresas, y también persisten las empresas que realizan todo el proceso (conocidas como *fab*), abriendo las puertas a otras empresas de diseño sin fábrica (conocidas como *fabless*), para evitar la capacidad ociosa instalada y disminuir sus costos.

5.3.1. Obleas multiproyectos

Dentro de este esquema, existe una empresa (MOSIS) que se dedica a recolectar proyectos de diseño que están en etapa de prototipo o de bajo volumen, creando obleas multiproyecto que se envían a fabricar, dividiendo los costos por la cantidad de proyectos que incluye. De esta forma, se logra acceder a la fabricación de semiconductores a muy bajo costo. Tiene sus limitaciones en cuanto a tecnologías de fabricación disponibles, cantidades, y tiempo de entrega largos, pero permite que proyectos educativos, de investigación o de baja escala sean económicamente factibles. Por ello, nuestras opciones serán alguna de las que MOSIS ofrece. En la tabla 5.1 vemos una lista que está en constante cambio y actualización, se brinda aquí de modo ilustrativo. Para una lista actualizada visitar <https://www.mosis.com/products/fab-processes>.

De todas estas, elegimos TSMC 180 nm por dos razones: la primera es que cuanto mayor es la dimensión de la tecnología, más simples son las herramientas de software necesarias y más bajo es el costo de fabricación. La segunda, es que con esta tecnología se pueden realizar sistemas de gran complejidad y alta performance¹

¹Claro que cuanto más nueva es la tecnología, los circuitos digitales son más rápidos y disipan menor potencia dinámica. Pero también es cierto que mayores son los tiempos para diseñar, principalmente porque con cada nuevo nodo aparecen nuevos efectos físicos que deben ser manejados, dificultando las tareas.

Fábrica	Proceso CMOS
TSMC	28 nm - 180 nm
Globalfoundries	14 nm - 180 nm
IBM	32 nm - 250nm
ON Semi	0.35 um - 0.7 um
Austria Micro Systems	180 nm - 0.35 um

Cuadro 5.1: Procesos disponibles por medio de MOSIS

Para dar cuenta de las capacidades de esta tecnología, vemos en la tabla 5.2 un conjunto de microprocesadores que la utilizaron cuando esta era la más avanzada en su tiempo (desde el año 1999 hasta 2001) e inclusive después. Pero el verdadero sustento de que esta tecnología es actual, es que al día de hoy se continúan desarrollando varias aplicaciones, siendo una mejor opción que nodos mas nuevos, por razones económicas. Con el desarrollo de nuevas técnicas para la disminución de consumo de energía¹, la gran colección de *IP*² analógico³ y digital que cada fábrica ofrece, las ventajas de necesitar menor poder de cálculo que para los nodos actuales (22 nm), mucha experiencia acumulada por parte de los diseñadores, se consigue un menor TTM⁴ con menores costos.

Procesador	Año de lanzamiento
Intel Coppermine E	1999
AMD Athlon Thunderbird	2000
Intel Celeron (Willamette)	2002
Motorola PowerPC 7445 y 7455 (Apollo 6)	2002

Cuadro 5.2: Procesadores fabricados en CMOS 180nm

MOSIS especifica los procesos disponibles que soportan las reglas escalables en su documento ***Design Rules. MOSIS Scalable CMOS (SCMOS), Revision 8.00.***, en el cuál encontramos que para 180 nm sólo podemos elegir a la fábrica TSMC.

Además, podemos optar entre las reglas **SCN6M_SUBM** y las **SCN6M_DEEP**. Decidimos utilizar la segunda, ya que tiene un valor de λ menor, lo cual significa un tamaño de transistor resultado más óptimo.

El proceso nos ofrece 6 capas de metal (cobre) para la interconexión, 1 capa de silicio policristalino (*poly*) para crear la compuerta y también para la interconexión de las mismas (distancias cortas sólomente, por su mayor resistividad que el cobre), con 2 tipos de óxidos para crear el aislante de las compuertas, los que pueden ser alimentados con tensión máxima de 1,8V, y los que

¹A modo de ejemplo, ver el procesador *Phoenix*, que en modo alerta consume 29.6 pW y 2.8 pJ/ciclo modo activo(15).

²Intellectual Property, nombre usual dado a los diseños listos para ser usados en un sistema, cuando se decide enfocar el diseño solamente en lo novedoso del producto, comprando el IP de todo lo que no diseñaremos.

³Los circuitos analógicos que ya fueron diseñados y probados para una tecnología deben ser diseñados nuevamente desde cero cuando se pasa de una tecnología a otra, ya que las arquitecturas de circuitos analógicos no son escalables (como si son los digitales)

⁴Time To Market (TTM), sigla utilizada para designar el tiempo que necesita un producto para ser diseñado, fabricado, testado y puesto en producción. Dependiendo de la aplicación, este tiempo va desde meses hasta años.

pueden ser alimentados con 3,3V (pensados principalmente como transistores para los circuitos de entrada y salida del chip). MOSIS denomina a las reglas de diseño que utilizaremos para esta tecnología como SCN6M_DEEP, que significa:

- S: Escalable
- C: Tecnología de fabricación CMOS
- N: Pozo N.
- 6M: 6 metales y un poly
- DEEP: Reglas *deep submicron*.

Las reglas escalables se crearon originalmente para tecnologías desde 3 μ m hasta 1 μ m. Cuando aparecieron tecnologías nuevas, se hicieron modificaciones a las reglas para ajustarse a las nuevas posibilidades. Entonces se crearon primero las reglas *submicron*, y luego las *deep submicron*

Una vez definido la herramienta de diseño (**Electric**) y el proceso de fabricación a utilizar (TSMC 180 nm), definimos la variable λ , que es la unidad que utilizará nuestro software para las dimensiones físicas. Electric define a λ como la mitad del largo de canal mínimo para la tecnología que se está utilizando. En nuestro caso, el largo de canal mínimo es de 180 nm (de allí viene la designación del nombre), por lo tanto λ sería 90 nm. Otra elección que hay que hacer es cuál de los

5.4. Selección de las Celdas estándar

El resultado de la síntesis que realizamos en el capítulo 3 es un *netlist* VHDL que contiene sólo compuertas lógicas. Estas son compuertas lógicas abstractas, es decir que nuestro circuito fué mapeado a un conjunto finito de funciones logicas como las *and*, *or*, *xor*, *xnor*, etc.

Ahora es necesario mapear estas funciones lógicas a compuertas lógicas reales, que serán tambien un conjunto finito de compuertas, pero con dimensiones físicas definidas, y con una caracterización de su funcionamiento real. Estas compuertas lógicas se denominan celdas estándar, que sirven específicamente para la tecnología de fabricación que hayamos definido usar. Por cada función lógica existen distintas versiones de la misma función, pero con distintas características eléctricas. Mostramos en la figura 5.3 un ejemplo de una celda estándar que implementa la función lógica *xnor*

Es común elegir las celdas estándar según el tipo de aplicación a desarrollar. Existen celdas estandar que fueron diseñadas para bajo consumo, o alta velocidad, o de mínima área. También existe la posibilidad de diseñar celdas que busquen la mejor relación velocidad-consumo-area que puedan ser utilizadas en muchas aplicaciones. En circuitos integrados para sistemas alimentados a batería se intentará utilizar las celdas de menor consumo y evitar siempre que sea posible las de mayor velocidad, en función del presupuesto de potencia disponible para el mismo.

En nuestro caso, aprovechando que la suma se realiza con apenas 3 compuertas: *and*, *or*, *xor*, podemos construir nuestro propio conjunto de celdas. Como punto de partida, utilizamos celdas que fueron realizadas para un estudio sobre circuitos digitales operando por debajo de la región de inversión debil(3).

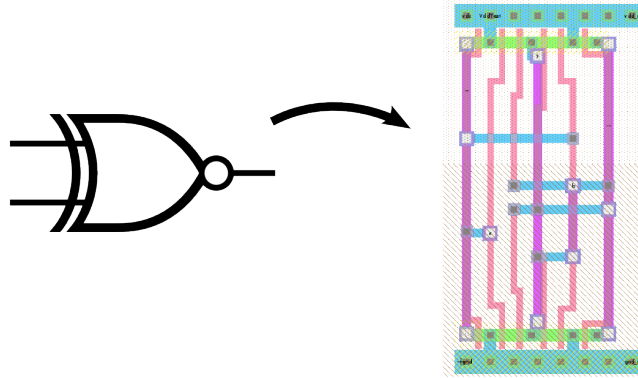


Figura 5.3: Mapeo de una función lógica a una celda estándar

5.4.1. Características

Estas celdas están correctamente dimensionadas para lograr el apilamiento en filas y columnas, y una grilla de interconexión amplia, que nos evitará problemas de este tipo. Para nuestros objetivos, modificamos las dimensiones de los transistores de canal P, para lograr un tiempo de crecida y bajada más simétricos, y así mejorar la velocidad. Resumimos las características de nuestras celdas estándar:

Altura $128 \lambda^1$, es la distancia desde el riel de Vdd hasta Vss, lo cual permite el ruteo horizontal de 16 pistas de metal por encima de las celdas, con metal 3 hasta capas superiores, como vemos en la figura 5.4.

Ancho del riel de alimentación 8λ

Tamaño de los transistores Transistores n: largo y ancho mínimo ($L_n = 2\lambda$, $W_n = 4\lambda$, $\frac{W_n}{L_n} = 2$). Transistores p: Largo mínimo. Dos versiones, una de ancho mínimo con $\frac{W_p}{L_p} = 2$ y otra de mayor fuerza $\frac{W_p}{L_p} = 4$, a la que le agregamos $_1x$ al final del nombre.

Disposición de los pines Se ubican siempre en la intersección de las pistas horizontales y verticales, que tienen una separación de 8λ , ver figura 5.4.

Conexión a bulk Todas las celdas tienen conexión a bulk cada 8λ para evitar el problema conocido como *latch up* que surge a causa de malas conexiones entre el *bulk* y la alimentación.

Otra característica de estas celdas es la distancia entre el riel de VSS hasta el riel de VDD. Esta distancia es de 128λ , lo cual permite el ruteo horizontal de 16 pistas de metal por encima de las celdas, con metal 3 hasta capas superiores, como podemos ver en la figura 5.4.

5.5. Ubicación y Cableado (*Place & Route*)

Partimos desde la descripción estructural² que producimos en el capítulo 3. De *Electric* usaremos la herramienta llamada *Silicon Compiler*, que se encarga de ubicar y conectar las celdas

¹En la bibliografía en ingles se denomina *pitch*

²El resultado de la síntesis hecha con *lava* es un *netlist* VHDL a nivel de compuerta, listo para ser usado por una herramienta de *PnR*.

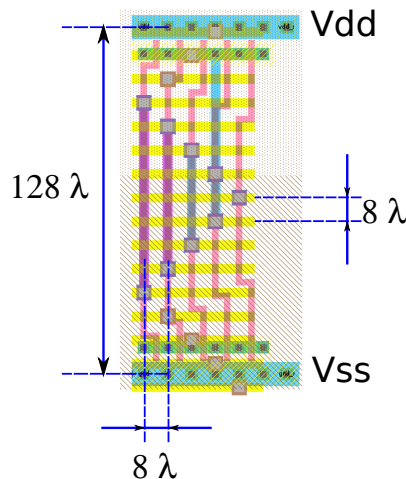


Figura 5.4: Grilla de interconexión y riel de alimentación de las celdas estándar de 128λ . Por encima de cada celda, pueden pasar 16 pistas horizontales que la herramienta de conexión tendrá a disposición, a partir del metal 3 para arriba. Notar la separación de 8λ para todas las pistas horizontales, y 8λ para las verticales también. Sólo en la intersección de las pistas puede ubicarse los pines de entrada/salida de la celda, así como los contactos a *bulk*.

según el *netlist* VHDL.

5.5.1. Modificación al código fuente de la herramienta de síntesis lógica

La herramienta *Silicon Compiler* requiere que el *netlist* VHDL sea modificado levemente:

1. Es necesario agregar las celdas estándar como componente en la porción declarativa de la arquitectura de la entidad en VHDL, que serán utilizadas instanciadas en el circuito.
2. Los nombres no pueden usar los símbolos [y], por lo cual es necesario eliminarlos.

Modificamos el código del programa de **lava** llamado `VhdlNew.hs`, encargado de crear el *netlist* VHDL para realizar (1), y para lograr (2) agregamos una línea para que lance un pequeño programa escrito en **perl** que mostramos en el apéndice B. Luego de esta modificación, cuando realizamos la síntesis lógica descrita en el capítulo 3, el *netlist* VHDL obtenido ya puede ser utilizado por *Silicon Compiler*.

5.5.2. Configuración de la herramienta de PnR

En la figura 5.6 vemos la configuración necesaria para hacer el PnR con nuestras celdas estándar. La configuración se realiza para ajustar la herramienta a las reglas de drc) y las dimensiones de nuestra celdas estándar. La variable de ajuste para modificar el *floorplan* es el parámetro *Number of rows of cells*. Según la cantidad de filas que asignemos, será el resultado obtenido para cada circuito.

5.5.3. Distintas alternativas y resultados

La regla de oro para todo *layout* es que sea lo más cuadrado posible, ya que de esta forma es más eficiente el uso del área cuando integramos nuestro circuito con otros de mayor jerarquía. La

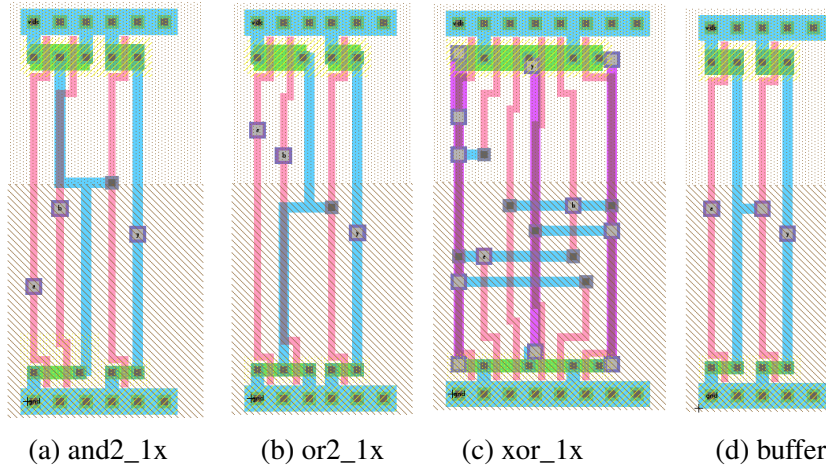


Figura 5.5: Conjunto de celdas estándar

Layout Number of rows of cells: <input type="text" value="4"/>		Well P-Well height (0 for none): <input type="text" value="41"/> P-Well offset from bottom: <input type="text" value="0"/> N-Well height (0 for none): <input type="text" value="51"/> N-Well offset from top: <input type="text" value="0"/>	
Arcs Horizontal routing arc: <input type="text" value="Metal-1"/> Horizontal wire width: <input type="text" value="3"/> Vertical routing arc: <input type="text" value="Metal-2"/> Vertical wire width: <input type="text" value="3"/> Power wire width: <input type="text" value="8"/> Main power wire width: <input type="text" value="30"/> Main power arc: <input type="text" value="Metal-2"/>		Design Rules Via size: <input type="text" value="3"/> Minimum metal spacing: <input type="text" value="10"/> Routing feed-through size: <input type="text" value="16"/> Routing min. port distance: <input type="text" value="8"/> Routing min. active distance: <input type="text" value="-9"/>	

Figura 5.6: Configuración del Silicon Compiler

métrica de selección del resultado será el área que ocupe nuestro circuito y la relación entre sus lados: cuanto más pequeño¹ y cuadrado mejor. En las tablas 5.3, 5.4 y 5.5 vemos los resultados para todos los sumadores analizados, con 3 tamaños distintos y para diferentes alternativas de *floorplan*, variando el parámetro *Number of rows of cells*.

En la figura 5.7 presentamos todos los *layout* seleccionados con el criterio recién mencionado. Resumimos con este gráfico el resultado de PnR de cada arquitectura para 3 tamaños de sumandos distintos.

5.6. Comparación de las distintas arquitecturas

5.6.1. Performance y Potencia

Para realizar la comparación, necesitamos simular nuestros circuitos

¹El área es una métrica de calidad del circuito, según lo planteamos en el capítulo 2.

Ripple Carry	8			16			32		
filas	3	4	5	5	6	7	8	7	6
ancho	1297	966	843	1562	1350	1142	1881	2169	2581
alto	665	839	958	1227	1196	1600	2000	1850	1360
área	862505	810474	807594	1916574	1614600	1827200	3762000	4012650	3510160
ancho/alto	0,51	0,87	1,14	0,79	0,89	1,40	1,06	0,85	0,53

Cuadro 5.3: Ubicación y conexionado para Ripple carry en 3 tamaños: 8, 16 y 32 bits

Skalansky	8			16			32		
filas	3	4	5	4	5	6	7	6	8
ancho	1516	1167	954	3538	2042	1825	1536	3678	3229
alto	810	973	1252	1345	1581	1878	2063	2639	2695
área	1227960	1135491	1194408	4758610	3228402	3427350	3168768	9706242	8702155
ancho/alto	0,53	0,83	1,31	0,38	0,77	1,03	1,34	0,72	0,83

Cuadro 5.4: Ubicación y conexionado para Skalansky en 3 tamaños: 8, 16 y 32 bits

Motores de simulación analógica

Para realizar una simulación que sea la mejor estimación de la performance y potencia, es necesario realizar una extracción del circuito a partir del *layout*, y configurar **Electric** de la siguiente forma:

- Extracción de parásitos: Para que también extraiga las capacidades y resistores parásitos que significan las interconexiones:

File -> Preferences -> Tools -> Spice/CDL -> Parasitics: Conservative RC

- Cargarle los modelo de transistores de la tecnología que estamos usando (TSMC 180 nm). Este es un archivo que nos brinda MOSIS que nosotros debemos editar para quitarle información de la tecnología (resistividad y capacidades de los metales, *poly*, el sustrato y el silicio dopado N^+ y P^+).
- Seleccionara el formato (language) del *netlist spice* del simulador que utilizaremos: Las opciones son: Spice 2, Spice 3, HSpice, PSpice, Gnucap, SmartSpice, Spice Opus, Xyce, HSpice for Assura, HSpice for Calibre. De los cuales, sólo 3 opciones son libres: Spice 3, Gnucap y Xyce.

Gnucap Simulador de circuitos analógicos y señal mixta, está diseñado para reemplazar Spice, pero con ventajas técnicas significantes. Más rápido e igual de preciso, diseñado para alta flexibilidad por medio de un sistema de plugins, algoritmos modernos, relación de compromiso entre precisión y velocidad controlado por el usuario, totalmente interactivo por medio de *scripting*. Puede leer *netlists* en formato tradicional *spice 3*, Spectre¹ y *netlist* Verilog.

Ngspice Simulador de circuitos de señal mixta. Se basa en tres paquetes de software libre: Spice3f5, Cider1b1 y Xpice. Implementa muchas mejoras y corrección de *bugs*. Fué ampliamente incorporado en otros entornos de simulación de circuitos.

¹Motor de simulación de Cadence.

Brent-Kung	8			16				32			
filas	3	4	5	4	5	6	7	6	7	8	9
ancho	1386	1090	945	2268	1757	1545	1429	3196	1983	2569	2424
alto	746	910	1199	1255	1436	1540	1959	2024	2871	2927	2882
área	1033956	991900	1133055	2846340	2523052	2379300	2799411	6468704	5693193	7519463	6985968
ancho/alto	0,54	0,83	1,27	0,55	0,82	1,00	1,37	0,63	1,45	1,14	1,19

Cuadro 5.5: Ubicación y conexionado para Brent-Kung en 3 tamaños: 8, 16 y 32 bits

Xyce Simulador de circuitos con compatibilidad Spice, con mejoras para lograr simulaciones de millones de transistores con la misma precisión. Implementa alto nivel de paralelismo, *solvers* iterativos mejorados y simula efectos de radiación (corriente de fotones y destrucción de neutrones).

Medición de la performance

Medición de la potencia

5.6.2. Tablas comparativas de performance, potencia y área

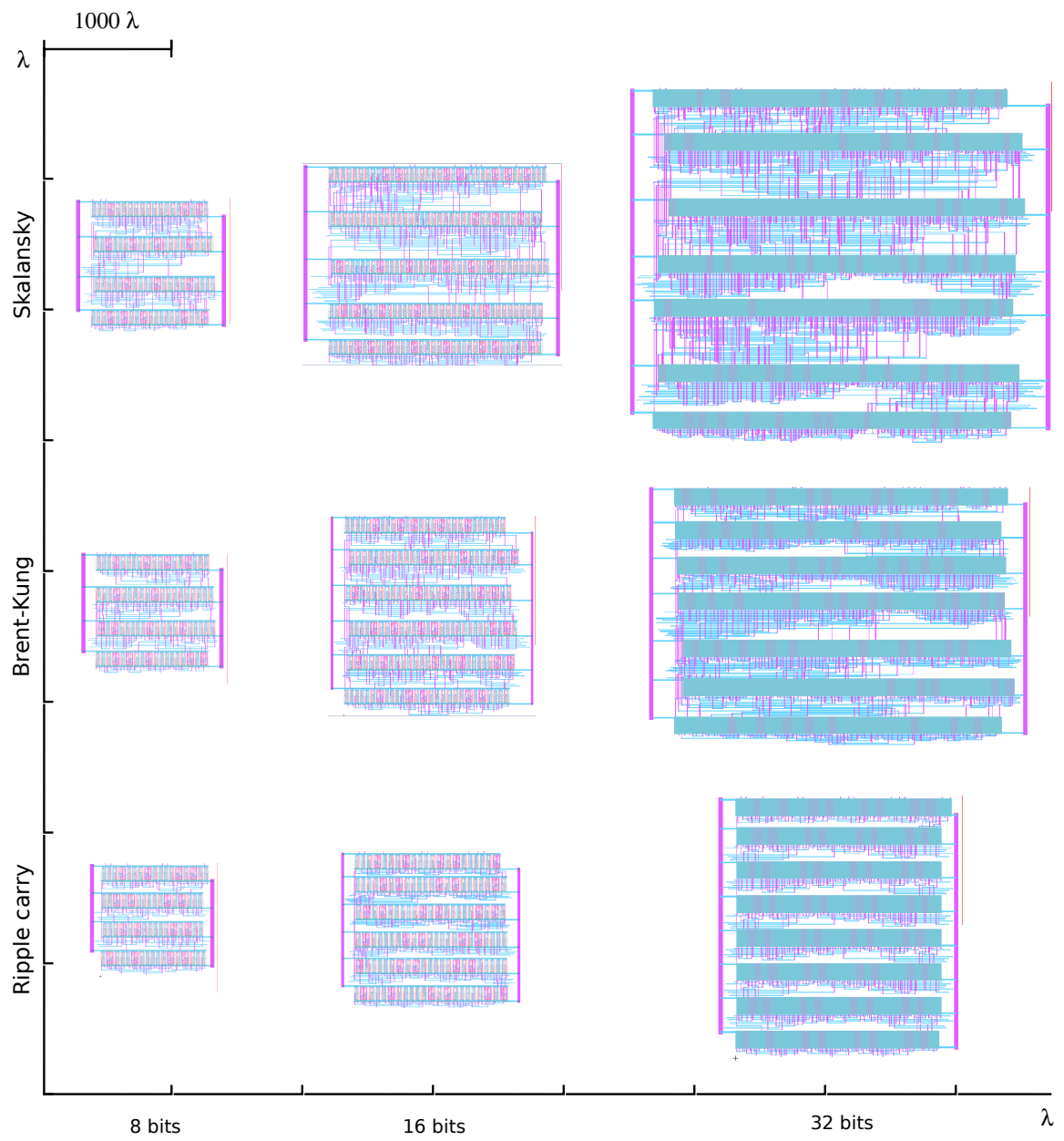


Figura 5.7: Tres arquitecturas y tres tamaños de sumandos distintos. El gráfico está en escala, la unidad de los dos ejes es λ

Capítulo 6

Sign Out y Tape Out

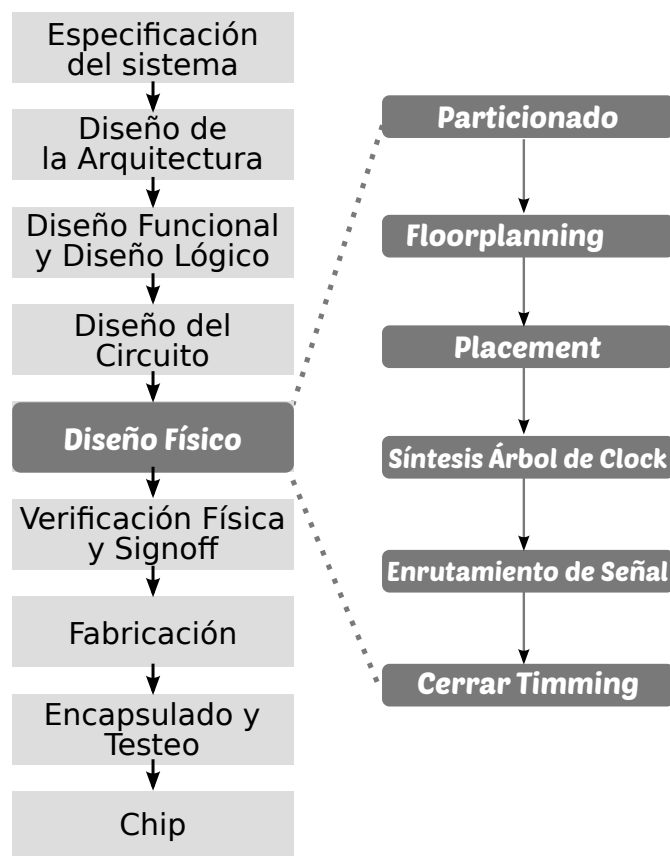


Figura 6.1: Flujo de diseño Físico

...

Parte III

Comparación de resultados

Capítulo 7

Comparación de resultados

7.1. Tablas comparativas

Parte IV

Conclusiones

Capítulo 8

Conclusiones

...

Apéndice A

NETLIST VHDL

```
library ieee;

use ieee.std_logic_1164.all;

entity
    BrentKungFastAdder
is
port
    (

        a_0 : in std_logic
    ; a_1 : in std_logic
    ; a_2 : in std_logic
    ; a_3 : in std_logic
    ; a_4 : in std_logic
    ; a_5 : in std_logic
    ; a_6 : in std_logic
    ; a_7 : in std_logic
    ; b_0 : in std_logic
    ; b_1 : in std_logic
    ; b_2 : in std_logic
    ; b_3 : in std_logic
    ; b_4 : in std_logic
    ; b_5 : in std_logic
    ; b_6 : in std_logic
    ; b_7 : in std_logic

    ; sum_0 : out std_logic
    ; sum_1 : out std_logic
    ; sum_2 : out std_logic
    ; sum_3 : out std_logic
    ; sum_4 : out std_logic
    ; sum_5 : out std_logic
    ; sum_6 : out std_logic
    ; sum_7 : out std_logic
    ; cout : out std_logic
    );
```

```

end BrentKungFastAdder;

architecture
    structural
of
    BrentKungFastAdder
is
    signal w1 : std_logic;
    signal w2 : std_logic;
    signal w3 : std_logic;
    signal w4 : std_logic;
    signal w5 : std_logic;
    signal w6 : std_logic;
    signal w7 : std_logic;
    signal w8 : std_logic;
    signal w9 : std_logic;
    signal w10 : std_logic;
    signal w11 : std_logic;
    signal w12 : std_logic;
    signal w13 : std_logic;
    signal w14 : std_logic;
    signal w15 : std_logic;
    signal w16 : std_logic;
    signal w17 : std_logic;
    signal w18 : std_logic;
    signal w19 : std_logic;
    signal w20 : std_logic;
    signal w21 : std_logic;
    signal w22 : std_logic;
    signal w23 : std_logic;
    signal w24 : std_logic;
    signal w25 : std_logic;
    signal w26 : std_logic;
    signal w27 : std_logic;
    signal w28 : std_logic;
    signal w29 : std_logic;
    signal w30 : std_logic;
    signal w31 : std_logic;
    signal w32 : std_logic;
    signal w33 : std_logic;
    signal w34 : std_logic;
    signal w35 : std_logic;
    signal w36 : std_logic;
    signal w37 : std_logic;
    signal w38 : std_logic;
    signal w39 : std_logic;
    signal w40 : std_logic;
    signal w41 : std_logic;
    signal w42 : std_logic;
    signal w43 : std_logic;
    signal w44 : std_logic;
    signal w45 : std_logic;
    signal w46 : std_logic;
    signal w47 : std_logic;
    signal w48 : std_logic;
    signal w49 : std_logic;

```

```

signal w50 : std_logic;
signal w51 : std_logic;
signal w52 : std_logic;
signal w53 : std_logic;
signal w54 : std_logic;
signal w55 : std_logic;
signal w56 : std_logic;
signal w57 : std_logic;
signal w58 : std_logic;
signal w59 : std_logic;
signal w60 : std_logic;
signal w61 : std_logic;
signal w62 : std_logic;
signal w63 : std_logic;
signal w64 : std_logic;
signal w65 : std_logic;
begin
  c_w2      : wire port map (a_0, w2);
  c_w3      : wire port map (b_0, w3);
  c_w1      : xor2 port map (w2, w3, w1);
  c_w6      : wire port map (a_1, w6);
  c_w7      : wire port map (b_1, w7);
  c_w5      : xor2 port map (w6, w7, w5);
  c_w8      : and2 port map (w2, w3, w8);
  c_w4      : xor2 port map (w5, w8, w4);
  c_w11     : wire port map (a_2, w11);
  c_w12     : wire port map (b_2, w12);
  c_w10     : xor2 port map (w11, w12, w10);
  c_w14     : and2 port map (w6, w7, w14);
  c_w15     : and2 port map (w5, w8, w15);
  c_w13     : or2  port map (w14, w15, w13);
  c_w9      : xor2 port map (w10, w13, w9);
  c_w18     : wire port map (a_3, w18);
  c_w19     : wire port map (b_3, w19);
  c_w17     : xor2 port map (w18, w19, w17);
  c_w21     : and2 port map (w11, w12, w21);
  c_w22     : and2 port map (w10, w13, w22);
  c_w20     : or2  port map (w21, w22, w20);
  c_w16     : xor2 port map (w17, w20, w16);
  c_w25     : wire port map (a_4, w25);
  c_w26     : wire port map (b_4, w26);
  c_w24     : xor2 port map (w25, w26, w24);
  c_w29     : and2 port map (w18, w19, w29);
  c_w30     : and2 port map (w17, w21, w30);
  c_w28     : or2  port map (w29, w30, w28);
  c_w32     : and2 port map (w17, w10, w32);
  c_w31     : and2 port map (w32, w13, w31);
  c_w27     : or2  port map (w28, w31, w27);
  c_w23     : xor2 port map (w24, w27, w23);
  c_w35     : wire port map (a_5, w35);
  c_w36     : wire port map (b_5, w36);
  c_w34     : xor2 port map (w35, w36, w34);
  c_w38     : and2 port map (w25, w26, w38);
  c_w39     : and2 port map (w24, w27, w39);
  c_w37     : or2  port map (w38, w39, w37);
  c_w33     : xor2 port map (w34, w37, w33);

```

```

c_w42      : wire port map (a_6, w42);
c_w43      : wire port map (b_6, w43);
c_w41      : xor2 port map (w42, w43, w41);
c_w46      : and2 port map (w35, w36, w46);
c_w47      : and2 port map (w34, w38, w47);
c_w45      : or2 port map (w46, w47, w45);
c_w49      : and2 port map (w34, w24, w49);
c_w48      : and2 port map (w49, w27, w48);
c_w44      : or2 port map (w45, w48, w44);
c_w40      : xor2 port map (w41, w44, w40);
c_w52      : wire port map (a_7, w52);
c_w53      : wire port map (b_7, w53);
c_w51      : xor2 port map (w52, w53, w51);
c_w55      : and2 port map (w42, w43, w55);
c_w56      : and2 port map (w41, w44, w56);
c_w54      : or2 port map (w55, w56, w54);
c_w50      : xor2 port map (w51, w54, w50);
c_w60      : and2 port map (w52, w53, w60);
c_w61      : and2 port map (w51, w55, w61);
c_w59      : or2 port map (w60, w61, w59);
c_w63      : and2 port map (w51, w41, w63);
c_w62      : and2 port map (w63, w45, w62);
c_w58      : or2 port map (w59, w62, w58);
c_w65      : and2 port map (w63, w49, w65);
c_w64      : and2 port map (w65, w27, w64);
c_w57      : or2 port map (w58, w64, w57);

c_sum_0    : wire port map (w1, sum_0);
c_sum_1    : wire port map (w4, sum_1);
c_sum_2    : wire port map (w9, sum_2);
c_sum_3    : wire port map (w16, sum_3);
c_sum_4    : wire port map (w23, sum_4);
c_sum_5    : wire port map (w33, sum_5);
c_sum_6    : wire port map (w40, sum_6);
c_sum_7    : wire port map (w50, sum_7);
c_cout     : wire port map (w57, cout);
end structural;

```

Apéndice B

SCRIPT PERL

```
#!/usr/bin/perl

#### Import Classes
use File::Copy;
use FileHandle;
####
#### Define constants
my $idPort = "id";

# Input File
my $file = $ARGV[0];

#
my %ports; # to store ports (ins & outs)
# and wire's name given
# by the VhdlNew.hs

### OPEN INPUT FILE
print "$file\n";
open(my $fhi, '<', $file) or die "archivo no encontrado";
open(my $fho, '+>', "temp-$file");
while(<$fhi>){
#Primero obtengo las entradas
if(m/.$idPort.+port.map.+\\(([A-Za-z0-9_]+).+(w[0-9]+)/)
{
push @wires,$2;
push @wires,$1;
$ports {$2} = $1;
# print $fho "--$_"; # comento la linea
}

#Ahora obtengo las salidas, por ejemplo:
# c_sum_0 : std_wire port map (w1, sum_0);
# o como estas:
# c_cout : std_wire port map (w131, cout);
if(m/.$idPort.+(w[0-9]+)\\.?.?([A-Za-z0-9_]+)\\.*/)
```

```

{
$ports {$1} = $2;
print $fho "--$_"; # comento lo que quiero eliminar
} else {
print $fho "$_";}
# and the outputs
#if(m/([a-z]_[a-z]*_[0-9]+).+out/) {push @outs,$1;}
# I could use ins and outs to make the %ports hash table
}#while
close($fhi);
close($fho);

#copy("temp-$file", "temp2-$file");
# Replace all signals connected to wire with the inputs
#   c_w131      : std_or2   port map (w132, w141, w131);
# w131 should be replaced by the output

while(my($key,$value) = each(%ports)) {
open(my $fhi, '<', "temp-$file") or die "archivo no encontrado";
open(my $fho, '+>', "stripped-$file");
while(<$fhi){
if(s/(.+map.+)$key(\,|\))/ $1$value$2/g) {

#need to delete entries with the next pattern:  c_w18      : std_wire  port map ...
if(m/.$idPort.+/) {print $fho "-- deleted $_";}
else {print $fho "$_";}
else { print $fho "$_";}
    }# while file

close($fho);
close($fhi); #atenti no hacer close($fho, $fhi) porque no es lo mismo que en 2 reng
copy("stripped-$file", "temp-$file");
}#while hash table

```


Bibliografía

- [1] MANUEL VALENCIA ADRIÁN ESTRADA, CARLOS J. JIMÉNEZ. Características de Sumadores en Tecnologías Fuertemente Submicrónicas. *IBERCHIP, DOC (TEC 2004-01509/MIC)*, 1982. [13](#)
- [2] A. BALIGA AND D. YAGAIN. Design of high speed adders using cmos and transmission gates in submicron technology: A comparative study. In *Emerging Trends in Engineering and Technology (ICETET), 2011 4th International Conference on*, pages 284–289, 2011. [13](#)
- [3] D. BLAAUW, J. KITCHENER, AND B. PHILLIPS. Optimizing addition for sub-threshold logic. In *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pages 751–756, Oct 2008. [35](#)
- [4] R. P. BRENT AND H. T. KUNG. A Regular Layout for Parallel Adders. *IEEE Transaction on Computers*, **C-31**, Issue: 3:260–264, 2006. [13](#), [14](#), [15](#)
- [5] K. CLAESSEN AND M. SHEERAN. A tutorial on Lava: A hardware description and verification system. Website, 2014. <http://projects.haskell.org/chalmers-lava2000/Doc/>. [18](#)
- [6] STEVE RUBIN ET AL. Electric VLSI design system. <http://www.staticfreesoft.com/>. [32](#)
- [7] J. HENNESSY AND D. PATTERSON. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition edition, 2007. [5](#)
- [8] S. KNOWLES. A family of adders. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 277–281, 2001. [13](#)
- [9] PETER M. KOGGE AND HAROLD S. STONE. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, **C-22**(8):786–793, Aug 1973. [13](#), [14](#)
- [10] RICHARD E. LADNER AND MICHAEL J. FISCHER. Parallel prefix computation. *JACM, Journal of the ACM*, **C-27**(4):831,838, Oct 1980. [14](#)
- [11] L. MARSO. Brent-kung fast adder description, simulation and formal verification using lava. In *Micro-Nanoelectronics, Technology and Applications, 2008. EAMTA 2008. Argentine School of*, pages 111–114, Sept 2008. [18](#)
- [12] CARVER MEAD AND LYNN CONWAY. *Introduction to VLSI Systems*. Addison-Wesley, 1980. [31](#)
- [13] B. PARHAMI. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000. [10](#)

-
- [14] J.M. RABAEY, A.P. CHANDRAKASAN, AND B. NIKOLIC. *Digital integrated circuits: a design perspective*. Prentice Hall electronics and VLSI series. Pearson Education, 2ed edition, 2003. 3, 5
 - [15] MINGOO SEOK, S. HANSON, YU-SHIANG LIN, ZHIYOONG FOO, DAEYEON KIM, YOONMYUNG LEE, N. LIU, D. SYLVESTER, AND D. BLAAUW. The phoenix processor: A 30pw platform for sensor applications. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 188–189, June 2008. 34
 - [16] M. SHEERAN. Parallel prefix network generation: an application of functional programming In Hardware Design and Functional Languages. In *Hardware design and Functional Languages (HFL07)*, Braga, Portugal, March 2007. 19
 - [17] LIP6 UNIVERSITÉ PIERRE ET MARIE CURIE. Alliance VLSI CAD System. <https://soc-extras.lip6.fr/en/alliance-abstract-en/>. 32
 - [18] CLIFFORD WOLF. Yosys open synthesis suite. <http://www.clifford.at/yosys/>. 32