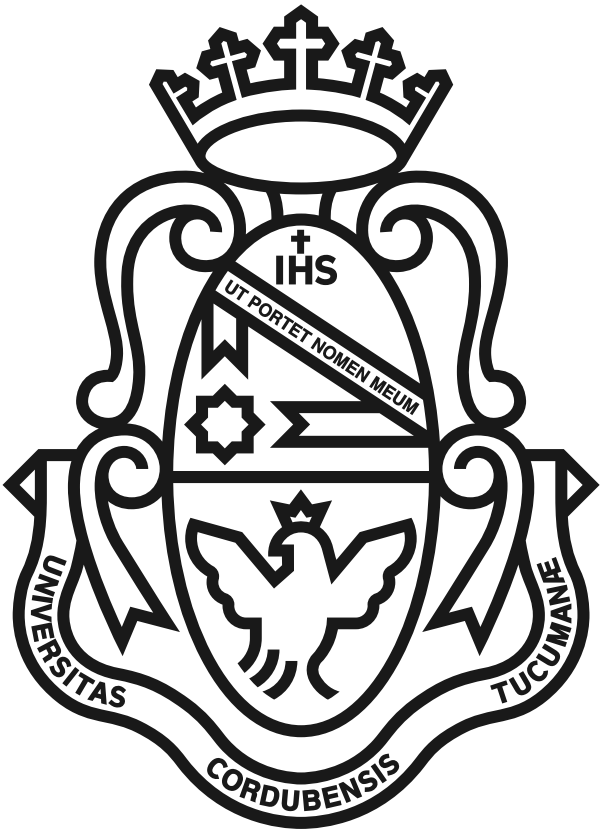
Universidad Nacional de Córdoba
Facultad de Ciencias Exáctas, Físicas y Naturales



Proyecto Integrador de la Carrera Ingeniería Electrónica

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

Alumno: Leandro Marsó
Director: Ing. Pablo O. Cayuela
Codirector: Ing. Hugo S. Carrer

Noviembre 2014

Resumen

El presente trabajo aborda el problema del diseño de un circuito presente en la mayoría de los sistemas digitales, el sumador binario. Se realiza utilizando únicamente herramientas de software libre y en una tecnología que nos permite gran escala de integración de transistores de efecto de campo y elementos pasivos, todos en una pastilla de silicio de dimensiones milimétricas.

Índice general

Índice general I

Índice de figuras V

Índice de cuadros VII

Glossary IX

1. INTRODUCCIÓN XI
1.1. Estructura del Proyecto Integrador . . . XI
1.2. Planteamiento del problema y motivación . . . XI
1.3. Objetivos generales . . . XII
1.4. Objetivos particulares . . . XII
1.5. Plan de Trabajo . . . XIII

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. SUMADORES 7
3.1. Introducción . . . 7
3.1.1. Selección de la arquitectura . . . 7
3.2. Fundamentos teóricos de la suma . . . 8
3.2.1. Semisumador y sumador completo . . . 9
3.2.2. Sumadores *carry lookahead* . . . 10
3.2.3. Desenrollando la recurrencia del acarreo . . . 11
3.2.4. Sumadores de Prefijo Paralelos (*Parallel Prefix Adders*) . . . 11
3.2.5. Sumador de Brent-Kung . . . 13
3.2.6. Sumador de Sklansky . . . 15

4. IMPLEMENTACIÓN DE LOS CIRCUITOS EN LENGUAJE DE DESCRIPCIÓN DE HARDWARE **19**
4.1. Introducción . . . 19
4.1.1. Breve reseña de los Hardware Description Language (HDL) . . . 19
4.1.2. Nuevos HDL . . . 20
4.1.3. ¿Por qué Lava? . . . 20
4.2. Implementación en lenguaje de descripción de hardware . . . 21
4.2.1. Implementación del sumador de **ripple carry** en Lava . . . 21
4.2.2. Patrones de conexión . . . 21
4.2.3. Sumador de Brent-Kung . . . 23
4.2.4. Sumador de Sklansky . . . 25
4.2.5. Simulación . . . 25
4.2.6. Síntesis del *netlist* VHDL . . . 26

5. VERIFICACIÓN FORMAL **27**
5.1. Modelo de referencia . . . 27
5.2. Verificación de las Propiedades . . . 28
5.2.1. Propiedades de la suma . . . 28
5.2.2. Descripción de las propiedades en el Ripple Carry Adder . . . 28
5.2.3. Equivalencia lógica entre el modelo de referencia y los sumadores de Brent-Kung y Sklansky . . . 29

II Diseño Físico **31**

6. Flujo de Diseño Físico **33**
6.1. Introducción . . . 33
6.1.1. Etapas del diseño físico . . . 33
6.2. Relevamiento, comparación y selección de las herramientas disponibles . . . 35
6.2.1. Relevamiento . . . 36
6.2.2. Comparación . . . 36
6.2.3. Selección . . . 37
6.3. Selección del proceso de fabricación . . . 37
6.3.1. Obleas multiproyectos . . . 38
6.3.2. *Corners* de simulación . . . 40
6.4. Selección de las Celdas estándar . . . 40
6.4.1. Características . . . 41
6.5. Ubicación y Cableado (*Place & Route*) . . . 42
6.5.1. Modificación al código fuente de la herramienta de generación del *netlist* VHDL . . . 42
6.5.2. Configuración de la herramienta de Place & Route (PnR) . . . 43
6.5.3. Distintas alternativas y resultados . . . 44
6.6. Comparación de las distintas arquitecturas . . . 46
6.6.1. Simulación post *layout* para calcular performance y potencia . . . 46
6.6.2. Medición de la performance en un sumador . . . 48
6.6.3. Medición de la potencia en un sumador . . . 49
6.6.4. Potencia y performance de todas las arquitecturas . . . 50

6.6.5. Comparación de performance, potencia y área 51

III Conclusiones **53**

7. Conclusiones finales **55**
7.1. Metodología 55
7.2. Resultados 56
7.3. Aplicaciones 56
7.4. Desafíos futuros 57

A. NETLIST VHDL **59**

B. LIBRERÍA DE LAVA PARA GENERAR NETLIST VHDL **63**

C. SCRIPT PERL **69**

D. DESCRIPCION EN LAVA DE LOS SUMADORES SKLANSKY Y BRENT-KUNG **71**
D.1. Sumador de Sklansky 71
D.2. Sumador de Brent-Kung 72

E. BANCO DE PRUEBA PARA SIMULACIONES EN GNUCAP **75**
E.1. Análisis de potencia de los sumadores 75
E.2. Análisis de performance en los sumadores 77

F. SCRIPTS PARA GRAFICAR LAS FORMAS DE ONDAS **79**

Bibliografía **81**

Índice de figuras

2.1. Retardo de propagación de un inversor 4
2.2. Estimación de Potencia Promedio Disipada 5

3.1. Retardo respecto al tamaño de los operandos 8
3.2. Área respecto al tamaño de los operandos 8
3.3. Bit adders 9
3.4. Ripple Carry Adder 10
3.5. CLA 4-bits 12
3.6. Red de prefijo en serie, una forma gráfica de ver la ecuaciones del cálculo prefijo en 3.8. Notar que cada línea representa dos bits. Los puntos negros representan al operador acarreo. 14
3.7. Operator Punto de Brent-Kung 14
3.8. Generación y Propagación del Acarreo 15
3.9. Sumador de prefijo paralelo 16
3.10. Red de prefijo paralelo para Brent-Kung (ejemplo de 16 bits) 16
3.11. Sumador de Brent-Kung 17
3.12. Red de prefijo paralelo para Sklansky (ejemplo de 16 bits) 17

4.1. Diferentes patrones de conexión de circuitos 22
4.2. Construcción de la red de prefijos paralelos de Brent-Kung. 23

5.1. Circuito que nos sirve para describir la suma del elemento neutro 29

6.1. Flujo de diseño Físico 34
6.2. Representación en tres dimenciones de una celda estándar con 3 capas de metales en color arena, y una capa de silício policristalino en color ladrillo. Azul y rojo son dopado N^+ y P^+ respectivamente 35
6.3. Mapeo de una función lógica a una celda estándar 41
6.4. Grilla de interconexionado y riel de alimentación de las celdas estándar de 128λ. Por encima de cada celda, pueden pasar 16 pistas horizontales que la herramienta de conexionado tendrá a disposición, a partir del metal 3 para arriba. Notar la separación de $8\ \lambda$ para todas las pistas horizontales, y $8\ \lambda$ 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*. 42
6.5. Conjunto de celdas estándar 43
6.6. Configuración del Silicon Compiler de Electric 43

6.7. Tres arquitecturas y tres tamaños de sumandos distintos. Los circuitos están en escala, la unidad de los dos ejes es $\lambda = 90$ nm 45
6.8. Configuración de Electric: Especificar los modelos de transistores a utilizar por el motor tipo Spice. 46
6.9. Oscilador anillo de 31 etapas 48
6.10. Simulación de régimen transitorio del circuito Ripple Carry 8 bits. De los vectores de entradas, sólo mostramos el bit menos significativo de la entrada A, porque es el único que cambia. Las señales de salidas están ordenadas para mostrar la más lenta cerca de la entrada y poder calcular el retardo. 49
6.11. Simulación de régimen transitorio del circuito Ripple Carry 8 bits. Para calcular t_{pHL} y t_{pLH} lo hacemos sobre la señal más lenta del circuito, que en este caso es el bit más significativo de la suma 50
6.12. Simulación de régimen transitorio del circuito Ripple Carry 8 bits. Mostramos la corriente instantánea a través de la fuente de alimentación, el valor negativo se debe a que la corriente sale de la fuente. 50

7.1. Flujo de diseño analógico. 57

Índice de cuadros

2.1. Especificaciones de diseño para el sumador binario 3
2.2. Métricas de comparación 6

3.1. Resumen de las funciones de retardo y área algunos sumadores 9

6.1. Procesos disponibles por medio de MOSIS 38
6.2. Procesadores fabricados en CMOS 180nm 39
6.3. Ubicación y conexionado para Ripple carry en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente. 44
6.4. Ubicación y conexionado para Skalanksy en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente. 44
6.5. Ubicación y conexionado para Brent-Kung en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente. 44
6.6. Simulación post *layout* del oscilador anillo de 31 etapas 48
6.7. Comparación de los resultados de las 3 arquitecturas 51

Glossary

CMP Organización que presta el servicio para la fabricación de circuitos integrados y MEMS para prototipado y bajo volumen de producción. Sitio web: http://cmp.imag.fr/. XII

CTS Clock Tree Synthesis. 34

DRC Design Rule Check. 36, 43, 57

HDL Hardware Description Language. II, 19, 56

IDMs Integrated Device Manufacturers. 38

LVS Layout Vs. Schematic. 36, 57

MEMS Sistemas Micro-Electro-Mecánicos, por sus sigas en inglés. IX

PDK Process Design Kit. 36

PnR Place & Route. II, 34, 36, 37, 42–44, 56, 57

SPICE Simulation Program with Integrated Circuit Emphasis. 57

STA Static Timing Analisys. 36, 37, 55, 57

VHDL VHSIC Hardware Description Language. II, 19–21, 26, 56

VHSIC Very High Speed Integrated Circuit. IX

Capítulo 1
INTRODUCCIÓN

En el presente capítulo se describe de forma general la estructura y contenido de todo el trabajo. Detallamos el planteo del problema y la motivación para llevarlo acabo. Luego definimos los objetivos generales y particulares, y enunciamos el plan de trabajo que nos hemos propuesto.

1.1. Estructura del Proyecto Integrador

Parte I - Diseño digital: Definimos las métricas que utilizaremos para seleccionar calificar la implementación de los sumadores. Seleccionamos tres arquitecturas para implementar y desarrollamos los fundamentos teóricos de estas implementaciones. Seleccionamos un lenguaje de descripción de *hardware*, implementamos los circuitos, simulamos y verificamos formalmente los mismo. Generamos tres *netlist* VHDL de cada arquitectura para 3 tamaños de sumandos distintos: 8, 16 y 32 bits.

Parte II - Implementación Física: Explicamos el flujo de diseño físico, seleccionamos herramientas de software para realizarlo, seleccionamos una tecnología de fabricación, creamos nuestras celdas estándar y realizamos de forma automática el *layout* del circuito. Simulamos los circuitos para obtener las mediciones de nuestras métricas de calidad y comparamos todas las implementaciones.

Parte III - Conclusiones: Realizamos un resumen de lo que se logró, haciendo énfasis en la metodología, los resultados y las aplicaciones. Planteamos mejoras a nuestro trabajo y desafíos futuros para el diseño de circuitos integrados.

1.2. Planteamiento del problema y motivación

En la actualidad los microprocesadores, DSP, microcontroladores, y otro hardware específico para cálculo computacional o de aplicación específica son desarrollados en tecnología CMOS submicrónica. El problema planteado es, ¿Cómo hacer para diseñar circuitos integrados en ésta tecnología, con herramientas flexibles, libres[1] y accesibles para todo tipo de uso: académico y comercial?.

Nos interesa este problema por varios motivos, pero el principal es poder acercar estas tecnologías sin restricciones a los estudiantes de grado. Las posibilidades de aplicaciones de los

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

circuitos integrados son infinitas, ya que podemos controlar todas las dimensiones físicas de los transistores y elementos pasivos que integramos. Utilizando software libre, también existe la posibilidad de adaptar o mejorar las herramientas que utilizamos para el diseño.

Otra motivación importante es la económica: Los costos por licencias de las herramientas de software, pueden transformar un proyecto de bajo volumen de producción en económicamente no factible. Damos un ejemplo real:

> El costo de 25 chips en tecnología de 0.35 μm con un área máxima de 3mm^2, con encapsulado DIP-20 fabricados por medio de CMP[1] es de €2787.5, resultando el costo de fabricación en €111.5 por unidad. Pero si tenemos en cuenta el costo de las licencias de software por un año, el costo total se incrementa en un orden de magnitud, como mínimo[2].

En función del problema planteado y de las motivaciones mencionadas, nos propusimos implementar un sumador binario ya que es un circuito presente en la mayoría de los sistemas, que además puede ser el bloque de mayor requerimiento de performance.

1.3. Objetivos generales

El objetivo del proyecto es diseñar un sumador de n-bits, que pueda ser utilizado en un sistema digital, especificando las características más importantes que nos permitan determinar su utilización según requerimientos de tamaño en bits, performance, potencia y área. Lograremos un diseño paramétrico según la cantidad de bits, que podrá ser usado en unidades aritméticas o formar parte de un sistema de procesamiento de señales digitales. Integrar y documentar una metodología de diseño utilizando herramientas de software libre también es un objetivo de este trabajo, para que pueda ser reproducida, mejorada o adaptada a las necesidades de futuros proyectos que se planteen el diseño de circuitos integrados.

1.4. Objetivos particulares

El objetivo general de este proyecto integrador se puede lograr si nos planteamos los siguientes objetivos particulares:

- Estudiar las técnicas actuales para implementar sumadores rápidos y definir métricas de calidad.
- Relevar las herramientas de software disponibles para hacer diseño digital y diseño físico, y las tecnologías de fabricación de circuitos integrados.
- Implementar la suma con una arquitectura básica para referencia y comparación, y dos arquitecturas rápidas. Implementar los circuitos en un lenguaje de descripción de hardware y realizar la implementación física.

[1] CMP es una organización que presta el servicio para la fabricación de circuitos integrados y MEMS para prototipado y bajo volumen de producción

[2] El costo de licencia de un conjunto completo de herramientas por un año puede llegar hasta $1.000.000 (dólares norteamericanos). Este tipo de información no está públicamente disponible.

- Crear *scripts* que nos permitan pasar de una etapa a la otra del diseño de forma automática, realizando las modificaciones necesarias para adaptar la salida de un proceso a la entrada del siguiente.
- Verificar su funcionamiento y extraer métricas de calidad para poder compararlas.

1.5. Plan de Trabajo

Se estableció el siguiente plan de trabajo para llevar adelante el proyecto:

1. Comprender los requerimientos del diseño y fabricación de los circuitos integrados, y la relación con las herramientas de software.
2. Diseñar cada una de las arquitecturas a implementar, simularlas y verificar su correcto funcionamiento.
3. Realizar bancos de prueba (*test benches*) para poder generar las métricas de calidad.
4. Generar conclusiones a partir de los resultados y proponer desafíos futuros.

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(20)**. 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 perpectiva se esté realizando el análisis. Pero si nos enfocamos puramente en el diseño, la performance en

los sistemas síncronos se define usualmente(20) como la duración del período del reloj (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 camíno 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.

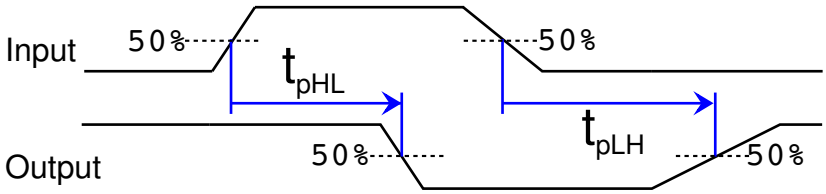


Figura 2.1: Retardo de propagación de un inversor

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 oscilla espontaneamente, 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 $t_p = 47,8\ ps$

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.

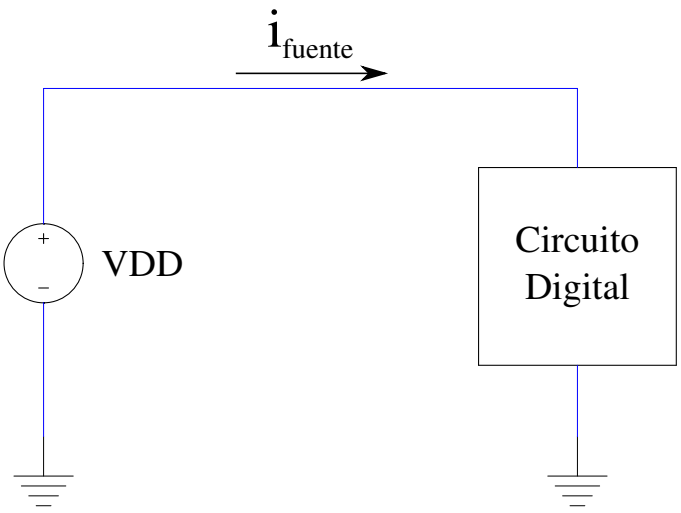


Figura 2.2: Estimación de Potencia Promedio Disipada

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

El período de tiempo que tomaremos para medir la potencia promedio lo elegiremos de forma tal que la medición resulte representativa del funcionamiento del circuito.

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*(12), ya que el costo es una función que depende de la cuarta potencia del área del circuito(20). 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:

Métrica	**Unidades**
Retardo de propagación	[ns]
Potencia promedio disipada	[mW]
Área de circuito	[μm^2]

Cuadro 2.2: Métricas de comparación

Capítulo 3

SUMADORES

3.1. Introducción

Tal como mencionamos en la sección 2.1 (pág. 3), nuestro objetivo es implementar un sumador binario de n bits, manteniendo la mejor relación de compromiso entre performance, potencia y área según crece n.

3.1.1. Selección de la arquitectura

Se puede afirmar que los sumadores llamados (según la bibliografía en inglés) como *parallel prefix adders*[1] son los mejores con respecto al producto potencia-retardo[2]. Estos sumadores se clasifican dentro de un mismo tipo, porque reducen el problema de calcular las señales de acarreo como el **problema de cálculo de prefijo**[3]. A su vez, son implementaciones particulares de los sumadores conocidos como *carry look-ahead adders*, ya que todos se basan en el cálculo en paralelo de los acarreos.

Sumadores de prefijo paralelo

Brent-Kung(5), Sklansky(24), Kogge-Stone (14), Ladner-Fisher(15), Hans-Carlson(14) y Knowles(13) son implementaciones de este tipo de sumadores, que se diferencian cada uno por minimizar alguna relación de compromiso, en el espacio de diseño para el retardo, área y potencia(25) del circuito.

Si tenemos en cuenta el área utilizada por estos circuitos, no podemos asegurar que una de estas se clasifique globalmente como la mejor, ya que algunas implementaciones favorecen una métrica a costa de la otra. Citamos un estudio que presenta los siguientes resultados de la figuras 3.1 y 3.2 de un estudio comparativo (1) para tecnología CMOS 0.13 μm .

Arquitecturas a implementar

De estas arquitecturas mencionadas, vamos a implementar un sumador rápido basado en la idea original de Sklansky(24) publicado en 1960. También vamos a implementar una arquitectura

[1] Nosotros los nombraremos como *sumadores de prefijo paralelo*.

[2] Cuando decimos retardo, nos referimos al retardo de propagación máximo de un circuito, nuestra métrica elegida para caracterizar la performance.

[3] En 3.2.4 definimos precisamente el problema.

que busca la mejor relación entre interconexiones y cantidad de compuertas utilizadas, a costa de un pequeño aumento en la cantidad de etapas, conocido como sumador de Brent-Kung(5). Además, implementaremos el sumador de ripple carry para utilizarlo de referencia comparativa.

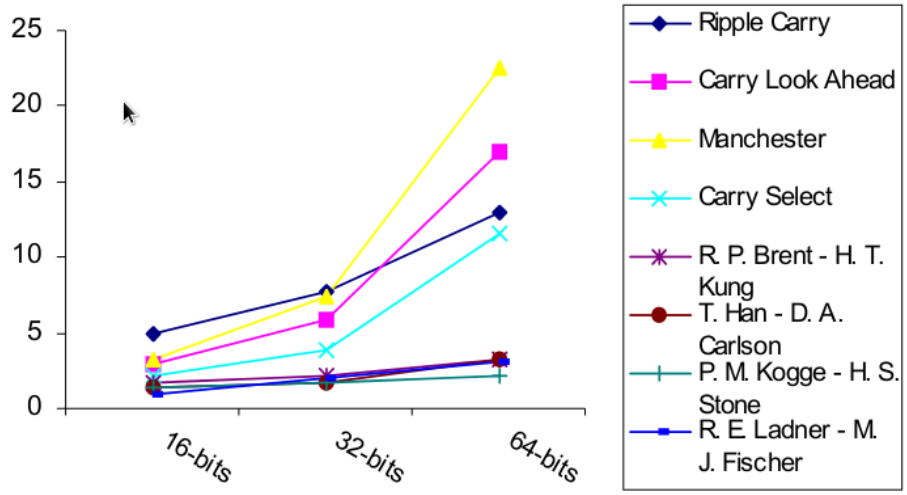


Figura 3.1: Retardo respecto al tamaño de los operandos

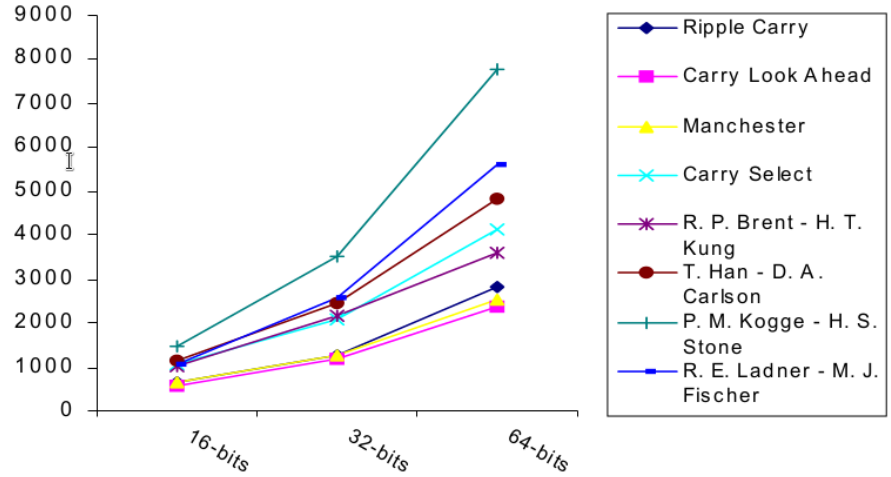


Figura 3.2: Área respecto al tamaño de los operandos

Resumimos en la tabla 3.1 las características y diferencias entre los distintos sumadores(3). Incluimos al **ripple carry**, por ser la implementación más simple, y al **carry look-ahead** por ser el sumador que propone el cálculo en paralelo de los acarreos para disminuir logarítmicamente el tiempo de retardo.

3.2. Fundamentos teóricos de la suma

A los fines de poder implementar estos sumadores, desarrollaremos las ecuaciones que nos permitan llegar a la descripción del *hardware*.

Arquitectura	Retardo Máx.	Área
Ripple Carry	$O(n)$	$O(n)$
Carry Look-Ahead	$O(\log(n))$	$O(n\log(n))$
Ladner-Fisher	$O(\log_2(n))$	$O(n\log(n))$
Sklansky	$O(\log_2(n))$	$O(n\log^2(n))$
Kogge-Stone	$O(\log_2(n))$	
Han-Carlson	...	...
Brent-Kung	$O(\log_2(n))$	$O(n\log_2(n))$

Cuadro 3.1: Resumen de las funciones de retardo y área algunos sumadores

3.2.1. 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.

$$s = a \oplus b \tag{3.1a}$$
$$c = ab \tag{3.1b}$$

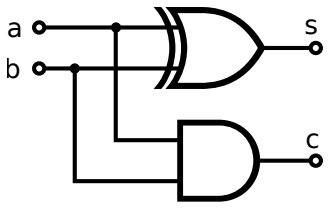
Sumador Completo

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

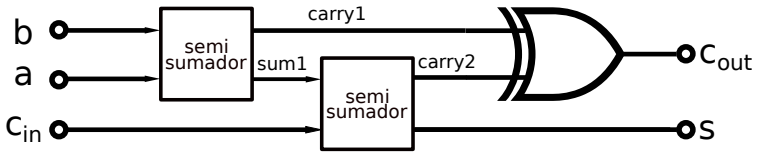
Entradas:	Bits de operandos a , b y carry-in c_{in}	(o a_i, b_i, c_i para la etapa i)
Salidas:	Suma s y carry-out c_{out}	(o s_i y c_{i+1} para la etapa i)

$$s = a \oplus b \oplus c_{in} \tag{3.2a}$$
$$c_{out} = ab + ac_{in} + bc_{in} \tag{3.2b}$$

Podemos construir un **sumador completo** (full-adder) combinando las ecuaciones del sumador y semisumador, como vemos en la figura 3.3b:



(a) Semisumador



(b) Sumador completo

Figura 3.3: Bit adders

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 3.4 el bloque `fullAdd` (Sumador Completo). El camino crítico de la señal se determina considerando el peor camino de propagación de la señal.

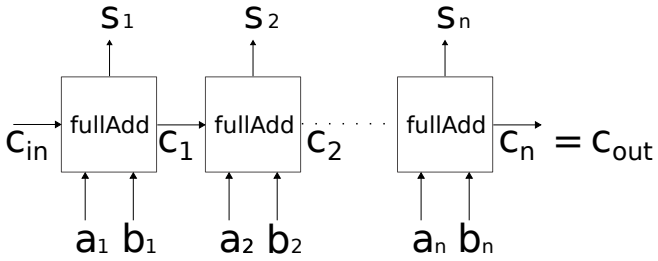


Figura 3.4: Ripple Carry Adder

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

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

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.

3.2.2. Sumadores *carry lookahead*

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 ecuación 3.4. Por lo tanto, el objetivo será lograr un bloque generador de las señales de acarreo de baja latencia(19).

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

$$s_i = a_i \oplus b_i \oplus c_i \tag{3.4}$$

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:

$$c_{i+1} = g_i + c_i p_i \tag{3.5}$$

Esto quiere decir que un acarreo entrará en la etapa $i+1$ si éste se genera en la etapa i, o si entra en la etapa i y se propaga.

3.2.3. 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 $c_0 = c_{\text{in}}$. 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:

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

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, no es necesario aplicar la ecuación 3.6 y lo podemos obtener usando una ecuación más simple, sin tener casi un deterioro en velocidad:

$$c_4 = g_3 + c_3p_3$$

La red de acarreo que resulta de estas ecuaciones la podemos ver en la figura 3.5.

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.

$$T_{CLA} = T_p + T_{AND_n} + T_{OR_n} + T_{FA} \tag{3.7}$$

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 $\log_2 n$.

3.2.4. 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 prefijo paralelo**.

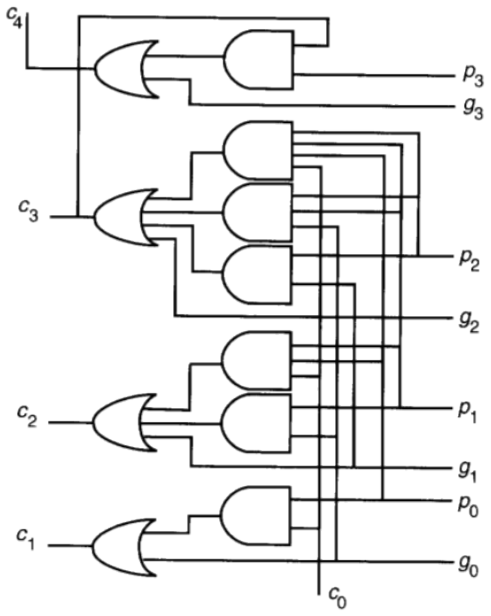


Figura 3.5: CLA 4-bits

Problema de prefijo paralelo (*parallel prefix problem*)

El problema de prefijo paralelo es:

$$\begin{aligned}
&\text{Dado:}\\
&\qquad \text{Entradas:} x_0, x_1, \ldots, x_{k-1}\\
&\qquad \text{Un operador + asociativo}\\
&\text{Computar :} x_0\\
&\qquad x_0 + x_1\\
&\qquad x_0 + x_1 + x_2 +\\
&\qquad \vdots\\
&\qquad x_0 + x_1 + x_2 + \cdots + x_{k-1}
\end{aligned}$$

Cómputo del acarreo como un problema de prefijo paralelo

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

$$\begin{aligned}
g_{[i,i+3]} &= g_{i+3} + g_{i+2}p_{i+3} + g_{i+1}p_{i+2}p_{i+3} + g_i p_{i+1}p_{i+2}p_{i+3}\\
p_{[i,i+3]} &= p_i p_{i+1}p_{i+2}p_{i+3}
\end{aligned}$$

Podemos interpretar estas ecuaciones de la siguiente forma: las cuatro posiciones de bits propagan colectivamente un acarreo c_{in} si y solo sí cada una de las posiciones propaga; y el bloque gener a un acarreo si en la posición $i+3$ se genera uno, o se podrouce 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 $[i_1, j_i]$ y $[i_0, j_0]$, con $i_0 \leq i_1 - 1 \leq j_0 < j_i$:

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

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'),... que nos permiten plantear el problema de la construcción de la red (o bloque) de acarreos, como un *problema de prefijo paralelo*:

$$\begin{aligned}
&\text{Dados:} \\
&\qquad \text{Entradas:} (g_0, p_0), (g_1, p_1), \ldots, (g_{k-1}, p_{k-1}) \\
&\qquad \text{Un operador } \circ \text{ asociativo} \\
&\text{Computar :} \\
&(G_0, P_0) = (g_{[0,0]}, p_{[0,0]}) \\
&(G_1, P_1) = (g_{[0,0]}, p_{[0,0]}) \circ (g_{[0,1]}, p_{[0,1]}) \\
&\qquad \vdots \\
&(G_{k-1}, P_{k-1}) = (g_{[0,0]}, p_{[0,0]}) \circ (g_{[0,1]}, p_{[0,1]}) \circ \ldots \circ (g_{[0,k-2]}, p_{[0,k-2]}) \circ (g_{[0,k-1]}, p_{[0,k-1]})
\end{aligned} \tag{3.8}$$

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 *acarreo* (a veces también mencionado como *operador punto*). La implementación mas básica (y lenta) sería la de ir asociando en serie a este operador, como vemos en la figura 3.6. Todos los sumadores de prefijo paralelo se diferencian escencialmente en la forma de asociar.

3.2.5. Sumador 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 de Brent-Kung(5). Este sumador 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(15), Kugge-Stone(14) y Sklansky(24) que en $\log_2(n)$ etapas calculan todas las señales de acarreo.

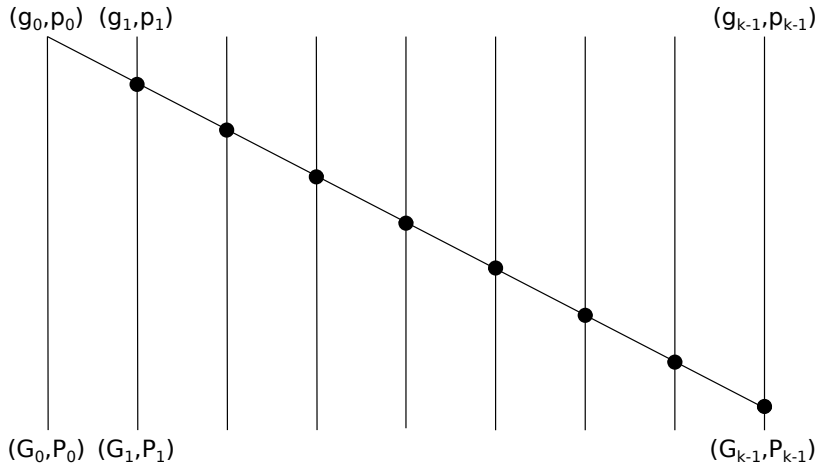


Figura 3.6: Red de prefijo en serie, una forma gráfica de ver la ecuaciones del cálculo prefijo en 3.8. Notar que cada línea representa dos bits. Los puntos negros representan al operador acarreo.

Operador de Brent-Kung

El operador $\circ$ se define[1] como:

$$(g, p) \circ (\hat{g}, \hat{p}) = (g \vee (p \wedge \hat{g}), p \wedge \hat{g}) \tag{3.9}$$

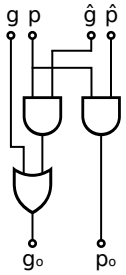


Figura 3.7: Operator Punto de Brent-Kung

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:

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

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.

[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

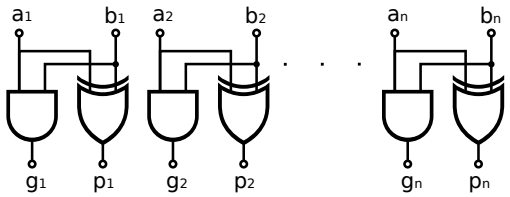


Figura 3.8: Generación y Propagación del Acarreo

Red de Prefijo Paralelo

Con la figura 3.10, detallamos ahora la red de prefijo paralelo con un fan-out máximo de dos, lo cuál diferencia a el sumador de Brent-Kung de los otros sumadores de prefijo paralelo. La red se realiza con 2 elementos: Los puntos negros son los operadores punto de Brent-Kung de la figura 3.7 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 3.11.

Circuito completo

Con la red de prefijo paralelo lista, podemos armar el circuito propuesto en el paper de Brent-Kung(5). A los fines de la implementación en HDL, mostramos el circuito visto de una forma alternativa en la figura 3.11. Pero aprovechamos la oportunidad para generalizar un poco este resultado. Si retomamos la definición de la suma planteada en la ecuación 3.4:

$$s_i = a_i \oplus b_i \oplus c_i$$

y notamos que en la figura 3.11 se calcula $a_i \oplus b_i$, y que con los G_i y los p_{i-1} podemos construir la suma. Por lo tanto, no importa de qué forma se generen estas señales en la red de prefijo paralelo, podemos calcular el valor de s_i, como vemos de forma más genérica en la figura 3.9.

3.2.6. Sumador de Sklansky

Para realizar este sumador, debemos desarrollar la red de cálculo paralelo de los acarreos, a la forma propuesta por Skalansky. Esta forma se conoce como *divide and conquer*, y la evidenciamos con un ejemplo para sumandos de 16 bits en la figura 3.12. Los puntos negros representan el operador punto (definido anteriormente como el operador de Brent-Kung), notar que tiene menos cantidad de etapas de operaciones (en proporción de $\log_2(n)$)) por lo tanto es un sumador más rápido que el de Brent-Kung, pero hay nodos que tienen un fan-out de hasta $\frac{n}{2}$, lo cuál resulta en mayor capacidades parásitas, haciendo más lento el circuito en ese nodo.

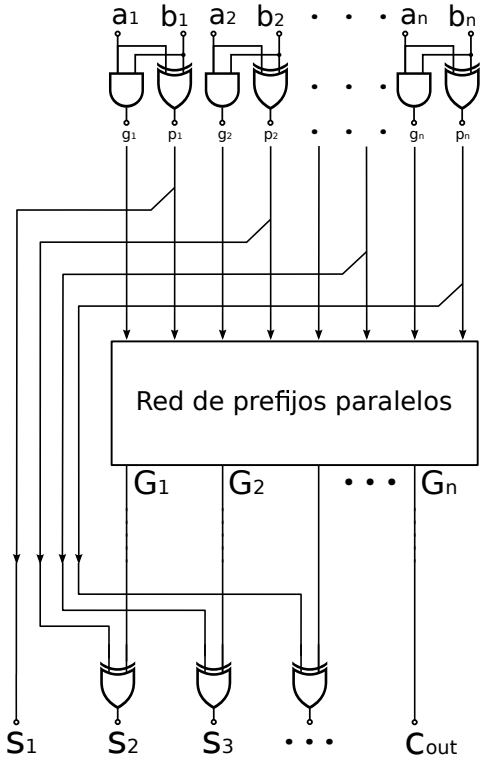


Figura 3.9: Sumador de prefijo paralelo

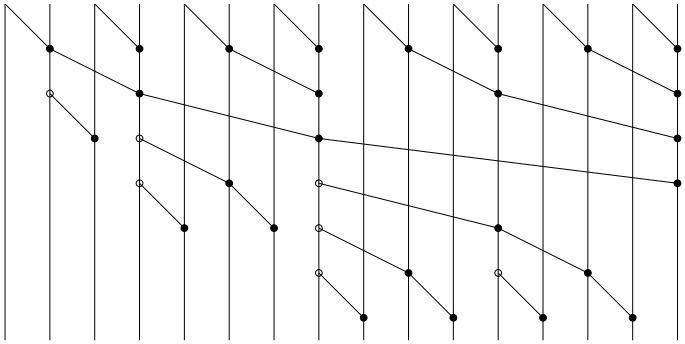


Figura 3.10: Red de prefijo paralelo para Brent-Kung (ejemplo de 16 bits)

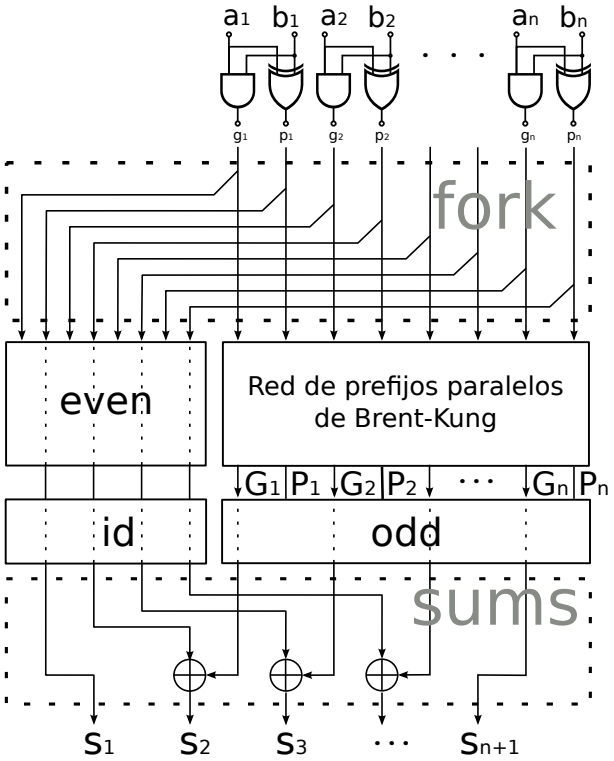


Figura 3.11: Sumador de Brent-Kung

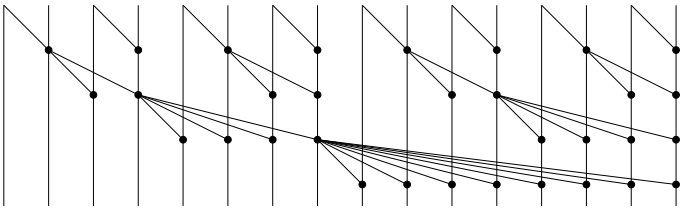


Figura 3.12: Red de prefijo paralelo para Sklansky (ejemplo de 16 bits)

Capítulo 4

IMPLEMENTACIÓN DE LOS CIRCUITOS EN LENGUAJE DE DESCRIPCIÓN DE HARDWARE

4.1. Introducción

Utilizamos un lenguaje de descripción de *hardware* (HDL[1] por su sigla en inglés) para implementar los distintos circuitos que evaluaremos, porque el problema planteado en el capítulo 2, requiere que podamos crear un sumador de n bits arbitrario, para lo cuál los HDL son la herramienta mas apropiada. Implementarlos por medio de esquemáticos llevaría mucho tiempo, por eso descartamos esa metodología. En cambio, cuando describimos un circuito con un HDL lo hacemos parametrizando el tamaño del mismo, y a la hora de simularlo o implementarlo físicamente, determinamos su tamaño y generamos automáticamente el circuito.

4.1.1. Breve reseña de los HDL

VHDL y Verilog son los lenguajes más utilizados y conocidos para el diseño de circuitos integrados. VHDL nace como lenguaje de documentación del comportamiento de los circuitos integrados de aplicación específica (ASIC por su siglas en inglés). El departamento de defensa de Estados Unidos lo desarrolló para poder especificar a sus proveedores, cómo debía comportarse el sistema digital que les encargaba diseñar. Luego, surgio la idea de realizar una simulación lógica a partir de estos archivos VHDL. Lo siguiente fué el desarrollo de herramientas de síntesis lógica a partir de estos archivos, para generar una implementación física del circuito. Con Verilog sucedió algo muy parecido, aunque dentro del ámbito de la industria. Por esta historia en común, se puede decir que estos dos lenguajes tienen varias finalidades, siendo la implementación en *hardware* tan sólo una de ellas, es decir, uno puede describir circuitos que no son sintetizables. Por esa razón, y a pesar de ser los más utilizados y conocidos para describir circuitos digitales, no siempre son la mejor alternativa a elegir.

[1] HDL

4.1.2. Nuevos HDL

Podemos mencionar al menos tres lenguajes de descripción de *hardware* que están siendo utilizados para el diseño de circuitos integrados, que nacieron con el objetivo de aprovechar las ventajas de nuevos lenguajes de programación, bajo el paradigma de la programación funcional. Estos son, **Lava**(8) y **CλaSH**(2) basados en **Haskell**, y **Chisel**(18), basado en **Scala**. En este tipos de lenguajes, un circuito que no sea sintetizable es un error de sintáxis.

Otro lenguaje que queremos mencionar es **MyHDL**(9), un lenguaje basado en Python que brinda muchas ventajas de este lenguaje, que por estar basado en otro paradigma de programación, no lo agrupamos con los cuatro anteriores. Pero estos cinco lenguajes nos permiten:

- Usar un único lenguaje para describir (con distintos niveles de abstracción), simular, verificar e implementar el circuito.
- Los circuitos se describen en Haskell, Scala o Python (según correspoda), el HDL es simplemente un conjunto de módulos que permiten realizar nuevas tareas relacionadas al diseño de *hardware*, como puede ser crear un netlist VHDL, simulación simbólica, etc.
- Generar automáticamente una descripción en VHDL o Verilog, lo cuál nos permite utilizar herramientas de diseño físico que usan este tipo de lenguajes como entrada.
- Describir circuitos que construimos a partir de subcircuitos, además de la posibilidad de reutilizar fácilmente patrones de conexión.

4.1.3. ¿Por qué Lava?

Podemos elegir arbitrariamente cualquiera de estos lenguajes, ya que el circuito que lograremos es independiente del lenguaje utilizado. Por lo tanto, para este proyecto en particular[1] se puede elegir el HDL basado en el conocimiento y experiencia de uso en Haskell, Python o Scala.

Para describir el circuito, elegimos Lava. Quien haya utilizado alguna vez un lenguaje de programación funcional, entenderá fácilmente cómo describir circuitos. Quien no conozca este paradigma de programación, podrá aprender a utilizar Lava por medio de su documentación(7) sin muchas dificultades, ya que el paradigma de programación se basa en definir a los programas como se definen las funciones matemáticas. En Lava los circuitos son descriptos como funciones que operan sobre listas, tuplas o sobre circuitos. Esto último se debe a que el lenguaje Haskell permite la definición de funciones de alto orden, es decir podemos definir funciones que su dominio e imagen son funciones.

Lava es simplemente un conjunto de módulos de Haskell, por lo tanto estamos unificando un conjunto de tareas del diseño digital utilizando un lenguaje de programación de propósito general. Las ventajas de este acercamiento al diseño son varias:

- Disponibilidad de todas las librerías existentes en Haskell para ampliar las posibilidades de nuestra herramienta.
- No se generan *bugs* típicos de implementar dos veces el mismo circuito en dos lenguajes de programación distintos: El primero en la implementación a nivel de sistema y el otro a nivel de *hardware*.

[1] Es un circuito puramente combinacional

- Se pueden aplicar técnicas de programación propias del lenguaje como *testing* o *model checking* con el circuito.
- Permite manejar otros programas externos, por ejemplo lanzamos **minisat**(11) para verificar formalmente nuestro circuito.

Con la implementación del circuito utilizando este HDL, podemos generar un **netlist** VHDL, hacer simulación digital, verificación formal de propiedades, y la ventaja de

4.2. Implementación en lenguaje de descripción de hardware

Ya hemos presentado una descripción esquemática del sumador binario de n bits en la figura 3.4 y en la 3.11. El objetivo es implementar estos circuito en Lava parametrizando el tamaño n de los sumandos.

4.2.1. Implementación del sumador de ripple carry en Lava

Siguiendo la figura 3.3a, 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 3.3b, 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)
```

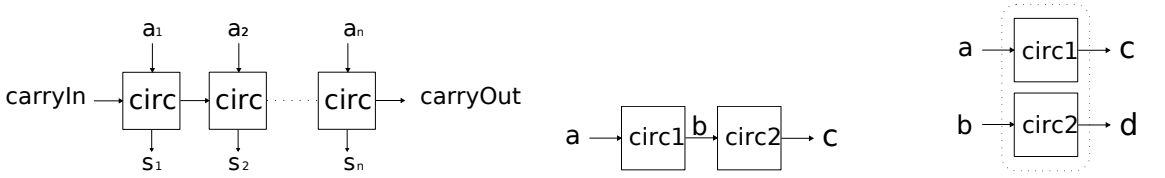
Por último escribimos la descripción del sumador binario (RCA) de la figura 3.4 de la siguiente forma:

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

4.2.2. Patrones de conexión

Patrones de conexión estandars. Los patrones de conexión son funciones de alto orden[1] que pueden ser utilizadas para construir circuitos, les llamamos circuitos de alto orden o generadores de circuitos.

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



(a) row (b) serial (c) parallel

Figura 4.1: Diferentes patrones de conexión de circuitos

Observando la definición de `rcAdder` y su topología, podemos generalizar esa estructura de conexión reemplazando el circuito por un parámetro, que en la definición[1] 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 4.1a. Ahora, usando el generador de circuito `row`, el sumador binario lo podemos describir mas simplemente asi:

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

Inclusive para simplificar mas, podemos currificar[2] la definición:

```
rcAdder'' = row fullAdd
```

Definir `rcAdder'` y `rcAdder''` 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
```

[1] Esto es posible dado que Haskell implementa *pattern matching*.

[2] Currificar, 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.

4.2.3. 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 3.9 siguiendo la figura 3.7:

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

Generación y Propagación del Acarreo

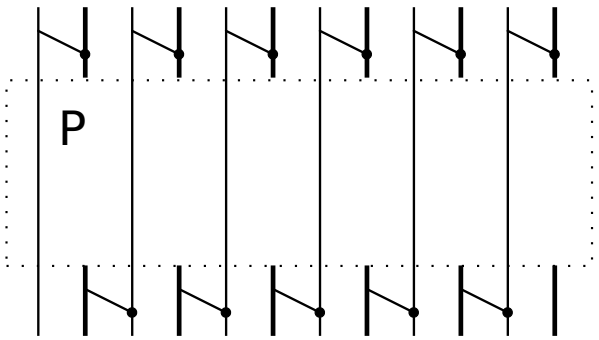
En Lava escribimos asi lo que captamos de la figura 3.8:

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

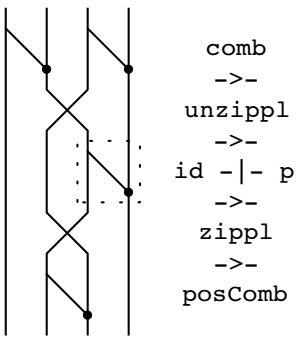
Para ver una explicación con mayor nivel de detalles de cómo construir el circuito, ver el manual de Lava (7) en conjunto con el paper aqui citado (16)

Red de Prefijos Paralelos para el sumador de Brent-Kung

Ahora para describir esta red que usamos en la figura 3.11 y mostramos un ejemplo de una red para 16 bits en a figura 3.10, nos basamos en un patrón recursivo que propone Sheeran (23) 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 4.2a. Esto nos lleva a construir redes como la de la figura 3.10.



(a) Patrón de recurrencia



(b) Primeras iteraciones de ppNet

Figura 4.2: Construcción de la red de prefijos paralelos de Brent-Kung.

La figura 4.2b representa las dos primeras iteraciones del circuito `ppNet`, en el cual la caja de lineas 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 3.10.

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 descripcion 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 finalemente, 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 3.11. 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
```

4.2.4. Sumador de Sklansky

En el apéndice D brindamos la descripción en Lava del sumador.

4.2.5. 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 enteros en base diez

Lava nos permite una interfase con números enteros, por si nos interesa simular utilizando esta representación numérica en vez de la binaria. 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 VCD[1] a partir de vectores de entrada[2].

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 5.

4.2.6. 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")
```

Y el segundo paso para crear el netlist, debemos especificar el valor real de sumador, por lo tanto valuamos el circuito con en 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 utilizaramos con una otra herramienta de síntesis, deberemos especificar que no modifique los cables para preservar la estructura de esta red.

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

[2] Podemos usar una libreria de Haskell llamada *casualmente* `vcd.` que nos permite escribir y leer archivos con este formato

Capítulo 5

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 asi: 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(11)[1].

5.1. 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.

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

5.2. Verificación de las Propiedades

5.2.1. Propiedades de la suma

La suma tiene las siguientes propiedades:

- Asociativa
- Conmutativa
- Existencia del elemento neutro cero.

5.2.2. Descripción de las propiedades en el Ripple Carry Adder

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 Ripple Carry de la siguiente forma:

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

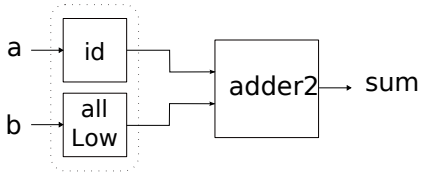


Figura 5.1: Circuito que nos sirve para describir la suma del elemento neutro

```
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
```

5.2.3. Equivalencia lógica entre el modelo de referencia y los sumadores de Brent-Kung y Sklansky

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
```

Y lanzamos el minisat desde Lava:

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

Parte II

Diseño Físico

Capítulo 6
Flujo de Diseño Físico

6.1. 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 6.1, se realiza iterativamente hasta lograr que el circuito cumpla las especificaciones con el menor costo en potencia disipada y área ocupada.

6.1.1. 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*. Depediendo de las herramientas que utilicemos, también se puede tener una estimación sobre la velocidad de las señales.

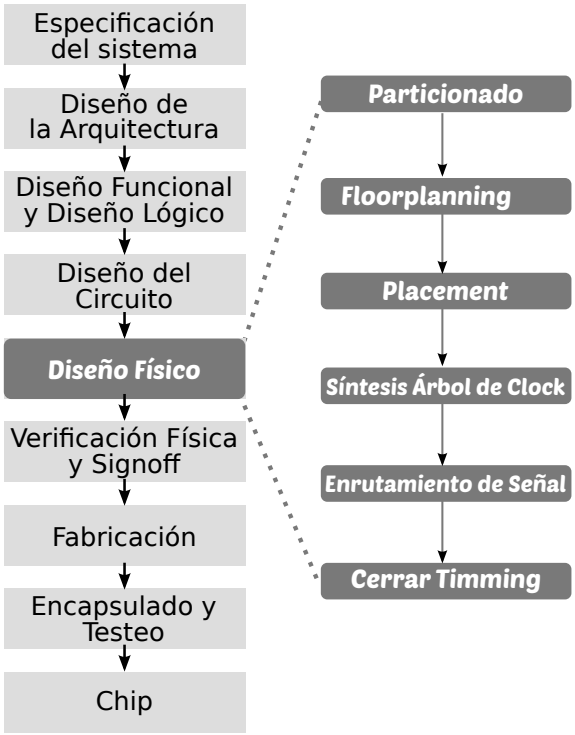


Figura 6.1: Flujo de diseño Físico

Síntesis del árbol de reloj Una vez que todos los elementos esten 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 6.2. 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 camínos críticos y se realizan las modificaciones necesarias para que el circuito cumpla con la especificaciones de retardo de propagacion 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.

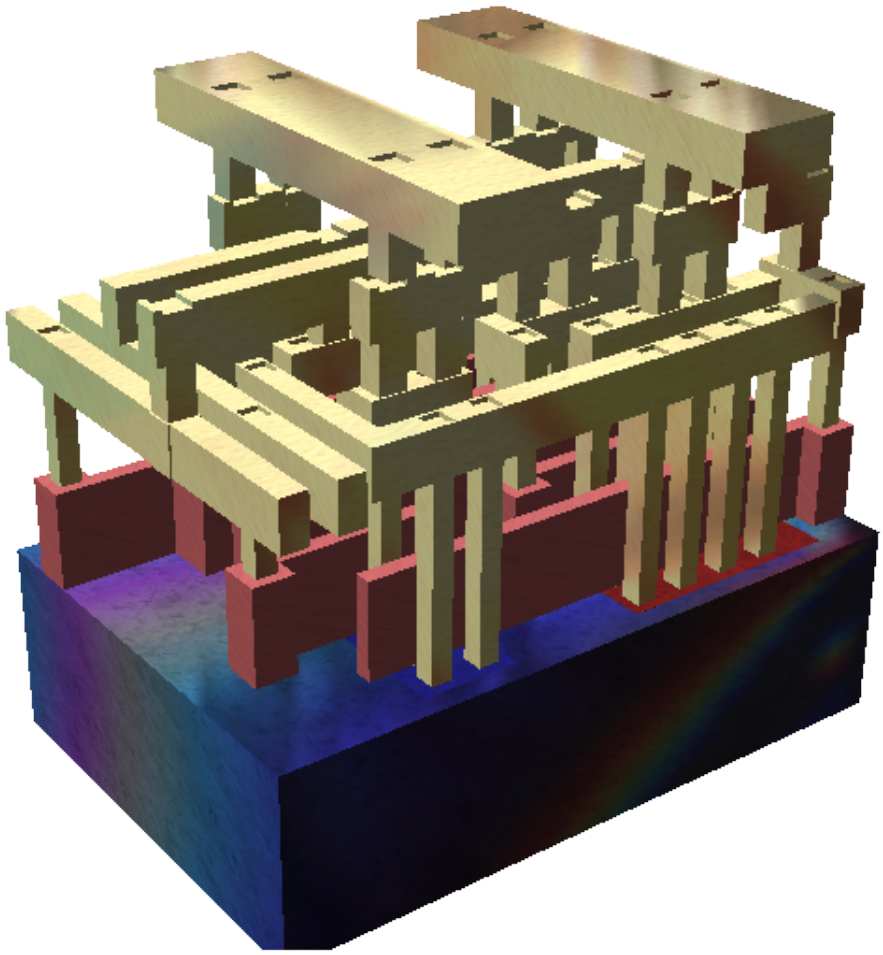


Figura 6.2: Representación en tres dimenciones de una celda estándar con 3 capas de metales en color arena, y una capa de silício policristalino en color ladrillo. Azul y rojo son dopado N^+ y P^+ respectivamente

6.2. Relevamiento, comparación y selección de las herramientas disponibles

Para realizar las tareas que describimos en la sección 6.1.1, será necesario buscar una o varias herramientas de software que se ajusten a los requerimientos del diseño y la tecnología de fabricación[1] del circuito integrado.

Características esperadas de las herramientas

- Desarrollo activo y existencia de una comunidad de usuarios/as y desarrolladores/as que brinden soporte

[1] Se utilizará una tecnología definida por Mead y Conway(17), 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.

- Mayor cantidad de herramientas integradas
- 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).

6.2.1. 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 modelo de retardo simplificado; **vesta**: herramienta para hacer Static Timing Analisys (STA); **Qflow**: entorno para realizar la síntesis digital con celdas estándar, utiliza Yosys(27); **graywolf**: programa que realiza el *placement*; **Qrouter**: programa que realiza el conexionado.

Electric VLSI Design System(10) 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(26) 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.

6.2.2. Comparación

En la tabla 6.2.2 resaltamos las ventajas y desventajas de cada herramienta, que nos permitirá hacer una selección en función de las necesidades del proyecto.

[1] El *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.

	Ventajas	Desventajas
Electric	Fácil instalación, acepta Python y Java como lenguajes para automatizar el diseño o implementar nuevos algoritmos de PnR. Todas las herramientas están integradas. Soporta muchos formatos de entrada y salida, que nos permite utilizar otras herramientas.	No tiene herramienta para STA. No incluye un compilador lógico, sólo acepta *netlist* VHDL para hacer PnR.
Open Circuit Design	Para realizar STA utiliza el estándar abierto Liberty. Extracción de parásitos muy precisa. Permite integración con otras herramientas fácilmente.	Muchos programas para instalar por separado. Díficil de instalar en otros sistemas operativos no libres.
Alliance	Incluye varias opciones de celdas estándar convenientemente integradas en la herramienta. Tiene todas las herramientas necesarias para un flujo físico completo.	Las herramientas para STA y conexionado no tienen una licencia libre. No facilita la integración con otras herramientas externas.

6.2.3. 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. Podemos resumir :

- 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 6.2.2, 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.

6.3. 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 semicoductores 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 sólamente a desarrollar el producto, otras que se dedican únicamente a desarrollar herramientas de software para el diseño, otras que sólamente se dedican a fabricar los diseños de otras empresas, y también persisten las empresas que realizan todo el proceso (conocidas como Integrated Device Manufacturers (IDMs)), abriendo las puertas a otras empresas de diseño sin fábrica (conocidas como *fabless*), para evitar la capacidad ociosa instalada y disminuir sus costos.

6.3.1. 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 circuitos integrados 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. Es importante mencionar que MOSIS cuenta con un programa especial para las universidades, que permite acceder a ciertos nodos[1] a muy bajo costo.

Por ello, nuestras opciones serán alguna de las que MOSIS ofrece. En la tabla 6.1 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.

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 6.1: Procesos disponibles por medio de MOSIS

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 performance[2]

Para dar cuenta de las capacidades de esta tecnología, vemos en la tabla 6.2 un conjunto de microprocesadores que la utilizaron cuando ésta 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

[1] Nodo es una forma alternativa de llamar al proceso tecnológico de fabricación que toma como segundo nombre el largo mínimo de canal de un trasistor MOS; por ejemplo: nodo de 180 nm se refiere al proceso con el cuál se puede fabricar un transistor con un mínimo de 180 nm de ancho de canal.

[2] Claro 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.

para la disminución de consumo de energía[1], la gran colección de *IP*[2] analógico[3] 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 TTM[4] 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 6.2: 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 (aluminio) para la interconexión, 1 capa de silicio policristalino (*poly*) para crear la compuerta y también para la interconexión de las mismas (distancias cortas sólamente, 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 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 conductor policristalino (*poly*) para crear las compuertas.
- DEEP: Reglas *deep submicron*.

Las reglas escalables se crearon originalmente para tecnologías desde 3um hasta 1um. 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*

[1] A modo de ejemplo, ver el procesador *Phoenix*, que en modo alerta consume 29.6 pW y 2.8 pJ/ciclo modo activo(22).

[2] Intelectual 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.

[3] Los 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)

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

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 es de 90 nm.

6.3.2. *Corners* de simulación

Debido a las variaciones propias del proceso de fabricación, obtendremos variaciones en las dimensiones físicas de los transistores y capas de metales. Por ello podemos esperar que el chip que probemos en el laboratorio luego de la fabricación contenga transistores que pueden ser lentos, típicos o rápidos, y que las interconexiones sean mas o menos capacitivas y resistivas. Generalmente todos los transistores e interconexiones dentro del chip seran de un sólo tipo. Por ello, el fabricante nos brinda modelos de simulación para los distintos casos de transistores que podemos esperar; y lo mismo con las capas de metal, nos brinda unas tablas que representan las distintas resistividades y capacidades parásitas que podemos esperar. Existe también la variación de tensión que puede tener la alimentación del circuito que vayamos a estudiar, ya que el diseño de la malla interna de alimentación se calcula con un margen de tolerancia de caída de tensión máxima. Por lo tanto podemos esperar que la alimentación de nuestro circuito sea de un %10 menor que la tensión nominal, por ejemplo. Por último, la temperatura ambiente impacta fuertemente en las características eléctricas del circuito integrado.

Por todo esto que mencionamos, se definen casos de simulación para contemplar las peores y mejores condiciones que podemos esperar. Este conjunto de casos se denominan *corners* de simulación. Por ejemplo, la peor condición para la performance es una temperatura de 0 grados, transistores lentos, capacidades mayores que la media y la tensión de alimentación un %10 menos que la nominal. Pero la peor condición desde el punto de vista del consumo de energía es 80 grados de temperatura ambiente, transistores rápidos, tensión nominal y capacidades mayores que la media. Como conclusión, debemos tener en cuenta las peores y mejores condiciones para realizar las simulaciones correspondientes, si esperamos que la simulación nos sirva para diseñar y lograr que la mayoría de los chips de un lote fabricado cumpla con las especificaciones de diseño.

En nuestro caso, utilizaremos un único modelo de transistores y capacidades y resistores parásitos que MOSIS brinda en su sitio web a modo de ejemplo del proceso de fabricación, ya que para obtener los *corners* de simulación es necesario contratar el servicio previamente. Tomamos ese modelo como el caso típico, y todas las simulaciones se hacen con 27 grados de temperatura ambiente y tensión de alimentación nominal (1.8 V).

6.4. 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 6.3 un ejemplo de una celda estándar que implementa la función lógica `xnor`

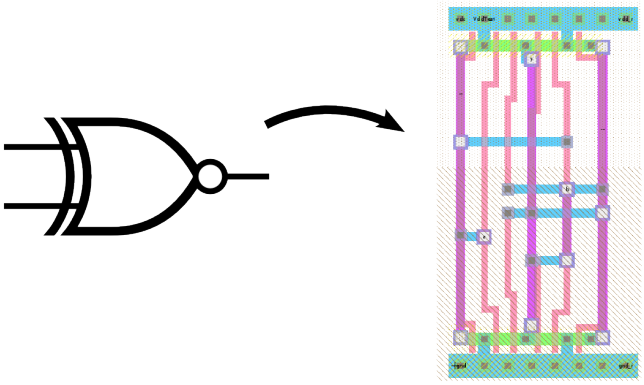


Figura 6.3: Mapeo de una función lógica a una celda estándar

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 débil(4).

6.4.1. Características

Estas celdas están correctamente dimensionadas para lograr el apilamiento en filas y columnas, y una grilla de interconexionado 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 λ[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 superiories, como vemos en la figura 6.4.

Ancho del riel de alimentación 8 λ

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.

[1] En la bibliografía en ingles se denomina *pitch*

Disposición de los pines Se ubican siempre en la intersección de las pistas horizontales y verticales, que tienen una separación de 8λ, ver figura 6.4.

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.

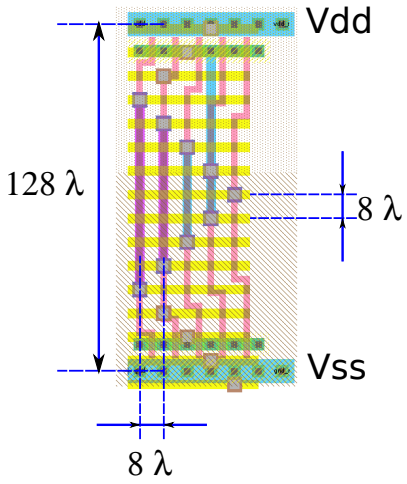


Figura 6.4: Grilla de interconexionado y riel de alimentación de las celdas estándar de 128λ. Por encima de cada celda, pueden pasar 16 pistas horizontales que la herramienta de conexionado tendrá a disposición, a partir del metal 3 para arriba. Notar la separación de $8\ \lambda$ para todas las pistas horizontales, y $8\ \lambda$ 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*.

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

Partimos desde la descripción estructural[1] que producimos en el capítulo 3. De *Electric* usaremos la herramienta llamada ***Silicon Compiler***, que se encarga de ubicar y conectar las celdas según el *netlist* VHDL.

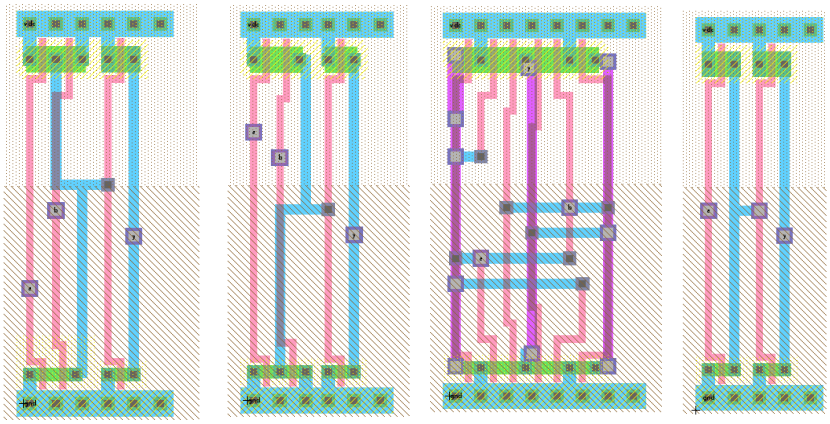
6.5.1. Modificación al código fuente de la herramienta de generación del *netlist* VHDL

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.

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



(a) and2_1x (b) or2_1x (c) xor_1x (d) buffer

Figura 6.5: Conjunto de celdas estándar

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 C. Luego de esta modificación, cuando realizamos la síntesis lógica descripta en el capítulo 3, el *netlist* VHDL obtenido ya puede ser utilizado por ***Silicon Compiler***.

6.5.2. Configuración de la herramienta de PnR

En la figura 6.6 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.

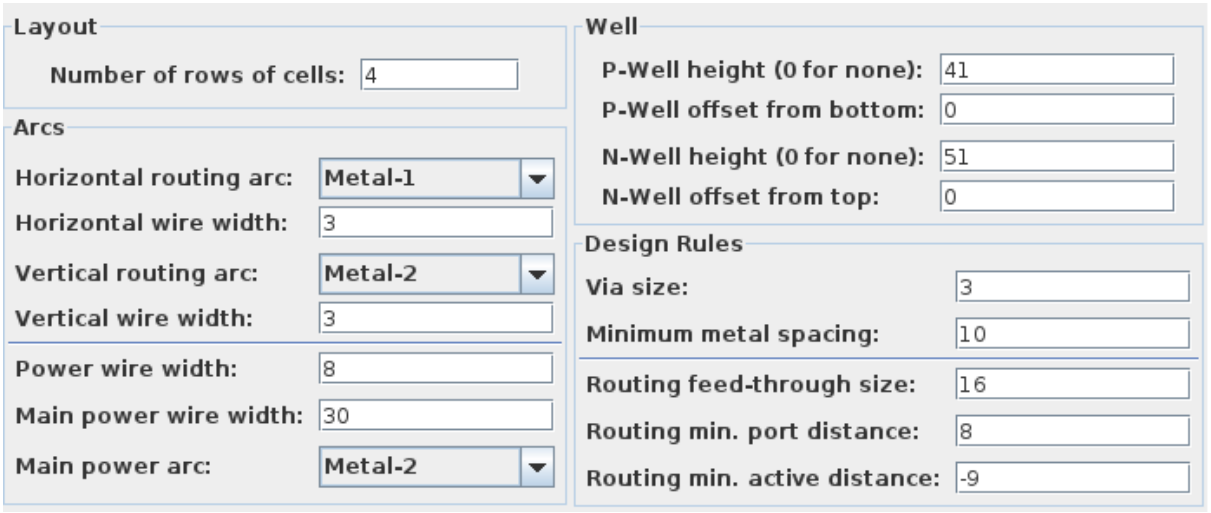


Figura 6.6: Configuración del Silicon Compiler de Electric

6.5.3. 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 métrica de selección del resultado será el área que ocupe nuestro circuito y la relación entre sus lados: cuanto más pequeño[1] y cuadrado mejor. En las tablas 6.3, 6.4 y 6.5 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 6.7 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.

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 6.3: Ubicación y conexionado para Ripple carry en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente.

Sklansky	8			16				32		
filas	3	4	5	4	5	6	7	6	7	8
ancho	1516	1167	954	3538	2042	1825	1536	3678	3229	2860
alto	810	973	1252	1345	1581	1878	2063	2639	2695	3072
área	1227960	1135491	1194408	4758610	3228402	3427350	3168768	9706242	8702155	8785920
ancho/alto	0,53	0,83	1,31	0,38	0,77	1,03	1,34	0,72	0,83	1,07

Cuadro 6.4: Ubicación y conexionado para Skalanksy en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente.

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 6.5: Ubicación y conexionado para Brent-Kung en 3 tamaños: 8, 16 y 32 bits. Las dimensiones de los lados y el área están en λ y λ^2 respectivamente.

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

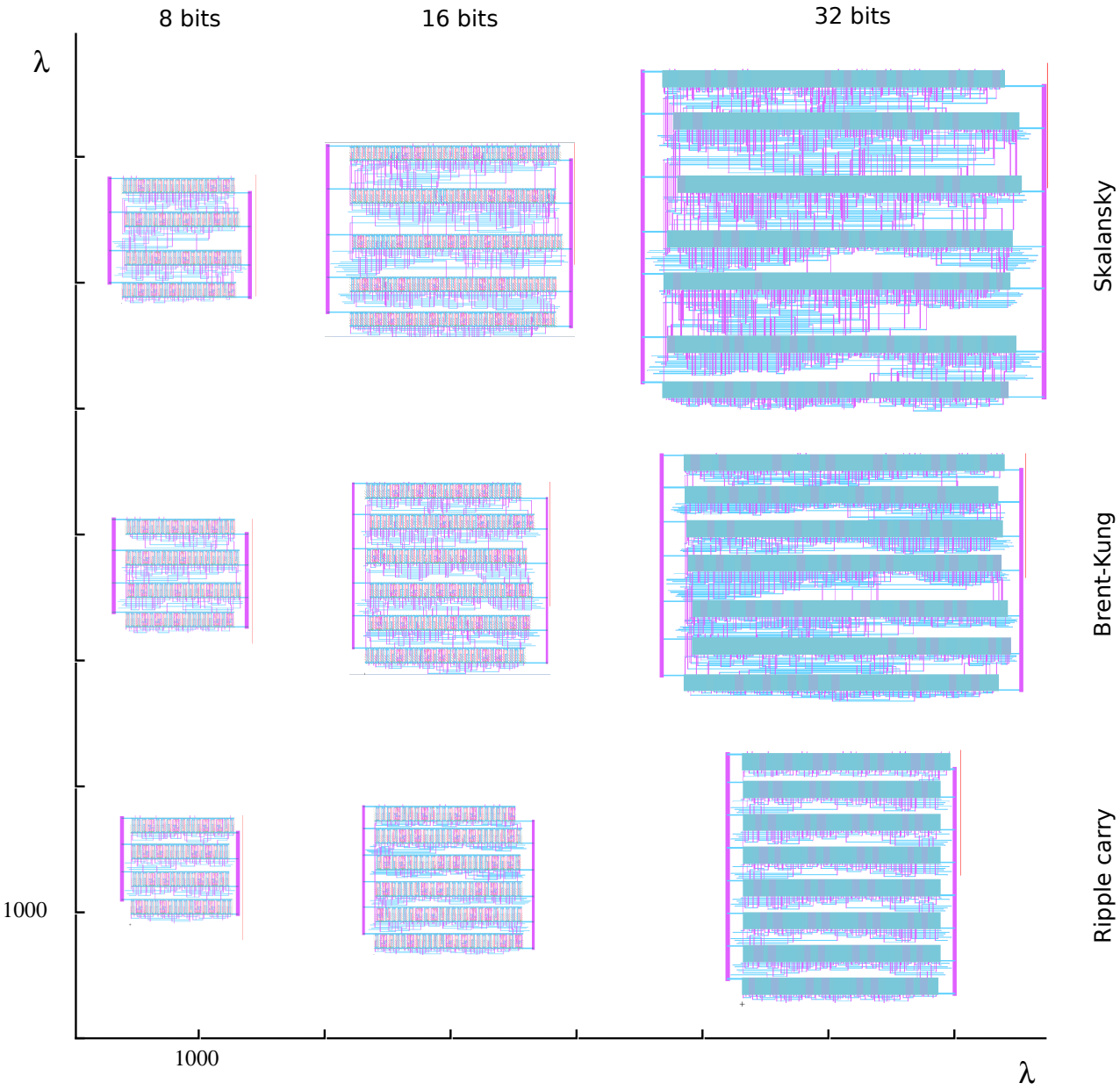


Figura 6.7: Tres arquitecturas y tres tamaños de sumandos distintos. Los circuitos están en escala, la unidad de los dos ejes es $\lambda = 90$ nm

Figura 6.8: Configuración de Electric: Especificar los modelos de transistores a utilizar por el motor tipo Spice.

6.6. Comparación de las distintas arquitecturas

6.6.1. Simulación post *layout* para calcular performance y potencia

Para realizar la comparación, necesitamos simular nuestros circuitos luego de hacer una extracción del *layout* del circuito, para obtener también las capacidades y resistores párasitos del mismo. A esta simulación se le suele denominar, simulación post *layout*.

Extracción del circuito, las dimensiones físicas de los transistores y elementos parásitos

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 para eso es necesario configurar **Electric** de la siguiente forma:

- Parámetros del modelo de transistor: Cargar los modelos de transistores de la tecnología que estamos usando (TSMC 180 nm). Este es un archivo que nos brinda MOSIS, del cual la primera parte nosotros debemos comentar (ya que es toda la información propia de la tecnología y no solamente la específica para el simulador), y renombramos como `tsmc180nm.model`. También es necesario realizar los siguientes cambios al archivo:
  `.MODEL CMOSN NMOS` cambiar por `.MODEL N NMOS`
  `.MODEL CMOSP PMOS` cambiar por `.MODEL P PMOS`
  Luego de esos cambios, cargar el archivo en:
  `File -> Preferences -> Tools -> Spice/CDL -> Use Header cards from file:`,
  como vemos en la figura 6.8

- Del archivo recién mencionado obtenemos la resistividad de los metales, el *poly*, el sustrato y el silicio dopado N^+ y P^+. También obtenemos la capacidad entre cada capa de material y el resto de las capas. Esta información la utilizamos para configurar **Electric** en:
  `File -> Preferences -> Tools -> Parasitic`

- Extracción de parásitos: Para realizar la extracción de las capacidades y resistores parásitos de las interconexiones, seleccionamos:
  `File -> Preferences -> Tools -> Spice/CDL -> Parasitics: Conservative RC`

- Seleccionar el formato (lenguaje) 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 tienen licencias calificadas como software libre: **Spice 3, Gnucap** y **Xyce**. **Spice 3** hace referencia a la versión **Spice3f5** del estándar de facto para las simulaciones de circuitos analógicos, en general todos los programas pueden leer un *netlist* en ese formato.

Motores de simulación analógica

De la lista de opciones que nos brinda **Electric**, hacemos una selección y breve reseña de los simuladores de circuitos que tienen licencia calificadas como software libre:

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*, permite elegir qué algoritmos utilizará el *solver*, 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**, **Spectre**[1] 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 cuenta con una comunidad de desarrolladores y usuarios que dan soporte y corrección de *bugs*. Fué ampliamente incorporado en otros entornos de simulación de circuitos, por ser la continuación con licencia libre del Spice3f5.

Xyce Simulador de circuitos con compatibilidad Spice, implementa 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).

Selección del simulador

Desde el punto de vista de la instalación de las herramientas, Ngspice y Xcye son más complicadas con respecto a Gnucap. Las características distintivas de estos dos proyectos no son necesarias para las simulaciones de nuestros diseños en particular. Por ello decidimos utilizar **Gnucap** ya que es de múy fácil instalación en el sistema operativo que estamos utilizando (Debian "Wheezy").

Ahora que ya tenemos seleccionado el simulador y configurada la herramienta para realizar la extracción, lo primero que simularemos es un oscilador anillo de 31 etapas, para dar cuenta de lo que logramos con nuestro flujo de herramientas (**Electric + Gnucap**), en comparación con lo que MOSIS brinda de esa prueba:

```
Ring Oscillator Freq.
 DIV1024 (31-stg,1.8V)              377.13  MHz
Ring Oscillator Power
 DIV1024 (31-stg,1.8V)                0.02  uW/MHz/gate
```

Simulación de un oscilador anillo

Realizamos el layout del oscilador anillo de 31 etapas que mostramos en la figura 6.9, y realizamos la simulación de régimen transitorio. Para ver la frecuencia de oscilación, guardamos los datos de la tensión de una salida de uno de los inversores, y para calcular la potencia, guardamos la corriente instantanea que sale de la fuente de alimentación. Nuestros resultados se pueden ver la tabla 6.6.

[1] Motor de simulación de Cadence.

Circuito	Frecuencia de oscilación	Potencia
Oscilador anillo de 31 etapas	325 MHz	0.026 uW/MHz/compuerta

Cuadro 6.6: Simulación post *layout* del oscilador anillo de 31 etapas

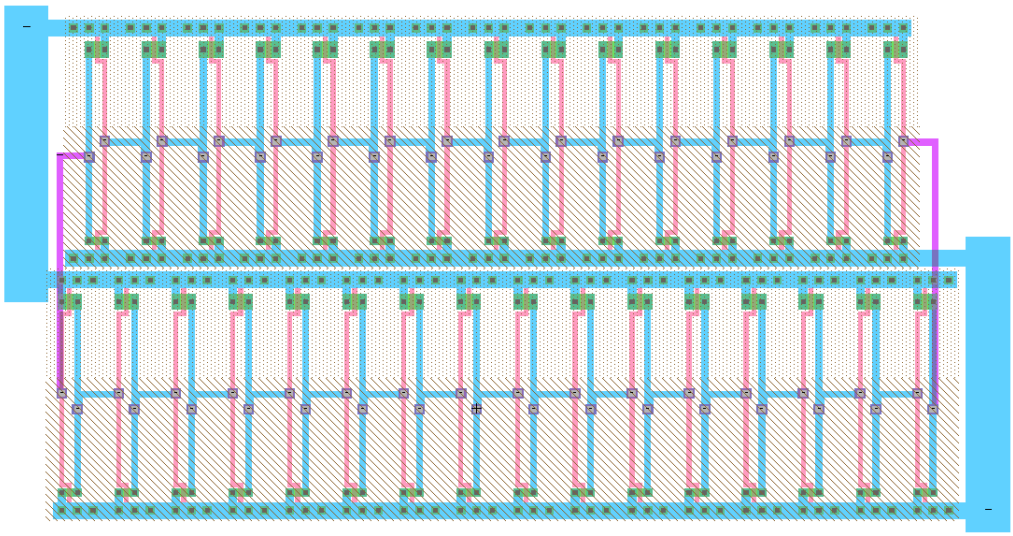


Figura 6.9: Oscilador anillo de 31 etapas

Justificamos la diferencia entre nuestros resultados y los datos brindados por el fabricante en que no existe un detalle de la prueba de caracterización de la tecnología de fabricación. Es decir, no brinda las dimensiones físicas del inveror utilizado, cómo fué interconectado, etc, lo cual impacta fuertemente en la performance y potencia.

6.6.2. Medición de la performance en un sumador

Siguiendo lo que definimos en el capítulo 2 para la performance, ahora seremos más específicos para los sumadores y definiremos las condiciones que nos permiten medir correctamente el tiempo de propagacion t_p del camino crítico. Realizaremos dos simulaciones de régimen transitorio para que el estado de las entradas cambie de forma tal que exite el camino crítico. Una simulación para encontrar t_{pHL} y otra para t_{pLH}.

Para eso necesitamos generar cambios de estados que generen acarreo en todos los bits, ya que el bit mas significativo de la suma y el acarreo de salida, dependen de los acarreos del bit anterior. Si logramos que desde el bit menos significativo se propague hasta el más significativo, estamos exitando el camino crítico para que se propague la señal desde los bits menos significativos a los más significativos. Esto se puede lograr con las siguientes 2 transiciones (ejemplo para un sumador de 8 bits):

$$(A_0, B_0) \rightarrow (A_1, B_1) = (\text{0x00}, \text{0xFF}) \rightarrow (\text{0x01}, \text{0xFF}) \tag{6.1}$$

$$(A_1, B_1) \rightarrow (A_0, B_0) = (\text{0x01}, \text{0xFF}) \rightarrow (\text{0x00}, \text{0xFF}) \tag{6.2}$$

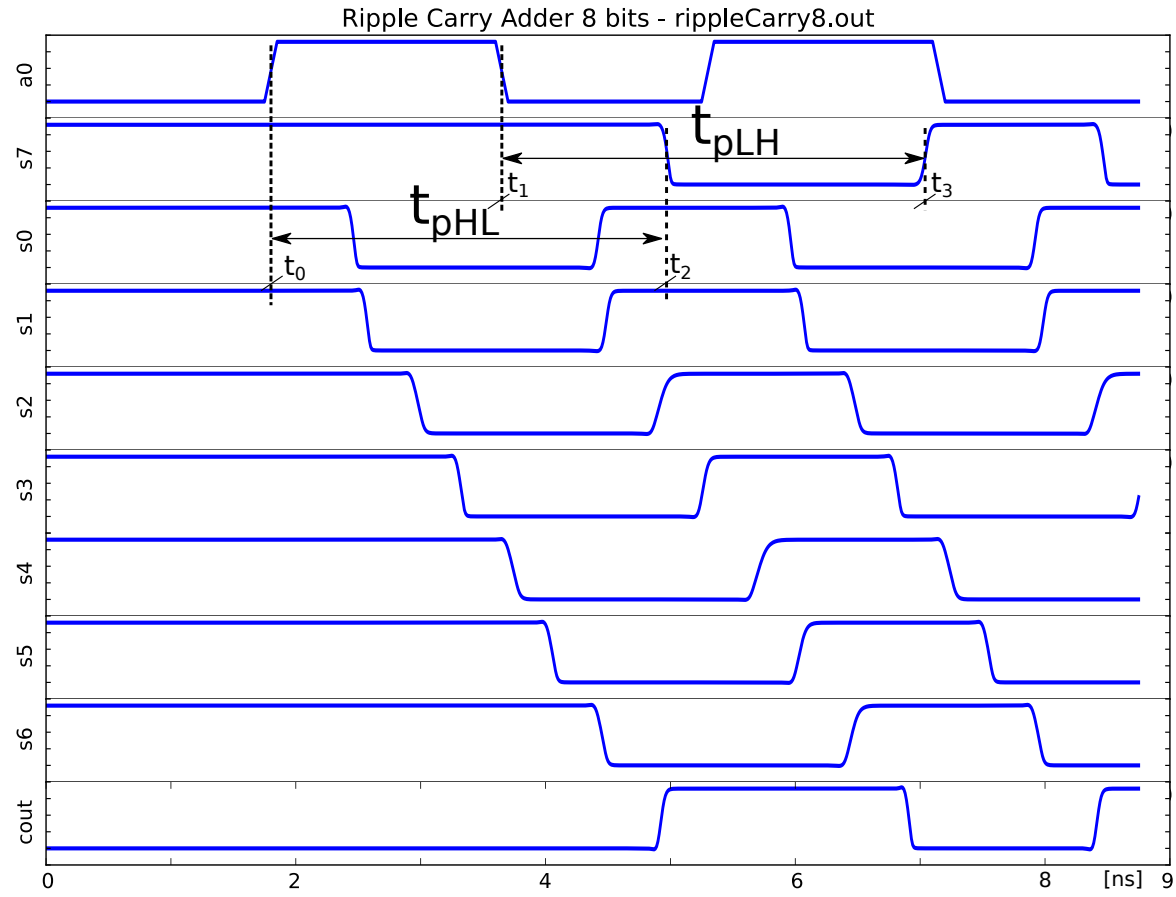


Figura 6.10: Simulación de régimen transitorio del circuito Ripple Carry 8 bits. De los vectores de entradas, sólo mostramos el bit menos significativo de la entrada A, porque es el único que cambia. Las señales de salidas están ordenadas para mostrar la más lenta cerca de la entrada y poder calcular el retardo.

Como el estado final de una transición es el estado inicial de la otra, y viceversa, podemos realizar estas dos mediciones con una sóla simulación de régimen transitorio que cíclicamente pase de un estado al otro. Eso es lo que mostramos en la figura 6.10, que utilizamos para encontrar el camino crítico. Vemos que la señal `s7` es la más lenta, y que la transición 6.1 nos sirve para calcular t_{pHL}. La transición 6.2 nos permite calcular t_{pLH}.

De la figura 6.11 extraemos los datos para realizar la medición del t_p del sumador de **ripple carry de 8 bits**:

$$t_{pHL} = t_2 - t_0 = 5{,}01 \text{ ns} - 1{,}8 \text{ ns} = 3{,}21 \text{ ns}$$

$$t_{pLH} = t_3 - t_1 = 7{,}05 \text{ ns} - 3{,}61\ ns = 3{,}44 \text{ ns}$$

$$t_p = \frac{(t_{pHL} + t_{pLH})}{2} = 3{,}325 \text{ ns}$$

6.6.3. Medición de la potencia en un sumador

La actividad de una señal afecta tanto a la potencia estática de un circuito CMOS como a la potencia dinámica. La potencia estática depende del estado de la señal, y la potencia dinámica

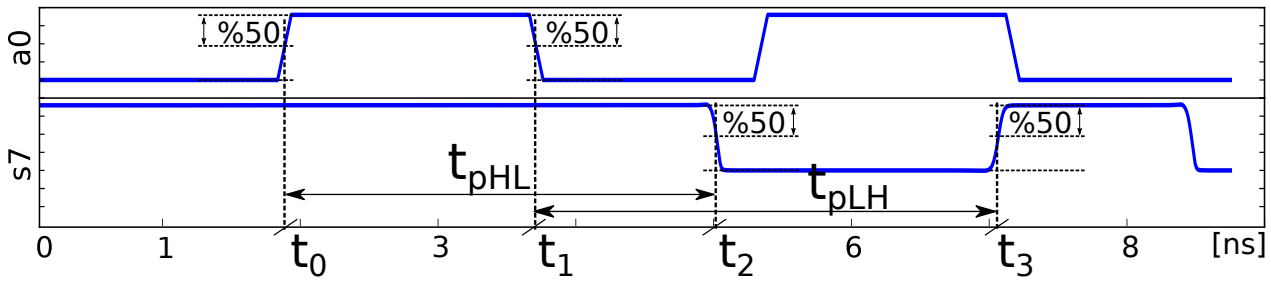


Figura 6.11: Simulación de régimen transitorio del circuito Ripple Carry 8 bits. Para calcular t_{pHL} y t_{pLH} lo hacemos sobre la señal más lenta del circuito, que en este caso es el bit más significativo de la suma

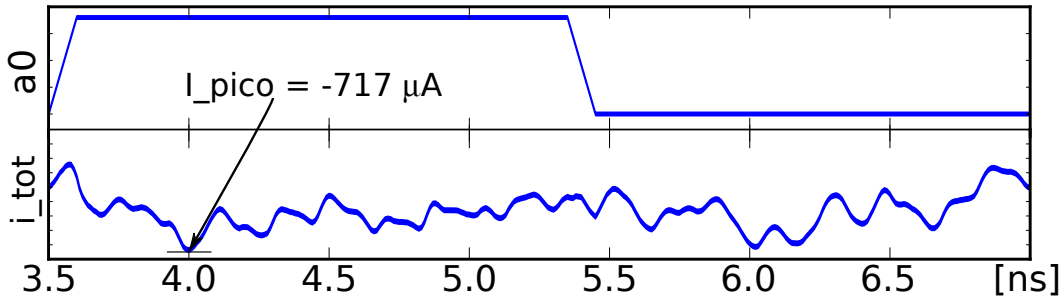


Figura 6.12: Simulación de régimen transitorio del circuito Ripple Carry 8 bits. Mostramos la corriente instantánea a través de la fuente de alimentación, el valor negativo se debe a que la corriente sale de la fuente.

depende de la tasa de cambio de los pines en cada celda estándar. Nuestra simulación para la potencia comparte los mismos vectores de entrada que el de performance, ya que con estos vectores generamos la mayor actividad posible en los pines de salida. En la figura 6.12 podemos ver las formas de onda de la exitación y la corriente instantánea que sale de la fuente de alimentación. Para obtener el valor de la potencia del circuito, aplicamos la ecuación 2.1, que lo realizamos directamente con el simulador, obteniendo una potencia promedio total de:

$$P_{av} = \frac{1}{\mathrm{t_f} - \mathrm{t_i}} \int_{t_i}^{t_f} p(t)dt = \frac{1{,}8\ \mathrm{V}}{3{,}5\ \mathrm{ns}} \int_{3{,}5\ \mathrm{ns}}^{7\ \mathrm{ns}} i_{\mathrm{fuente}}(t)\mathrm{d}t$$

$$P_{av} = -838{,}269\ \mu\mathrm{W}$$

El período de integración que elegimos está determinado por el t_p del circuito, lo que físicamente quiere decir: Medimos la potencia del circuito cuando está funcionando a la mayor velocidad posible.

6.6.4. Potencia y performance de todas las arquitecturas

La metodología que explicamos en las secciones 6.6.2 y 6.6.3, la aplicamos para todas las arquitecturas y todos los tamaños de bits que tomamos de prueba. En el apéndice E dejamos el código utilizado para realizar las simulaciones analógicas, junto con el código en **Python** para poder visualizar las formas de onda.

Producto performance-potencia

Si realizamos el producto t_p por la potencia de cada implementación, tenemos una métrica múy representativa del compromiso entre velocidad y potencia. Cuanto mas chico sea este número, mejor nuestro circuito. En la tabla 6.7 representamos esta métrica en la columna **PPP** (Producto Performance-Potencia).

6.6.5. Comparación de performance, potencia y área

Presentamos en la siguiente tabla el resumen de todas las simulaciones hechas para calcular la performance (midiendo el t_p) y la potencia media de cada circuito. Comparamos nuestras implementaciones de **Brent-Kung** y **Sklansky** con el equivalente en tamaño del sumador de **ripple carry**, para mostrar la mejora con respecto a este sumador.

		Área		t_p		Potencia		PPP
Arquitectura	bits	[μm^2]	%	[ns]	%	[μW]	%	[pJ]
Ripple Carry	8	7836.44		3.325		838.27		2.4
	16	16146		6.01		1033.07		6.2
	32	35101.6		11.5		1032.97		11.9
Brent Kung	8	9919	126.58	1.95	58.65	1095.91	149.23	2.1
	16	23793	147.36	4.1	68.22	1651.95	159.91	6.8
	32	56981.93	162.33	6.9	60.00	2049.86	198.44	14.1
Sklansky	8	11354.91	144.90	1.4	42.11	2330.71	317.38	3.3
	16	32284.02	199.95	3.1	51.58	2411.29	233.41	7.5
	32	87021.55	247.91	5.2	45.22	3237.50	313.42	16.8

Cuadro 6.7: Comparación de los resultados de las 3 arquitecturas

Podemos ver que el sumador de **Sklansky** en comparación con el **ripple carry** tiene una performance de hasta 2.2 veces más rápido para tamaños de 32 bits, con 3.13 veces mas de potencia y 2.47 veces más de área. Por otro lado, el sumador de **Brent-Kung** de 32 bits mejora la performance en 1.67 veces, con un costo en potencia de casi 2 veces más, y con una área 1.62 veces más grande.

Si comparamos estas dos arquitecturas entre sí, vemos que el Brent-Kung de 32 bits es un 25 % más lento que Sklansky, pero con un 37.7 % menos de potencia y un área 52 % más chica.

Por lo tanto, hemos logrado un conjunto de sumadores que según los requerimientos de área, potencia y performance, podremos elegir la arquitectura más adecuada. Para sumadores de 32 bits, la mayor velocidad se logra con Sklansky y el mejor compromiso entre velocidad, potencia y área con Brent-Kung. Para todos los tamaños de sumadores, si la performance no es un problema, un ripple carry es la solución optima de estos tres, ya que ahorra área y energía.

Parte III

Conclusiones

Capítulo 7
Conclusiones finales

Nos hemos propuesto resolver un problema que se encuentra en todos los sistemas de procesamiento de señales y los microprocesadores; lo hemos logrado resolver utilizando herramientas de software libre, y los resultados obtenidos son del mismo orden de magnitud que las soluciones propuestas en otros estudios en la misma tecnología. Para lograr este objetivo, y como subproducto del proceso, en la secciones 4.1.2 y 6.2 presentamos un breve informe sobre las alternativas en software libre disponibles para el diseño de circuitos integrados. Brindamos en la sección de apéndices todo el código fuente de las distintas tareas que hemos automatizado.

7.1. Metodología

Gran flexibilidad

Podemos elegir qué herramientas utilizar a lo largo de todo el flujo de diseño. Según la complejidad o magnitud del diseño, hay distintas alternativas. Si en alguna de las etapas del diseño es necesario o preferible utilizar otra herramienta, hemos encontrado que es posible realizarlo gracias a la existencia de formatos estándar para los archivos que utilizan las herramientas de software.

Electric resultó ser la herramienta más simple de instalar y utilizar, y que integra en un único entorno gráfico casi todas las herramientas necesarias para realizar el flujo físico. Utilizando la extracción del *layout* nos permitió elegir, un simulador analógico (entre varias alternativas posibles) y realizar simulaciones para calcular la potencia y performance. La herramienta de extracción de parásitos de egritaElectric tiene un modelo de parámetros concentrados que realiza una simplificación pesimista, la cual fué suficiente para realizar el análisis comparativo de las arquitecturas. La medición de performance y potencia se realizó con **Gnucap** a partir del *netlist* SPICE que generamos al hacer la extracción del circuito y parásitos del *layout* desde **Electric**.

La etapa de extracción de parásitos es realmente crítica, porque será la que nos permita predecir el real funcionamiento de nuestro sistema. En caso de buscar una mejor extracción de parásitos, se puede exportar el *layout* en formato `cif` para importar y extraer el cicuito desde Magic.

Aunque esta metodología nos permite obtener la mayor presición posible en performance y potencia, para circuitos mas grandes empieza a requerir de muchos recursos computacionales. Por ello, existe otra forma de realizar este análisis que se denomina STA que reduce enormemente el esfuerzo de cálculo. Para poder realizar el análisis de las métricas de esta forma, se puede exportar el circuito a `magic` y utilizar la herramienta llamada `vesta`.

El PnR lo realizamos de forma automática, con la única intervención nuestra para determinar las dimensiones físicas deseadas, en términos de filas y columnas de celdas estándar apiladas.

El HDL (**Lava**) que hemos utilizado es simplemente un conjunto de módulos de Haskell, por lo tanto realizamos el diseño digital utilizando un único lenguaje de programación de propósito general, aprovechando las ventajas de este. Además, hemos realizado la verificación formal de nuestros circuitos, de la misma forma en que describimos estos. Cabe mencionar la gran cantidad de alternativas para elegir el HDL, lo cual permite al diseñador optar por el lenguaje que se ajuste a sus necesidades, con la condición de que tenga la capacidad de producir un *netlist* VHDL estructural.

También es importante resaltar que esta metodología nos permite realizar **circuitos secuenciales**. Para ello, a partir del VHDL que nos genera Lava podemos utilizar una herramienta como YOSyS(27) para sintetizar este VHDL comportamental a un *netlist* a nivel de compuertas. YOSyS produce solamnete un *netlist* Verilog con el resultado de la síntesis, por lo tanto una alternativa es utilizar Magic para el *layout* .

7.2. Resultados

Los resultados obtenidos para el sumador de Brent-Kung y Sklansky están en el mismo orden de magnitud que otros estudios(6; 21) realizados en 180 nm. Considerando que no realizamos iteraciones para mejorar la implementación física, podemos afirmar que hay lugar para optimizar el resultado. Se pueden mejorar los resultados obtenidos por medio de la utilización de otros algoritmos para el PnR. Incluso hay mucho lugar para mejora si personalizamos las celdas estándar para favorecer alguna métrica a costa de otra, o si ampliamos nuestro conjunto de celdas estándar a versiones con mayor capacidad de manejo de corriente, sumado a un algoritmo de PnR que las utilice allí donde es necesario.

Otra fuente de mejora es realizar el *layout* completamente personalizado, sin utilizar herramientas de PnR, ya que en baja escala los resultados de un *layout* realizado por una persona son más optimos, si entendemos cabalmente el funcionamiento del circuito. Esta última opción sólo será aplicable si la performance de todo el sistema dependiese de éste circuito, porque en general no será posible realizar el layout manualmente ya que lleva mucho más tiempo. Por eso resaltamos que la importancia de estos resultados es que se lograron con una metodología automatizada, lo que nos permite pensar en escalas de circuitos más grandes que los sumadores que hemos implementado.

Como conclusión de los resultados obtenidos, podemos afirmar además de lograr un sumador rápido, hemos desarrollado la capacidad de implementar sumadores de cualquier tamaño de forma automática, y que se ajusten a los requerimientos de performance, potencia y área, según hemos detallado en la sección 6.6.5.

7.3. Aplicaciones

Con esta misma metodología queda abierta la posibilidad para la implementación de circuitos combinacionales en general: unidades aritméticas, decodificadores, codificadores, funciones lógicas, etc.

Para diseñar circuitos secuenciales se requiere de algunas tareas y herramientas extras que no hemos utilizado. La síntesis lógica del VDHL que generemos con Lava, y una herramienta

de STA para chequear si el circuito funciona a la velocidad esperada. Aunque en este trabajo no hemos abordado circuitos de este tipo, hemos mencionado las herramientas necesarias para poder realizarlos, con esta metodología levemente modificada.

Queremos resaltar que con esta metodología podemos diseñar circuitos analógicos, ya que hemos utilizado cinco de las seis herramientas básicas para realizarlo:

1. Simulador tipo SPICE
2. Editor de *layout*
3. Extracción de parásitos
4. LVS
5. DRC

La otra herramienta necesaria es el editor de esquemáticos, que también está disponible en **Electric**. En la figura 7.1 mostramos el flujo detalladamente.

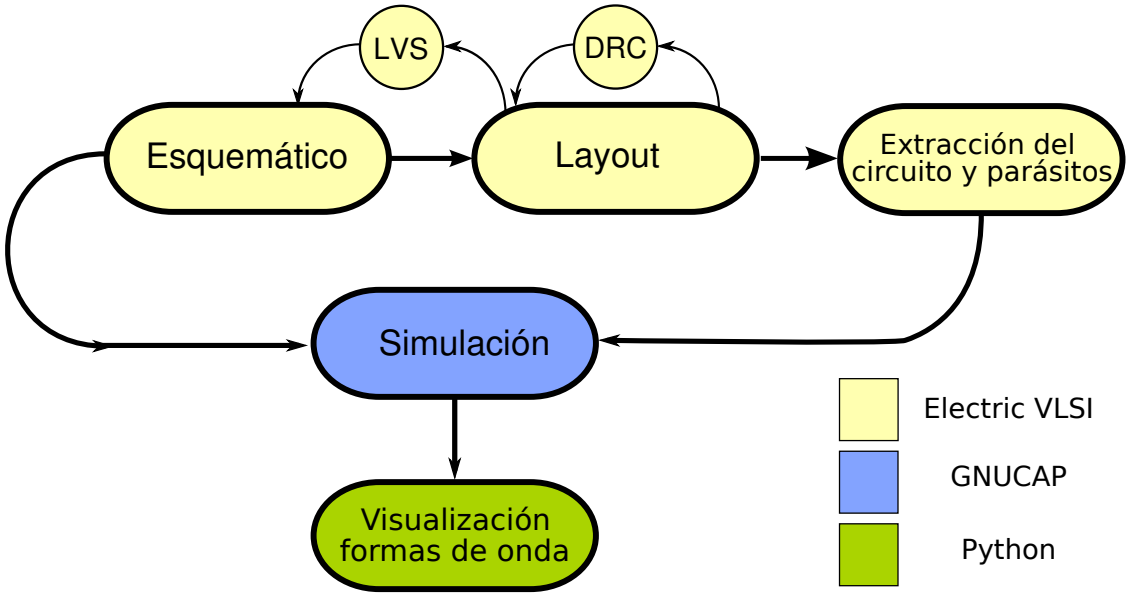


Figura 7.1: Flujo de diseño analógico.

7.4. Desafíos futuros

Al realizar este proyecto integrador, hemos detectado la oportunidad de desarrollar líneas de trabajo en pos de profundizar algunos aspectos de la metodología o de la aplicación.

- Crear un flujo que una los resultados esperados de PnR con la selección de la arquitectura a implementar, de forma automática.
- .
- .

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

LIBRERÍA DE LAVA PARA GENERAR NETLIST VHDL

Librería modificada para generar un *netlist* VHDL apropiado para el Silicon Compiler de Electric. Se puede descargar la última versión de esta librería en http://bit.ly/1vISP2r

```
module VhdlNew
  ( writeVhdlClk
  , writeVhdlNoClk
  , writeVhdlInputClk
  , writeVhdlInputNoClk
  , writeVhdlInputOutputClk
  , writeVhdlInputOutputNoClk
  )
 where

import Signal
import Netlist
import Generic
import Sequent
import Error
import LavaDir

import List
  ( intersperse
  , nub
  )

import IO
  ( openFile
  , IOMode(..)
  , hPutStr
  , hClose
  )

import System.IO
  ( stdout
  , BufferMode (..)
  , hSetBuffering
  )
```

```
import Data.IORef

import IOBuffering
  ( noBuffering
  )

--import IOExts
--  ( IORef
--  , newIORef
--  , readIORef
--  , writeIORef
--  )

import System
  ( system
  , ExitCode(..)
  )

-----------------------------------------------------------------
-- write vhdl

writeVhdlClk :: (Constructive a, Generic b) => String -> (a -> b) -> IO ()
writeVhdlClk = writeVhdl True

writeVhdlNoClk :: (Constructive a, Generic b) => String -> (a -> b) -> IO ()
writeVhdlNoClk = writeVhdl False

writeVhdl :: (Constructive a, Generic b) => Bool -> String -> (a -> b) -> IO ()
writeVhdl clocked name circ =
  do writeVhdlInput clocked name circ (var "inp")

writeVhdlInputClk :: (Generic a, Generic b) => String -> (a -> b) -> a -> IO ()
writeVhdlInputClk = writeVhdlInput True

writeVhdlInputNoClk :: (Generic a, Generic b) => String -> (a -> b) -> a -> IO ()
writeVhdlInputNoClk = writeVhdlInput False

writeVhdlInput :: (Generic a, Generic b) => Bool -> String -> (a -> b) -> a -> IO ()
writeVhdlInput clocked name circ inp =
  do writeVhdlInputOutput clocked name circ inp (symbolize "outp" (circ inp))

writeVhdlInputOutputClk :: (Generic a, Generic b)
                     => String -> (a -> b) -> a -> b -> IO ()
writeVhdlInputOutputClk = writeVhdlInputOutput True

writeVhdlInputOutputNoClk :: (Generic a, Generic b)
                     => String -> (a -> b) -> a -> b -> IO ()
writeVhdlInputOutputNoClk = writeVhdlInputOutput False


writeVhdlInputOutput :: (Generic a, Generic b)
                     => Bool -> String -> (a -> b) -> a -> b -> IO ()
writeVhdlInputOutput clocked name circ inp out =
  do writeItAll clocked name inp (circ inp) out
```

```
writeItAll :: (Generic a, Generic b) => Bool -> String -> a -> b -> b -> IO ()
writeItAll clocked name inp out out  =
  do noBuffering
     putStr ("Writing to file \"" ++ file ++ "\" ... ")
     writeDefinitions clocked file name inp out out
     putStrLn "Done."
 where
  file = name ++ ".vhd"

-----------------------------------------------------------------
-- definitions

writeDefinitions :: (Generic a, Generic b)
                 => Bool -> FilePath -> String -> a -> b -> b -> IO ()
writeDefinitions clocked file name inp out out  =
  do firstHandle  <- openFile firstFile WriteMode
     secondHandle <- openFile secondFile WriteMode
     var <- newIORef 0


     hPutStr firstHandle unlines
       [ "library ieee;"
       , ""
       , "use ieee.std_logic_1164.all;"
       , ""
       , "entity"
       , "  " ++ name
       , "is"
       , "port"
       , "  ( "
       , if clocked then "    clk : in std_logic ;" else " "
       , "    "] ++   -- , "  -- inputs"] ++
       [ "    " ++ v ++ " : in std_logic" | VarBool v <- [head inps]] ++
       [ "  ; " ++ v ++ " : in std_logic"
       | VarBool v <- tail inps
       ] ++
       [ ""
       , "  " -- outputs
       ] ++
       [ "  ; " ++ v ++ " : out std_logic"
       | VarBool v <- outs
       ] ++
       [ "  );"
       , "end " ++ name ++ ";"
       , ""
       , "architecture"
       , "  structural"
       , "of"
       , "  " ++ name
       , "is"
    , " --Agregado para que Electric encuentre las celdas estandards"
    , "component and2"
    , "port( A, B : in std_logic;  Y : out std_logic);"
    ," end component;"
    , "component or2"
    , "port( A, B : in std_logic;  Y : out std_logic);"
```

```
," end component;"
, "component xor2"
, "port( A, B : in std_logic;  Y : out std_logic);"
," end component;"
, "component id"
, "port( A : in std_logic;  Y : out std_logic);"
," end component;"
,"--"
   ]


 hPutStr secondHandle unlines
   [ "begin"
   ]

 let new =
       do n <- readIORef var
          let n  = n+1; v = "w" ++ show n
          writeIORef var n
          hPutStr firstHandle ("  signal " ++ v ++ " : std_logic;\n")
          return v

     define v s =
       case s of
         Bool True      -> port "vdd"  []
         Bool False     -> port "gnd"  []
         Inv x          -> port "inv"  [x]

         And []         -> define v (Bool True)
         And [x]        -> port "id"   [x]
         --And [x]       -> define v (Bool True) --modificaci n para probar, no funci
         And [x,y]      -> port "and2" [x,y]
         And (x:xs)     -> define (w 0) (And xs)
                        >> define v (And [x,w 0])

         Or  []         -> define v (Bool False)
         Or  [x]        -> port "id"   [x]
         Or  [x,y]      -> port "or2"  [x,y]
         Or  (x:xs)     -> define (w 0) (Or xs)
                        >> define v (Or [x,w 0])

         Xor  []        -> define v (Bool False)
         Xor  [x]       -> port "id"   [x]
         Xor  [x,y]     -> port "xor2" [x,y]
         Xor  (x:xs)    -> define (w 0) (Or xs)
                        >> define (w 1) (Inv (w 0))
                        >> define (w 2) (And [x, w 1])

                        >> define (w 3) (Inv x)
                        >> define (w 4) (Xor xs)
                        >> define (w 5) (And [w 3, w 4])
                        >> define v     (Or [w 2, w 5])

         VarBool s      -> port "id" [s]
         DelayBool x y -> if clocked then port "delay" [x, y] else wrong Error.DelayEva
         DlyBool x -> if clocked then port "dly" [x] else wrong Error.DlyEval
```

```
            _              -> wrong Error.NoArithmetic
          where
           w i = v ++ "[" ++ show i ++ "]"



           port "delay" [x, y] =
            do hPutStr secondHandle "" ++ make9("c" ++ v) -- ++ " : std" ++ " : " ++ make5" dff"
                       "  "
                  ++ make 9 ("c_" ++ v)
--                  ++ " :  std_"
                  ++ " :  "

                  ++ make 5 name
                  ++ " port map ("
                  ++ concat (intersperse ", " (args ++ [v]))
                  ++ ");\n"

    outvs <- netlistIO new define (struct out)
    hPutStr secondHandle unlines
      [ ""
      , "  " -- , "  -- naming outputs"
      ]

    sequence
      [ define v  (VarBool v)
      | (v,v ) <- flatten outvs  zip  [ v  | VarBool v  <- outs  ]
      ]

    hPutStr secondHandle unlines
      [ "end structural;"
      ]

    hClose firstHandle
    hClose secondHandle

    system ("cat " ++ firstFile ++ " " ++ secondFile ++ " > " ++ file)
--    cat firstFile secondFile file
    system ("rm " ++ firstFile ++ " " ++ secondFile)
   -- Create a new vhdl netlist without the wire or id gates:
    system ("/home/lean/tfinal/programas/lava/la-va/Lava2000/Scripts/deleteWire.pl "
    return ()
 where
  sigs x = map unsymbol . flatten . struct
```

Apéndice C

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
```

Apéndice D

DESCRIPCION EN LAVA DE LOS SUMADORES SKLANSKY Y BRENT-KUNG

D.1. Sumador de Sklansky

```
sklansky :: [(Bit, Bit)] -> ([Bit], Bit)
sklansky abs = (ss,cout)
  where
    gps             = map gpC abs
    (cs,cout)       = (skl (mkFan dotOp) ->- unsnoc ->- (map fst -|- fst) ) gps
    ((_,p) : gps )  = gps
    rs              = zip cs gps
    ss              = p : map sumC rs
--
--
{-- La funcion dotOp utilizada es la
 misma que la que definimos para Brent-Kung.

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

{--
Funciones auxiliares utilizadas en el top level de sklanksy:
--}
gpC:: (Bit,Bit) -> (Bit,Bit) -- Genera los g y p a partir de los bit de entrada a y b.
gpC (a,b) = (a <&> b,a <#> b)

sumC :: (Bit,(Bit,Bit)) -> Bit -- Realiza la ultima operacion (una XOR ) para calcular
sumC (cin, (_,p)) = cin <#> p

mkFan :: ((a,a) -> a) -> Fan a
mkFan op (i:is) = i:[op(i,k) | k <- is]

fsT f = (f  -|- id)
snD f = (id -|-  f)
```

```
unsnoc as = (init as, last as)
--

-- Red de prefijo de sklansky:
skl :: PP a
skl _ [a] = [a]
skl f as  = init los ++ ros
  where
    (los,ros) = (skl f las, skl f ras)
    ros       = f (last los : ros)
    (las,ras) = splitAt (cnd2 (length as)) as

cnd2 n = n - n  div  2 -- El techo de n/2
--

{--
 Version del sumador que acomoda el tipo de datos de sklansky
 para hacerlo compatible con el modulo para generar el netlist VHDL.
 La version definida arriba toma una lista de tuplas. Y la que
 creamos aqui toma una tupla de listas:
--}
sklansky_ :: ([Bit], [Bit]) -> ([Bit], Bit)
sklansky_ (as,bs) = sklansky (zip as bs)
--
```

D.2. Sumador de Brent-Kung

```
dotOp ((g1, p1) ,(g, p)) = (go, po)
   where
      go = or2 (g, and2 (p, g1))
      po = and2 (p, p1)
--
unzipl []         = ([],[])
unzipl [a]        = ([a], [])
unzipl (a:b:abs) = (a:as, b:bs)
   where
      (as, bs) = unzipl abs
--
zipl ([], [])     = []
zipl ([a], [])    = []
zipl (a:as, b:bs) = a:b:zipl(as, bs)
--
buf (gin,pin) = (gin, pin)
{--
dop toma un operador y un buffer y hace una funcion de
una lista de duplas a lista de duplas.
Esto es lo que vamos a usar como red de
prefijo cuando tenemos dos entradas
--}
dop [a,b] = [a, dotOp(a,b)]
--
```

```
miti p = unzipl ->- (id -|- p) ->- zipl
--

comb []     = []
comb [a]    = []
comb (a:as) = dop [a, head as]++comb (tail as)
--

posComb (a:as)  = a: (comb (init as))++ [last as]
--
wrapR p = comb ->- miti p ->- posComb

{--
Toda la version del sumador est  hecha para que no
 pida como par metro el dotOp, con esta version no
 se puede usar el programa de Mary para hacer los
dibujos de las redes, adem s de perder de vista
donde se puede poner un buffer real. Esto no es
un gran problema ya que las herramientas de
place & route tienen la capacidad de agregar
o quitar buffers segun sea necesario.
--}
bKung [a] = []
bKung [a, b] = dop [a, b]
bKung as  = wrapR (bKung) as

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

fork as = (as, as)
--

evens :: [(Signal Bool,Signal Bool)] -> [Signal Bool]
evens as = cs
   where
      (bs,cs) = unzipp as
--

odds ::  [(Signal Bool,Signal Bool)] -> [Signal Bool]
odds as = bs
   where
      (bs,cs) = unzipp as

--

dropP :: ([Signal Bool], [(Signal Bool,Signal Bool)]) -> ([Signal Bool],[Signal Bool])
dropP = id -|- odds
--

lastXor (as, bs) = map xor2 cs
   where
```

```
      cs = zipp (as, bs)
--

sums (a:as, bs) = (a: lastXor (as, init bs),carryOut)
   where
      carryOut = last bs
--

bKungFastAdder = gAndPs ->- fork ->- (evens -|- bKung) ->- dropP ->- sums
```

Apéndice E

BANCO DE PRUEBA PARA SIMULACIONES EN GNUCAP

E.1. Análisis de potencia de los sumadores

Estas simulaciones tardaron aproximadamente 10 minutos para los circuitos de 8 bits, 1 hora para los de 16 bits y 14 horas para los circuitos de 32 bits. La computadora en que realizamos las simulaciones tiene 4GB de RAM y las siguientes características detalladas por el programa `lshw`:

```
 *-cpu:0
      description: CPU
      product: Intel(R) Celeron(R) CPU 847 @ 1.10GHz
      vendor: Intel Corp.
      physical id: 41
      bus info: cpu@0
      version: 6.10.7
      serial: 0002-06A7-0000-0000-0000-0000
      slot: SOCKET 0
      size: 800MHz
      capacity: 3800MHz
      width: 64 bits
      clock: 100MHz
*-cpu:1
      physical id: 1
      bus info: cpu@1
      version: 6.10.7
      serial: 0002-06A7-0000-0000-0000-0000
      size: 800MHz
      capacity: 800MHz
```

```
*** Test bench para an lisis de potencia del sumador
* Para usar este script:
* Para correr este script hacer:
gnucap -i tb_potencia.gnucap

* Pero antes es necesario hacer algunos enlaces simbolicos:
```

```
* Extraccion post-layout:
* ln -s ext_circuit.spi nombre_real_del_netlist_spice_extraido

* Circuito de exitacion de las entradas:
* ln -s inputs.spi nombre_real_archivo_exitacion
* Este script asume que los nombres de las entradas son:
* a0, a1, a2, ..., a_{n-1} (para el primer sumando)
* b0, b2, b3, ..., b_{n-1} (para el segundo sumando)
* s0, s2, s3, ..., s_{n-1} (para la suma)
* cout (para el acarreo de salida)
* Fin de la explicacion de uso.
*************************

*************************
* Configuraciones posibles
* Configuraci n para la frecuencia:
* Seteo el par metro para fijar la frecuencia en el circuito de exitaci n:
param periodo = 2.2n
* Nombre de archivo donde se registran los valores
* de potecia::
log gnucap.log

* Para que converga la corriente con mejor resoluci n:
* Se podr a hacer m s preciso a n , pero las simulaciones tardar an mas
* sin tanta ganancia en precision.
options dtmin=1e-15
options reltol= 10u
options abstol= 1f
* Fin de la configuracion.
******************

*****************
* Comienza el testbench:
* Cargo el circuito:

get ext_circuit.spi

* Cargo la exitaci n de entrada
.include inputs.spi

* Comienzo la simulaci n llevando el circuito
* al punto de operaci n por defecto (Temperatura 27 grados)
op

* Elijo en qu  puntos voy a medir qu  cosa:
* Si el circuito es m s chico que 32 bits, da un error pero sigue todo bien.
probe tran   v(a0)
probe tran + v(sum0)  v(sum1)  v(sum2)  v(sum3)  v(sum4)  v(sum5)
v(sum6) v(sum7)
probe tran + v(sum8)  v(sum9)  v(sum10) v(sum11) v(sum12) v(sum13) v(sum14) v(sum15)
probe tran + v(sum16) v(sum17) v(sum18) v(sum19) v(sum20) v(sum12) v(sum22) v(sum23)
probe tran + v(sum24) v(sum25) v(sum26) v(sum27) v(sum28) v(sum29) v(sum30) v(sum31)
probe tran + v(cout)  i(VVDC_vdd)
* Luego elijo que voy a grabar para graficar y medir:
store tran i(VVDC_vdd)
```

```
trans 0 {periodo*3} 0.01n > salida.spo

measure area = integrate(probe="i(VVDC_vdd)" begin=0 end=6n)

* Para potencia:
param t_i = {2*periodo}
param t_f = {3*periodo}
param t_tot = {t_f - t_i}
measure area = integrate(probe="i(VVDC_vdd)" begin=t_i end=t_f)
param total_energy = area*1.8
eval total_energy
param av_power = total_energy/t_tot
eval av_power
```

E.2. Análisis de performance en los sumadores

Apéndice F

SCRIPTS PARA GRAFICAR LAS FORMAS DE ONDAS

```
#!/usr/bin/python
# coding: latin-1
# La segunda l nea (# coding: latin-1) es necesaria para
# poder usar acentos y   . Para entender el porqu , visitar:
# http://www.python.org/dev/peps/pep-0263/
from numpy import *
#import pylab
import sys
import matplotlib.pyplot as plt
# Primero cargo el resultado de la simulaci n en un arreglo:
filename_str = sys.argv[1]
data = genfromtxt(filename_str, skip_header=0,unpack=True)

# http://matplotlib.sourceforge.net/api/figure_api.html#module-matplotlib.figure
print sys.argv[1]
totalPlots = len(data)
############################
# nob subplots sharing both x/y axes
f, eje = plt.subplots(totalPlots, sharex=True, sharey=False)

#eje[0].plot(data[0], data[1])
#eje[0].set_ylabel( a0 )
#eje[0].set_title(filename_str )

# Comienzo las etiquetas del resultado a partir del 3 elemento:
for i in range(1,len(eje)):
   eje[i].plot(data[0], data[i])
   eje[i].set_ylabel("s"+str(i))

# Escribo la segunda etiqueta:
#eje[1].plot(data[0], data[2])
#eje[1].set_ylabel( cout )



# Fine-tune figure; make subplots close to each other and hide x ticks for
```

```
# all but bottom plot.
f.subplots_adjust(hspace=0)
plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)

# Escondo los ticks de todos los ejes Y:
plt.setp([a.get_yticklabels() for a in f.axes[:]], visible=False)

# Con este comando aparece la ventana con los gr ficos
plt.show()
```

Bibliografía

[1] MANUEL VALENCIA ADRIÁN ESTRADA, CARLOS J. JIMÉNEZ. Características de Sumadores en Tecnologías Fuertemente Submicrónicas. *IBERCHIP*, **DOC (TEC 2004-01509/MIC)**, 1982. 7

[2] CHRISTIAAN BAAIJS. Cλash CAES language for synchronous hardware. http://christiaanb.github.io/clash2/. 20

[3] 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. 8

[4] 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. 41

[5] R. P. BRENT AND H. T. KUNG. A Regular Layout for Parallel Adders. *IEEE Transaction on Computers*, **C-31, Issue: 3**:260–264, 2006. 7, 8, 13, 15

[6] BHASKAR CHATTERJEE, MANOJ SACHDEV, AND RAM KRISHNAMURTHY. A cpl-based dual supply 32-bit alu for sub 180nm cmos technologies. 56

[7] K. CLAESSEN AND M. SHEERAN. A tutorial on Lava: A hardware descritption and verification system. Website, 2014. http://projects.haskell.org/chalmers-lava2000/Doc/. 20, 23

[8] KOEN CLAESSEN. Lava. a hardware description and verification language. http://hackage.haskell.org/package/chalmers-lava2000. 20

[9] JAN DECALUWE. Myhdl. A hardware description and verification language. http://hackage.haskell.org/package/chalmers-lava2000. 20

[10] STEVE RUBIN ET AL. Electric VLSI design system. http://www.staticfreesoft.com/. 36

[11] NIKLAS EÉN AND NIKLAS SÖRENSSON. Minisat. Minimalistic, open-source SAT solver. http://minisat.se/. 21, 27

[12] J. HENNESSY AND D. PATTERSON. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition edition, 2007. 5

[13] S. KNOWLES. A family of adders. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 277–281, 2001. 7

[14] 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. 7, 13

[15] RICHAR E. LADNER AND MICHAEL J. FISCHER. Parallel prefix computation. *JACM, Journal of the ACM*, **C-27**(4):831,838, Oct 1980. 7, 13

[16] L. MARSO. Brent-kung fast adder dscription, simulation and formal verification using lava. In *Micro-Nanoelectronics, Technology and Applications, 2008. EAMTA 2008. Argentine School of*, pages 111–114, Sept 2008. 23

[17] CARVER MEAD AND LYNN CONWAY. *Introduction to VLSI Systems*. Addison-Wesley, 1980. 35

[18] UNIVERSITY OF CALIFORNIA BERKELEY. Chisel. constructing hardware in a scala embedded language. https://chisel.eecs.berkeley.edu/. 20

[19] B. PARHAMI. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000. 10

[20] 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, 4, 5

[21] P. RAMANATHAN AND P. T. VANATHI. A novel power delay optimized 32-bit parallel prefix adder for high speed computing. 56

[22] 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. 39

[23] 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. 23

[24] J. SKLANSKY. Conditional sum addition logic. *RE Transactions on Electronic Computers*, **EC-9**(6):226–231, June 1960. 7, 13

[25] B. SUGLA AND D.A. CARLSON. Extreme area-time tradeoffs in vlsi. *Computers, IEEE Transactions on*, **39**(2):251–257, Feb 1990. 7

[26] LIP6 UNIVERSITÉ PIERRE ET MARIE CURIE. Alliance VLSI CAD System. https://soc-extras.lip6.fr/en/alliance-abstract-en/. 36

[27] CLIFFORD WOLF. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/. 36, 56