Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño e implementación de una Unidad Aritmética de Coma Flotante (FPU) genérica y flexible.

Proyecto de Graduación

Jorge Esteban Sequeira Rojas

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

PROYECTO DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa de Proyecto de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del proyecto de graduación denominado Diseño e implementación de una unidad aritmética de coma flotante (FPU) genérica y flexible., realizado por el señor Jorge Esteban Sequeira Rojas y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Ing. José Alberto Díaz García

Profesor lector

Ing. Roberto Pereira Arroyo

Profesor lector

Dr. Ing. Alfonso Chacón-Rodríguez

Profesor asesor

Cartago, 1 de diciembre, 2016

Yo, JORGE ESTEBAN SEQUEIRA ROJAS, con cedula N. 1-1528-0392, declaro que el proyecto titulado: "Diseño e implementación de una Unidad Aritmética de Coma Flotante genérica y flexible", ha sido desarrollada con base a una investigación exhaustiva, respetando derechos intelectuales de terceros, cuyas fuentes se incorporan en la bibliografía. Consecuentemente este trabajo es de mi autoría. En virtud de esta declaración, me responsabilizo del contenido, veracidad y alcance científico del mismo proyecto de graduación.

Jorge Esteban Sequeira Rojas

Cartago, 6 de diciembre de 2016

Céd: 1-1528-0392

Resumen

Se realiza una propuesta metodológica para la medición del rendimiento de un ASP con una unidad de coma flotante con énfasis en la potencia.

Adicionalmente, se realiza la síntesis de una unidad de sumado, multiplicación, coseno y seno parametrizable, y verificados para una precisión simple y doble.

Se encuentra que el sumador segmentado se puede operar a una velocidad máxima de 350MHz, el multiplicador (con multiplicación a los significandos de Karatsuba) a una velocidad de 243MHz y el operador CORDIC a una velocidad de 537MHz, sin el operador de sumado.

Palabras clave: FPU, 0.13μm, Karatsuba, CORDIC, Benchmarking, ASP

Abstract

A methodology that measures the DSP performance of a ASP with low-power target applications is presented.

Additionally, the timing, area and power synthesis results for an adder, a multiplier and a CORDIC floating point units in Artix 7 FPGA family and 0.13μm technology for single and double precision in various system frequencies, are presented.

The pipelined adder achieves a maximum frequency of 350MHz, the multiplier (with a simple Karatsuba significand multiplication) reaches 243MHz, and lastly, the standalone CORDIC floating point operator reaches 537MHz.

Keywords: FPU, ASP, Benchmarking, 0.13μm, Karatsuba, CORDIC.

a mis queridos padres

Agradecimientos

Agradezco el apoyo incondicional de mi padre, el inagotable cariño de mi madre, y el cariño de mis hermanas.

Al país, a los profesores, a los maestros, a las amistades. Gracias.

Jorge Esteban Sequeira Rojas

Cartago, 6 de diciembre de 2016

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El Laboratorio de Diseño de Circuitos Integrados, DCILab, forma parte de la Escuela de Ingeniería Electrónica, del Instituto Tecnológico de Costa Rica. En este, se han impulsado y desarrollado diversos proyectos, entre los cuáles se encuentra el Sistema de Reconocimiento de Patrones Acústicos, SiRPA.

El SiRPA forma parte de un proyecto de la Escuela de Ingeniería en Electrónica denominado "Sistema Electrónico Integrado en Chip (SoC)", cuya finalidad es detectar disparos de armas de fuego y de motosierras, a partir del análisis de patrones de audio en una red inalámbrica de sensores, para monitorizar zonas protegidas a lo largo del territorio nacional ([15]).

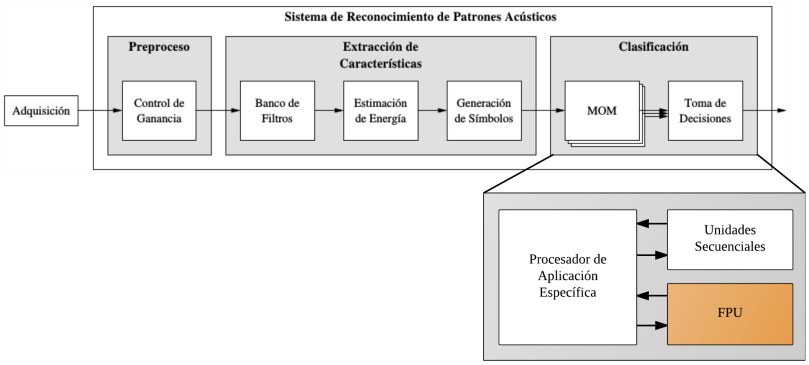

Figura 1.1: Diagrama de bloques del proyecto SiRPA. Tomado y modificado de [1]

El SiRPA es, entonces, la realización en hardware y software de la clasificación de señales que nos lleva la detección de señales. Debido a que el SiRPA se encarga de la extracción e identificación de las características de la señal, el cálculo de probabilidades y la toma de decisiones, es que se necesita la implementación una unidad de coma flotante, dado que el algoritmo matemático, Modelos Ocultos de Markov [16] (HMM por sus siglas en inglés), que se escogió para la implementación del SiRPA, requiere procesar datos en coma flotante.

La solución propuesta ha sido la de un procesador de aplicación específica (ASP), basado en el conjunto de instrucciones RISC-V, que requiere un coprocesador de coma flotante para

Tabla 1.1: Diferentes características de la FPU utilizada en el SiRPA y su respectiva latencia en el tiempo

Rasgos	Versión 1	Versión 2	FPUs en la industria	Versión 3
FPADD/FPSUB	46	16-9	??	??
FPMULT	47	14	??	??
FPCOS/FPSEN	-	437	??	??
Verificacion FPGA	✓	✓	??	??
Verificacion 0.13μm	×	×	??	??

acelerar el desarrollo de los algoritmos. Para el manejo aritmético de las operaciones en coma flotante se hace uso del estándar IEEE-754, el que define formatos tanto en números binarios flotantes de precisión simple, como de precisión doble.

1.1 Problema

En respuesta a la necesidad de operación en coma flotante del proyecto SiRPA, el entonces estudiante Diego Rodriguez realiza la primera versión de la FPU, descrita en [17]. Rodriguez realiza una propuesta de diseño para la suma/resta y la multiplicación en coma flotante, y esta es implementada y verificada para una FPGA. Desafortunadamente, esta implementación no logra un tiempo de ejecución ni una utilización de recursos satisfactorio.

Seguidamente, Francis López realizó como su proyecto de graduación mejoras sobre el trabajo de Rodríguez [13]. Las mejoras de López consistieron en la adición de un Desplazador de Barril, una reducción en la complejidad de la máquina de estados y la utilización de la multiplicación de Karatsuba en la operación de la mantisa del multiplicador en coma flotante. Estas mejoras resultaron en reducciones en la latencia, y la utilización de recursos de ambos operadores sobre FPGA.

A su vez, Jeffrey Quirós le agregó a la FPU en [5] el soporte de operaciones trigonométricas, además del manejo de excepciones. Este par de unidades fueron propuestas, implementadas en RTL, sintetizadas y verificadas en FPGA.

Finalmente, con lo que se cuenta actualmente son diferentes diseños en RTL de operadores en coma flotante, sintetizados y verificados en FPGA. El primero problema que se presenta está relacionado a los requerimiento del proyecto SiRPA, ya que este debe de ser fabricado en una tecnología comercial, por lo cual este diseño que se encuentra actualmente en RTL debe de ser sintetizado en esta tecnología y verificado.

Adicionalmente, el rendimiento del diseño actual no se ha contrastado con otras FPUs presentadas en la literatura, lo cual inhibe el análisis para la mejora en términos de procesamiento de datos del sistema, número de etapas de segmentación a utilizarse, entre otros.

Finalmente, el SiRPA tiene como requerimiento central la utilización de baja potencia, de-

bido al ambiente físico en el cual se espera operar el dispositivo [16]. En consecuencia, la sección del SiRPA que ejecuta el algoritmo HMM, el cual utiliza la FPU, se encuentra en estado de hibernación la mayor parte del tiempo [3].

Es unicamente en ciertos casos especiales que esta sección sale del estado de hibernación para ejecutar el algoritmo de HMM, y una vez completado este, vuelve al estado de hiberación [16]. Es por esta razón que el algoritmo debe ser procesado lo más rápido posible, ya que mientras más rápido se ejecute este algoritmo, menos potencia es disipada [18].

Reducir la potencia consumida es el parámetro central que se busca mejorar. Adicionalmente, el aumentar el procesamiento de datos de los operadores en coma flotante, disminuye el tiempo de ejecución del HMM [3], por lo cual se busca el reducir estos parámetros en la FPU, mientras se busca el reducir el área (para bajar la potencia dinámica) y disminuir la conmutación en los puertos del circuito (para disminuir la potencia dinámica) [18].

1.2 Objetivos

1.2.1 Objetivo general

Proponer una metodología para contrastar contra *benchmarks* reconocidos en la literatura, en términos de área, tiempo de ejecución y consumo de potencia, la implementación en HDL (lenguaje de descripción de hardware) de una unidad de coma flotante con funciones aritméticas y no lineales, obteniendo una unidad escalable y flexible.

1.2.2 Objetivos específicos

- Proponer al menos tres mejoras en la descripción en HDL de la FPU, utilizando los insumos recolectados en el objetivo anterior, y mejoras propuestas en los trabajos anteriores al actual, o aquellas halladas durante este trabajo.
- Sintetizar la descripción en HDL de la FPU haciendo utilización de una biblioteca de celdas 0.13um, usando aquellas optimizaciones en potencia, área y velocidad disponibles en la herramienta de síntesis a usar, dentro de las restricciones generales del proyecto macro al que pertenece este proyecto específico.
- Proponer un ranking de soluciones propuestas existentes en la literatura contra las cuáles contrastar la FPU bajo desarrollo.
- Proponer un flujo metodológico para realizar una comparación eficiente del rendimiento computacional de una unidad de coma flotante escalable y flexible, basada en un *benchmarking* establecido en los objetivos anteriores.

Capítulo 2

Marco Teórico

A continuación, se describe un perfil general de la estructura del trasfondo teórico de la presente tesis de licenciatura. Primero, se explorará las generalidades pertinentes al formato IEEE 754 y su correspondiente representación de los números a su una diferente precisión, las excepciones, y al redondeo.

Además, se describe la teoría sobre la cual se basan las mejoras presentadas en el capitulo 5.

2.1 Estándar IEEE 754

El estándar IEEE 754 especifica formatos, métodos, y excepciones para aritmética en coma flotante en sistemas computacionales, permitiendo obtener el mismo resultado, dado el mismo dato de entrada, sin importar si el proceso es llevado a cabo en hardware, software, o una combinación de ambos.

Para esto, los especificadores del estándar decidieron utilizar un formato de notación científica representado de esta forma un número segmentando este en tres partes: signo, potencia (o exponente), y la magnitud (o el significando).

Cabe también recalcar que este estándar es uno de los más utilizados y aceptados para la realización de computaciones en coma flotante.

2.1.1 Formatos de representación

La representación para cada valor binario se basa en que todo valor real puede representarse en notación científica, como se muestra en la ecuación (2.1) donde **s** es el signo del número, **b** es la base de la notación científica, **E** es el exponente del valor, y **M** representa la mantisa o significando (en este caso, se utiliza la notación "1.M" para ejemplificar la normalización parte del formato).

$$x_{\mathbf{b}} = 1.\mathbf{M} \times \mathbf{b}^{\mathbf{E}} \tag{2.1}$$

Los formatos de representación de números en coma flotante son utilizados para representar un subconjunto finito de números reales para su utilización y almacenado en sistemas computacionales. Estos se caracterizan principalmente por tres especificaciones:

- **Base**: en el caso del estándar, puede ser binaria o decimal.
- **Precisión**: describe la precisión del formato, que al mismo tiempo significa el ancho de la palabra. Este puede ser *half*, simple, doble, y extendida en el estándar.
- **Rango del exponente**: rango de números representables por el mismo exponente. Esta característica va directamente relacionada al sesgo (*bias*) que se le agrega al mismo exponente para utilizar únicamente números positivos en el exponente.

El estándar IEEE 754 cuenta con los siguientes posibles formatos básicos:

- Tres formatos binarios con longitud de palabra de: 32, 64 y 128 bits.
- Dos formatos decimales con longitud de palabra de: 64 y 128 bits.
- Una representación para cero ($Zero$), ∞ ($Overflow$ o sobredesborde) y $-\infty$ ($Underflow$ o subdesborde).
- Representaciones para los números que exceden los límites del formato o no pueden ser representados como números en el formato (NaN, $qNaN$ y $sNaN$).

A continuación, la tabla 2.1 resume los diferentes tipos de formatos representables en el estándar IEEE junto con el sesgo de cada forma, sus exponentes límites (e_{max} y e_{min}), representaciones numéricas límite (b_{max} y b_{min}), y la representación del sobre-desborde y sub-desborde (∞ y $-\infty$).

Tabla 2.1: Valores de los parámetros que definen los formatos básicos de números de coma flotante, para tres formatos binarios y dos formatos decimales. Fuente [14]

	Formato binario (b = 2)			Formato Decimal (b = 10)	
	simple	doble	doble ext.	decimal 64	decimal 128
M	24	53	113	16	34
e_{max}	+127	+1023	+16383	384	6144
e_{min}	-126	-1022	$\leq$16382	384	6144
b_{max}	$7F7FFFFF_h$	$7FEFFFFFFFFFFFFF_h$			
b_{min}	00800000_h	10000000000000_h			
$Sobredesborde$	$7F800000_h$	$7FF0000000000000_h$			
$Subdesborde$	$FF800000_h$	$FFF0000000000000_h$			

M: Precisión o largo de la mantisa, e_{max}: Máximo del exponente, e_{min}: mínimo del exponente, b_{max}: Límite máximo, b_{min}: Límite mínimo, $Sobredesborde$, $Subdesborde$

$$\underbrace{Signo}_{1bit}\underbrace{Exponente}_{5bit}\underbrace{Mantisa}_{10bit} \tag{2.2}$$

$$\underbrace{Signo}_{1bit}\underbrace{Exponente}_{8bit}\underbrace{Mantisa}_{23bit} \tag{2.3}$$

$$\underbrace{Signo}_{1bit}\underbrace{Exponente}_{11bit}\underbrace{Mantisa}_{52bit} \tag{2.4}$$

2.1.2 Excepciones

El punto de cualquier tipo de representación física del sistema de números reales, se define un subconjunto de números los cuales pueden representarse por un cierto formato, en este caso, cualquiera de los formatos del estándar IEEE. Además de un subconjunto de números representables, se ha comentado sobre la la representación que tiene cada formato de los números límites (*sobredesborde* y *subdesborde*).

Un ejemplo claro a un número no representable es la división con cero. Existen dos soluciones: definir un número muy grande como resultado de tal operación, lo cual pierde sentido al realizar 2/0 y 1/0. La otra opción es crear una bandera de notificación que le permita al programador encontrar tal situación y transferir la responsabilidad de esta tarea al programador.

Existen ciertas operaciones como las comentadas anteriormente ($\infty * 0$, ∞/∞ ó $0/0$), las cuales no tienen un resultado claro. El estándar IEEE define el resultado de estas operaciones como *NaN* o *Not a Number* (No un número).

Algunas excepciones que se pueden mencionar, son las siguientes:

Operación Inválida: Este es señalado por una bandera, usualmente denomida en la literatura como *INVALID* [2], cuando algún resultado yace fuera del rango representable por el formato, y cualquier otro valor o infinito causaría confusión. Este es definido como *NaN*.

Algunos ejemplos de operaciones son: $\sqrt{Negativo}$, $\infty * 0$, ∞/∞, etc.

DIVISIÓN entre CERO: Son aquellas operaciones las cuales tienen como resultado un infinito. Algunos ejemplos de estas operaciones son: $(\pm 0)^{(Negativo)}$, $LOG(\pm 0)$, $ATANH(\pm 1)$, etc.

SOBREDESBORDE: Esta excepción indica que el resultado yace más allá del rango representable, y por lo tanto se da un infinito como resultado. Esta es una aproximación, y como lo comenta [19], es una muy imprecisa.

SUBDESBORDE: Esta bandera ocurre después de intentar aproximar un resultado diferente de cero que es más cercano a cero que el número resultante de la operación.

2.1.3 Modos de redondeo

Las operaciones realizadas con el estándar IEEE 754 requieren del uso de bits adicionales, en la palabra de los operandos, para propósitos de precisión y eficiencia en memoria. Estos son el bit implícito, el bit de redondeo, el bit de guardia y el bit pegajoso.

El bit implícito es el bit más significativo en la mantisa, y se elimina del formato IEEE ya que se sabe de antemano que cualquier número normalizado debe de tener en el bit más significativo un bit en 1. A la hora de procesar los números en alguna operación, este bit es concatenado en la trama.

Por otro lado, el bit de redondeo, el de guardia y el sticky bit se ubican en la parte menos significativa del significando, inicialmente en 0, y se usan para evitar la perdida de precisión inherente de la representación de números reales en un formato finito. Estos bits de precisión permiten, a la hora de procesar un número con una cierta operación, utilizar alguno de los siguientes modos de redondeo:

REDONDEO HACIA $+INFINITO$**:** El resultado se redondea por exceso hacia el infinito positivo.

REDONDEO HACIA $-INFINITO$**:** El resultado se redondea por defecto hacia el infinito positivo.

REDONDEO HACIA $CERO$**:** Si el resultado es un número positivo, se redondea hacia abajo, si el resultado es un número negativo, se redondea hacia abajo. Este redondeo se puede comparar con el truncamiento, y es el más utilizado.

REDONDEO HACIA EL PAR MÁS PRÓXIMO: El resultado se redondea al valor más cercano. De encontrarse a una distancia igual de dos números pares, se redondea hacia aquel número que tenga un 0 en el bit menos significativo.

2.1.4 Operaciones en coma flotante

El estándar IEEE requiere que la adición, sustracción, multiplicación y división sean redondeados exactamente. En otras palabras, los operandos deben de ser procesados, y después redondeado al número flotante más cercano representable en el formato del estándar correspondiente.

Además de las operaciones aritméticas básicas (+, -, $\times$, $\div$), el estándar también especifica la raíz cuadrada y el residuo. En contraste, [14] nos explica que para funciones trascendentales, tales como el seno y el coseno, no es necesario el redondeo, si se toma en cuenta *el Dilema del Carpintero.*

A continuación, se describen los algoritmos para llevar a cabo ciertas operaciones de interés en coma flotante:

Suma/Resta en coma flotante

Para la realización de la suma en coma flotante, se tienen que considerar los números como si se encontraran en notación científica. Sean $A = 1.5 \times 10^{-1}$ y $B = 3 \times 10^{1}$ un par de números reales, y procédase a realizar la suma entre los dos.

El primer paso a tomar, es el asegurar que los dos números tengan el mismo exponente. Por otro lado, si se desea mantener un 1 en el bit más significativo, se deberá realizar el corrimiento de la coma (lo cual conlleva la suma o resta del exponente) sobre el número de menor valor.

$$A = 1.5 \times 10^{-1} \triangleright A = 0.015 \times 10^{1} \tag{2.5}$$

$$B = 3 \times 10^{1} \tag{2.6}$$

Seguidamente, se realiza la suma entre las mantisas, y de esta forma se obtiene el resultado:

$$C = 0.015 \times 10^{1} + 3 \times 10^{1} = 3.015 \times 10^{1} \tag{2.7}$$

El algoritmo 2.1 ejemplifica el procedimiento que se debe de tomar para realizar una suma en coma flotante utilizando el estándar IEEE 754. La variable de entrada $RTIPO$ es el tipo de redondeo a utilizar en la función $REDON(Numero, RTIPO)$, y se pueden utilizar algunos de los ya mencionados en secciones anteriores.

Por otro lado, queremos entregar en el resultado final un número normalizado y por lo tanto se debe de realizar un corrimiento en la mantisa. Esto conlleva una compensación en el valor del exponente, el cual debe tomarse en cuenta.

Finalmente, se concatenan diferentes partes del número a ser representado en el formato.

Multiplicación en coma flotante

A continuación, se tienen $X_{\mathbf{b}}$ y $Y_{\mathbf{b}}$, los cuales son un par de números en formato de coma flotante con base $\mathbf{b}$, si utilizamos la misma notación utilizada en la ecuación (2.1), se obtiene:

$$X_{\mathbf{b}} = 1.M_x \times \mathbf{b^x} \tag{2.8}$$

$$Y_{\mathbf{b}} = 1.M_y \times \mathbf{b^y} \tag{2.9}$$

El resultado de la multiplicación se obtiene al multiplicar el valor de las mantisas y sumar el valor de los exponentes.

$$X_{\mathbf{b}} \times Y_{\mathbf{b}} = (1.M_x \times 1.M_y) \times \mathbf{b^{y+x}} \tag{2.10}$$

Se debe de recordar que cada exponente en estándar IEEE de coma flotante tiene un sesgo intrínsico en su formato, que debe restarse al dar el valor final.

$$X_{\mathbf{b}} \times Y_{\mathbf{b}} = (1.M_x \times 1.M_y) \times \mathbf{b^{y+x}}^{-SESGO} \tag{2.11}$$

1: **procedure** FPADD/FPSUB($Op1$, $Op2$, Res, OP, $RTIPO$) ▷ Operandos 1 y 2, y resultado
2: $Signo_{Res} = SIGNO(Signo_1, Signo_2, OP)$
3: $Mant_{DMP} \leftarrow \{1.Mant_{Op1}\}$
4: $Mant_{DmP} \leftarrow \{1.Mant_{Op2}\}$
5: $Shift_{amnt} \leftarrow Exp_{DMP} - Exp_{DmP}$
6: $Mant_{DmP} \leftarrow Mant_{DmP} >> Valor_{corrimiento}$
7: $Mant_{Denorm} \leftarrow Mant_{DmP} \pm Mant_{DMP}$
8: $Mant_{Norm} \leftarrow NORM(Mant_{Denorm})$
9: $Mant_{Res} \leftarrow REDON(Mant_{Norm}, RTIPO)$
10: $Res \leftarrow \{Signo_{Res}, Exp_{DMP}, Mantisa_{Res}\}$
11: **procedure** NORM($Mant_{Denorm}$,$Mant_{Norm}$)
12: **if** $Sobredesborde == 1$ **then**
13: $Mant_{Norm} \leftarrow Mant_{Denorm} >> 1$
14: **else**
15: $Mant_{Norm} \leftarrow Mant_{Denorm} << LZD(Mant_{Denorm})$

Figura 2.1: Algoritmo para la suma/resta en coma flotante

1: **procedure** FPMULT($Op1, Op2, Res, RTIPO$) ▷ Operandos, resultado y tipo de redondeo
2: $Signo_{res} \leftarrow Signo_1 \oplus Signo_2$ ▷ XOR de los signos
3: $Exp_{Res} \leftarrow Exp_{DMP} + Exp_{DmP} - BIAS$ ▷ Se suman exponentes y se resta el sesgo
4: $Mant_{Denorm} \leftarrow \{1.Mant_{Op2}\} \times \{1.Mant_{Op1}\}$ ▷ Operación entre mantisas
5: $Mant_{Res} \leftarrow NORM(Mant_{Denorm})$ ▷ Se normaliza el resultado
6: $Mant_{Res} \leftarrow REDON(Mant_{Res}, RTIPO)$ ▷ Dependiendo del tipo de redondeo
7: $Res \leftarrow \{Signo_{res}, Exp_{Res}, Mantisa_{Res}\}$ ▷ Se concatena las partes del número

Figura 2.2: Algoritmo para la multiplicación en coma flotante

Finalmente, se deben de tener en cuenta la normalización y redondeo final del resultado, para su correcta representación en el estándar.

$$X_{\mathbf{b}} \times Y_{\mathbf{b}} = NORM(1.M_x \times 1.M_y) \times \mathbf{b}^{\mathbf{y+x}-SESGO} \tag{2.12}$$

2.2 Mejoras planteadas a nivel de hardware

2.2.1 Algoritmo de Karatsuba-Offman recursivo (RKOA)

La multiplicación binaria es una de las unidades más costosas en términos de retraso y área para una unidad de coma flotante [18]. Es por esta razón que López [13] decidió realizar la implementación del multiplicador de Karatsuba simple (KOA).

Este conlleva el realizar tres multiplicaciones simultáneas con la mitad del largo de palabra, en vez de una gran multiplicación con el largo completo de la palabra [20]. Este acercamiento de "divide y vencerás" presenta en la literatura mejoras en área y retardo en implementaciones de multiplicaciones de campo de Galois, gracias a la reducción en la complejidad del mismo sistema [21, 22].

La presente mejora a ser planteada, propone el tomar ventaja de la naturaleza recursiva del mismo algoritmo, y aplicarlo en los submultiplicadores resultados de la primera iteración. Esta recursividad en el algoritmo de Karatsuba es descrito como la implementación recursiva de Karatsuba [23].

2.2.2 Cambio de multiciclo a una segmentada

La implementación de ciertas tareas computacionales (FFT, CORDIC, CPU, etc.) en hardware, conlleva la toma de ciertas decisiones que tienen un gran impacto sobre el desempeño general del algoritmo o tarea (en este caso, de manera análoga, una instrucción) a ser implementada. Entre estas decisiones, se puede mencionar el tipo de arquitectura a elegir.

A continuación, se describirá las arquitecturas uniciclo, multiciclo, y segmentado tomando como referencia la descripción del diseño de un procesador de ISA MIPS tomado de [2] como analogía de un operador aritmético en coma flotante, para fundamentar la mejora propuesta en el presente trabajo, descrita en la sección 5.2.

Arquitectura uniciclo

La implementación de una arquitectura uniciclo pretende la realización de la computación del algoritmo o tarea por medio de segmentos combinacionales en un solo ciclo de reloj.

Tomemos como ejemplo el procesador MIPS uniciclo que se muestra en la figura 2.3. Aquí se muestra la simple implementación de las operaciones *load-store*, operaciones en la ALU, y saltos (*branches*).

Finalmente, aunque el CPI (Ciclos Por Instrucción) de esta arquitectura es 1, la velocidad máxima del reloj que se puede alcanzar va a ser determinada por la ruta combinacional más lenta.

Arquitectura multiciclo

La arquitectura multiciclo se caracteriza por la utilización de unidades combinacionales (ALU, Desplazador de Barril, etc) entre las unidades secuenciales. La utilización de este tipo de arquitectura acarrea la utilización de una unidad multiciclo de control, usualmente algún tipo de implementación de máquina de estado.

El porqué la arquitectura multiciclo es más rápida con respecto a la arquitectura uniciclo se puede analizar con la figura 2.4. De esta, se observa a la arquitectura uniciclo con un

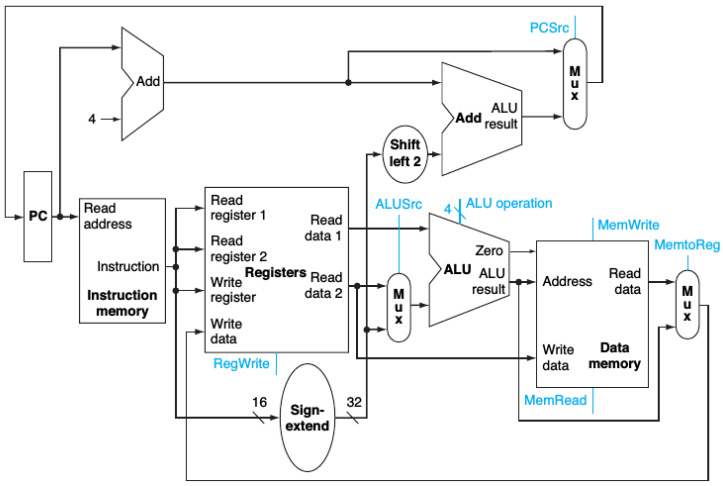

Figura 2.3: El camino de datos para una arquitectura MIPS uniciclo (figura tomada de [2]).

retardo de temporal máximo (la ruta más lenta) de $T_P \approx T_1 + T_2 + T_3$, y por ende, el ciclo de reloj máximo (clk_{max}) a ser utilizado va a estar dado por la ecuación (2.13).

$$clk_{uniciclo} = \frac{1}{T_P} = \frac{1}{T_1 + T_2 + T_3} \tag{2.13}$$

En la Figura 2.4, se puede observar como al segmentar las rutas combinacionales, el ciclo de reloj utilizable se va a incrementar.

$$T_1 < T_2 < T_3 \tag{2.14}$$

Entonces, el ciclo de reloj máximo utilizable por la arquitectura multiciclo se puede apreciar en la ecuación (2.15).

$$clk_{multiciclo} = \frac{1}{T_3} \tag{2.15}$$

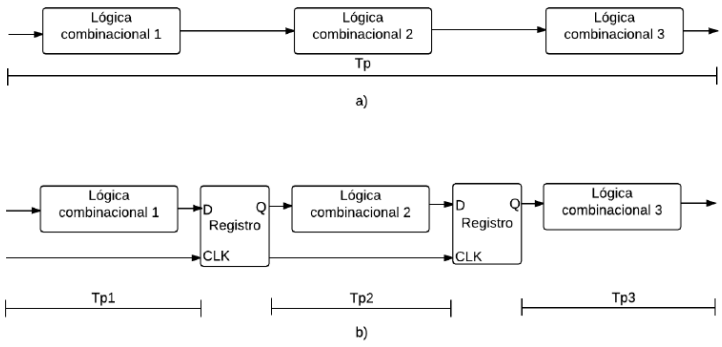

Figura 2.4: a) Arquitectura retardo temporal T_P. b) Arquitectura con retardo temporal de T_1, T_2 y T_3. Figura tomada de [3]

Arquitectura segmentada

La arquitectura segmentada propone el aprovechamiento de las etapas no utilizadas en la ejecución de una instrucción, agregando registros que almacenen el estado completo de cada etapa utilizada por la instrucción. En la figura 2.5 se puede observar una implementación segmentada.

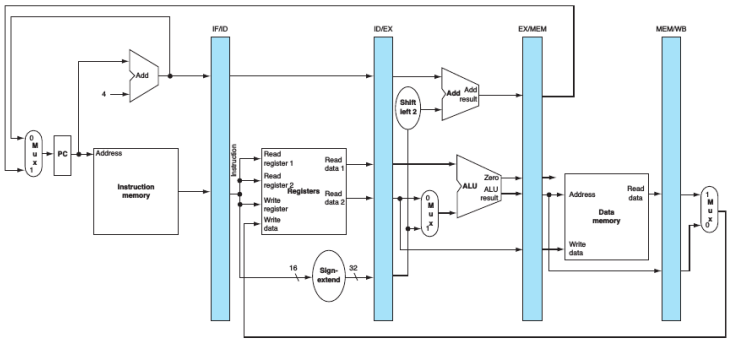

Figura 2.5: Implementación segmentada del procesador tomado de [2].

Un punto importante que debe recalcarse, es como la utilización de cada etapa, en las condiciones indicadas, se ve explotada al máximo (esto es parcialmente cierto en un procesador, debido a las dependencias entre instrucciones), ya que en vez de esperar la ejecución completa de una instrucción (en el caso de la suma en coma flotante, una operación), se ejecutan las instrucciones una detrás de la otra, como se muestra en la figura 2.6.

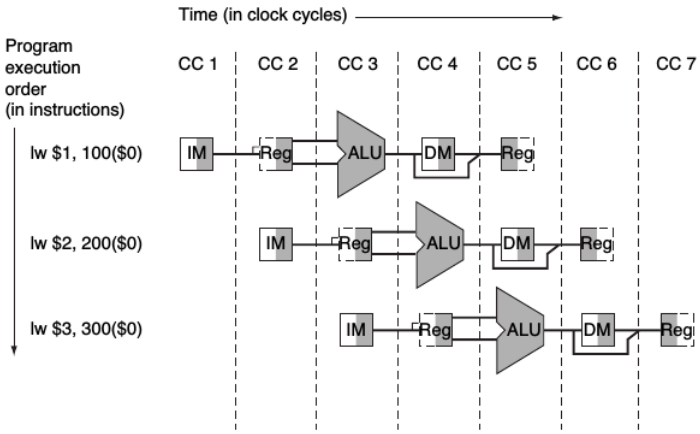

Figura 2.6: Instrucciones la arquitectura descrita en la figura 2.5 siendo ejecutadas en un camino de datos, suponiendo ejecución segmentada. Tomado de [2]

Ventajas y desventajas

El cambio de una arquitectura multiciclo (o iterativa) hacia una arquitectura segmentada, significa un aumento claro en el rendimiento operativo de la unidad bajo análisis ([2]), ya que se pueden realizar más instrucciones (u operaciones en coma flotante en este caso) en

menos tiempo. Por otro lado, al almacenar el estado completo de cada etapa (las señales de control, los resultados intermedios, etc), se obtiene aumento significativo en las unidades secuenciales, lo que representa un aumento en el área del diseño, y el gasto en potencia estática [18].

Además, al aumentar la utilización efectiva de cada segmento del diseño, se obtiene un aumento en la conmutación (el número oscilaciones) del sistema en general, por lo tanto, se puede esperar un aumento en la potencia dinámica utilizada [3, 18].

Para reducir el impacto en el consumo energético del diseño, se puede analizar la posibilidad de disminuir la frecuencia del sistema al recibir el aumento en rendimiento con la unidad segmentada [18].

2.2.3 Algoritmo de CORDIC para operaciones trigonométricas

El algoritmo de Computación Digital para la Rotación de Coordenadas (CORDIC por sus siglas en inglés), fue desarrollada para su utilización en computadoras digitales en tiempo real, donde la mayoría de la computación involucraba la resolución de las relaciones trigonométricas de las ecuaciones de navegación y una alta tasa de exactitud para las relaciones trigonométricas de las transformaciones de coordenadas.

Este método fue por primera vez propuesto en 1957 por Jack Volder [24], y tiene el propósito de calcular funciones trigonométricas por medio de la rotación de vectores. En la literatura se puede encontrar cómo, en estos más de cincuenta años desde su introducción, se ha extendido el CORDIC original para diferentes aplicaciones como la conversión de sistemas coordenados (polar a cartesiano, y viceversa), el cálculo del logaritmo natural de un número, además de su utilización dentro de funciones matemáticas más complejas como la Transformada Rápida de Fourier (FFT) [25] y la transformada del Coseno Directo (DCT) [26, 27, 28].

Tabla 2.2: Funciones resultantes de diferentes configuraciones y modos del algoritmo CORDIC.

Configuración	Rotación	Vectorización
Lineal	$OpY = X * Y$	$OpZ = \frac{X}{Y}$
Hiperbolico	$OpX = \cosh X$	$OpZ = \text{arctanh}$
	$OpY = \cosh Y$	
Circular	$OpX = \cos X$	$OpZ = \text{arctanh}\, Y$
	$OpY = \cos Y$	$OpX = \sqrt{x^2 + y^2}$

La belleza del algoritmo CORDIC yace en que con simples operaciones de corrimiento/suma, se pueden realizar diferentes tareas computacionales, tales como el cálculo de funciones trigonométricas, hiperbólicas y logarítmicas, además de multiplicación, división y raíz cuadrada real y compleja, la obtención de la solución de sistemas lineares, estimación de los *eigenvalores* de una matriz, descomposición de un valor único, factorización QR y muchos otros más. En [29] se puede encontrar una recopilación de la literatura, usos y aplicaciones que se le ha dado al CORDIC.

En la tabla 2.2 se pueden observar algunas de las combinaciones y modos que se pueden utilizar con el algoritmo CORDIC, y su correspondiente operación.

Fundamento teórico

El algoritmo originalmente propuesto por Volder en [24] utiliza la teoría de la rotación de vectores descrito por medio de la ecuación (2.16), y de ahí se lleva esta a la forma que se muestra en la ecuación (2.17). La figura 2.7 ejemplifica este proceso.

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= y\cos\theta - x\sin\theta \end{aligned} \tag{2.16}$$

$$\begin{aligned} x' &= \frac{x - y\tan\theta}{\sqrt{1+\tan\theta^2}} \\ y' &= \frac{y + x\tan\theta}{\sqrt{1+\tan\theta^2}} \end{aligned} \tag{2.17}$$

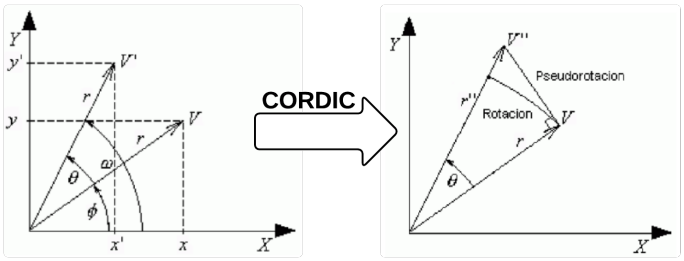

Figura 2.7: Aproximación de la cual se basa el algoritmo CORDIC. Tomado y modificado de [4].

$$\begin{aligned} x_{i+1} &= K_i(x_i - m * y_i d_i 2^i) \\ y_{i+1} &= K_i(y_i + x_i d_i e_i) \\ z_{i+1} &= z_i + d_i \arctan 2^{-1} \end{aligned} \tag{2.18}$$

Finalmente, las ecuaciones unificadas del algoritmo iterativo CORDIC, con las cuales se realizan las iteraciones para obtener las operaciones trigonométricas (coseno, seno, cosh, etc) se pueden observar en la ecuación (2.18), en dónde el sub indice i representa la iteración actual y $K_i = \dfrac{1}{\sqrt{1+2^{-2i}}}$.

Además, en la ecuación (2.18) d_i representa la dirección de la rotación que se esté utilizando. En modo de vectorización, $d_i = -1$ si $z_i < 0$, sino, entonces $d_i = +1$. Si se desea utilizar el modo circular, $d_i = +1$ si $z_i < 0$, sino, entonces $d_i = -1$.

Tabla 2.3: Angulo de rotación para el algoritmo CORDIC. Tomado de [4]

Configuración	e_i
Linear	2^{-i}
Hiperbólico	$\arctanh 2^{-1}$
Circular	$\arctan 2^{-1}$

Por otro lado, m define si la configuración es hiperbólica ($m = -1$), lineal ($m = 0$) o circular ($m = +1$). Finalmente, e_i es el angulo de rotación el cual cambia dependiendo del tipo de configuración. Usualmente el valor de e_i es implementado en hardware utilizando algún tipo de memoria pequeña como una LUT o una ROM.

Tabla 2.4: Configuración de industrialización del algoritmo CORDIC dependiendo la operación matemática deseada

Modo	X	Y	Z
Rotacion circular	0	K_i	Argumento Z
Vectorización circular	Argumento X	1	0
Rotación Hiperbólica	0	K_i	Argumento Z
Vectorización Hiperbólica	Argumento -1	Argumento +1	0
Rotación lineal	Argumento X	0	Argumento Z
Vectorización lineal	Argumento X	Argumento Y	0

En el presente trabajo, se trabaja sobre la implementación realizada por Quirós en [5]. La implementación del algoritmo de CORDIC realizada por Quirós permite el intercambiar entre el modo de rotación o vectorización.

Arquitecturas

Arquitectura bit-paralela iterativa : Es una de las más utilizadas. En esta, se segmentan diferentes secciones combinacionales, tales como la suma de los operandos, el desplazamiento de la iteración anterior y los ángulos pre-cargados.

Esta arquitectura pretende el ingresar las variables X_i, Y_i y Z_i comentados en la ecuación (2.18) de manera paralela. El modo de operación va a ser descrito por un multiplexor que elige el signo de Z_i o Y_i.

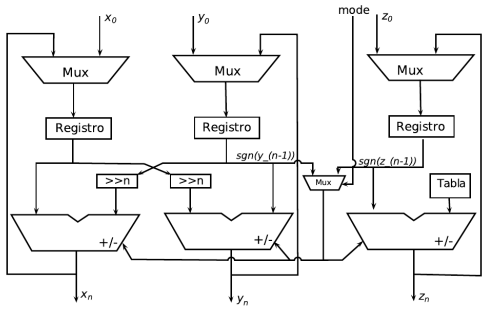

Figura 2.8: Arquitectura bit-paralela iterativa para la implementación en hardware del algoritmo CORDIC. La re-utilización de los componentes por iteración hacen de esta implementación altamente eficiente. Tomado de [4].

La ventaja principal de esta implementación es la utilización eficiente de los recursos, al ser una arquitectura multiciclo, en donde se reutiliza el mismo hardware para realizar las operaciones.

La desventaja principal es la velocidad de operación de la unidad, ya que por iteración, se requiere una numerosa cantidad de ciclos de reloj. Además, esta arquitectura no se puede segmentar, lo cual hace que la utilización de las diferentes unidades no sea la más eficiente.

Arquitectura bit-paralela desplegada: a diferencia de la arquitectura bit-paralela iterativa, la arquitectura desplegada no almacena los estados de cada iteración, sino que cada iteración se convierte en una etapa. Los desplazamientos son fijos, y por lo tanto no se requiere de unidades de corrimiento por iteración.

Este método es muy eficiente cuando se requiere la velocidad, pero el tamaño de la implementación aumenta con respecto al número de iteraciones.

La energía estática en un sistema digital es directamente proporcional al área utilizada, lo cual significa que el gasto energético en esta implementación aumenta con respecto al número de iteraciones. Como el diseño del SiRPA requiere de una utilización baja de potencia, [5] tomó la decisión de implementar la arquitectura bit-paralela iterativa.

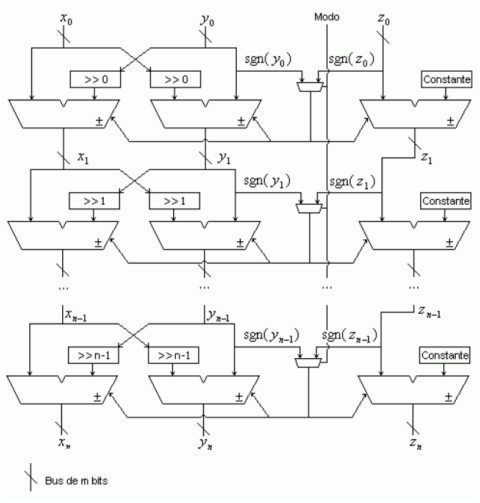

Figura 2.9: Arquitectura bit-paralela desplegada para la implementación en hardware del algoritmo CORDIC. Los componentes de cada iteración se replican, aumentado el consumo de los recursos. Tomado de [5].

Capítulo 3

Estado del arte en unidades de coma flotante

Una unidad de coma flotante (en adelante FPU, por sus signles en inglés: Floating Point Unit) es el principal componente en aceleradores gráficos, DSPs (Procesadores de Señales Digitales), y sistemas computacionales de alto rendimiento. En este caso, este componente es la realización en hardware que el estándar IEEE 754 especifica.

La integración de la presente FPU dentro de un ASP para la implementación del SiRPA es el objetivo final del proyecto macro. Una comparación del rendimiento/potencia/utilización de área se debe realizar para determinar como se compara la presente unidad con FPUs disponibles en la literatura.

Dado el carácter abierto de la FPU a desarrollar, se procedió a analizar FPUs de licencia abierta, con disponibilidad en la red y número de citaciones en la literatura.

Se comentan las unidades de código abierto más destacadas a continuación.

3.1 *Core* Coffee RISC y co-procesador Milk

COFFEE RISC *Core* es un *core* de Instrucción Reducida (RISC) desarrollado en el Instituto de Sistemas Computacionales y Digitales de la Universidad Tampere de Tecnología, Finlandia. A continuación se comentan algunos rasgos del mismo:

- Segmentación de 6 etapas
- Arquitectura de Hardvard
- Precisión completa para 64 bits en 4 ciclos de reloj.
- Interfaz para coprocesadores

Este *core* ha sido altamente documentado, y a través de la interfaz de coprocesador se han desarrollado dos aceleradores, el coprocesador Milk (operaciones en coma flotante) y el

coprocesador Butter (operaciones de multimedia, procesamiento de señales, aplicaciones en 3D).

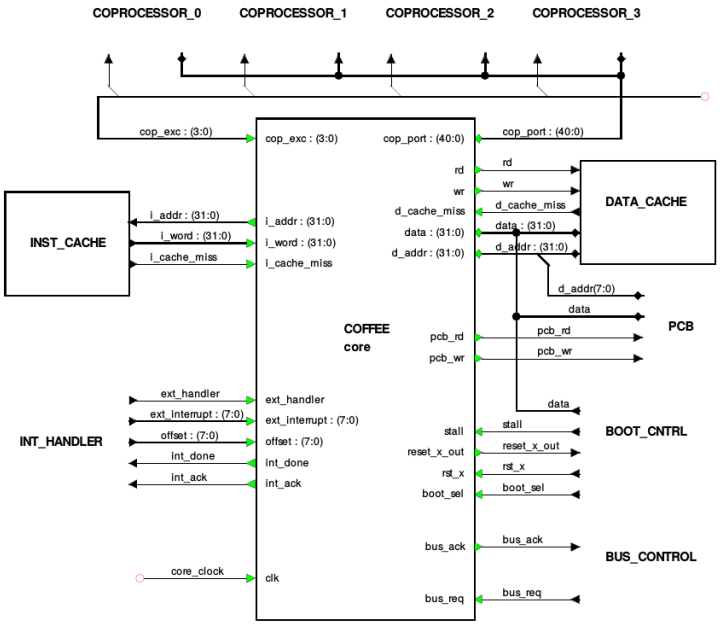

Figura 3.1: Interfaz del *core* Coffee RISC. Tomado de [6]

En la página del proyecto, `http://coffee.tut.fi`, se pueden encontrar los recursos necesarios para analizar al trabajo realizado y ejectuar el flujo creado, incluido el enlazador, compilador de C, simulador del ISA, núcleos de Linux, *binutils* y los aceleradores.

Para el presente trabajo, es de especial importancia el núcleo acelerador Milk, presentado en [30, 31, 6].

En [30], los autores describen la implementación en FPGA del coprocesador Milk junto al procesador de 32 bits, el QRISC. La implementación fue realizada en FPGA (Stratix EP1s40F780C5), y además, se sintetizó en celdas std con una tecnología de 0,13μm (utilizando `Synopsys Design Compiler`).

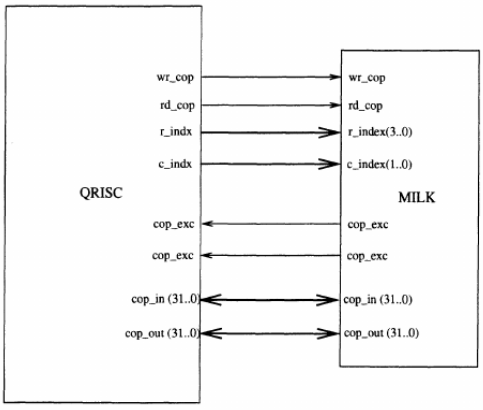

Figura 3.2: Interfaz entre el procesador QRISC y el co-procesador Milk

Seguidamente, la verificación de la misma unidad fue realizada utilizando algunos algoritmos

del *benchmark suite* `DSPStone`. La metodología utilizada para verificar el rendimiento es la siguiente (esta metodología también se comenta en [31]):

- Verificación de comportamiento sobre el diseño en RTL
- Sintetizar para la tecnología 0.13μm en `Synopsys Design Compiler`
- Verificación funcional con vectores (5 millones de operadores).
- Utilizar algoritmos conocidos de procesamiento de señales especificados por el benchmark `DSPStone`.
- Compilar el código fuente en C, utilizando una versión personalizada para el presente sistema basado en GNU GCC Toolchain.
- Simular el sistema sintetizado con la ejecución de las instrucciones con Mentor Graphics Modelsim.

La métrica utilizada para medir el rendimiento, y contrastar el mismo rendimiento con otros de la industria, es la densidad computacional comentada en [32].

$$densidad = (\frac{f_{clk}}{ciclos * CLB}) \tag{3.1}$$

3.2 *Cores* genéricos de OpenCores

OpenCores es una comunidad de software libre de hardware que desarrolla código de hardware abierto, a través de EDA, con un dogma similar al movimiento de software libre. Según el EE Times, a finales del 2008 la página contaba con más de 20,000 suscriptores, en octubre del 2010, llegó a los 95,000 suscriptores y 800 proyectos.

La biblioteca de OpenCores contiene una variedad de elementos de diseño, tales como CPUs, controladores de memorias, periféricos, y aceleradores de hardware, como la FPU. A continuación, se listarán las tres FPU que a la fecha se pueden encontrar en la página:

3.2.1 Usselman FPU

Este fue el proyecto de código libre llevado a cabo por Rudolf Usselmann [7]. Su FPU describe una FPU con soporte para sumas, restas, multiplicaciones, división, conversión fp2int en precisión simple. Este consiste de dos unidades de pre-normalización que puede ajustar los significandos al igual que los exponentes de un dado número, uno para las operaciones de suma/resta y el otro para las operaciones de multiplicación/división. En la figura 3.3 se puede observar un diagrama general de la arquitectura de esta FPU.

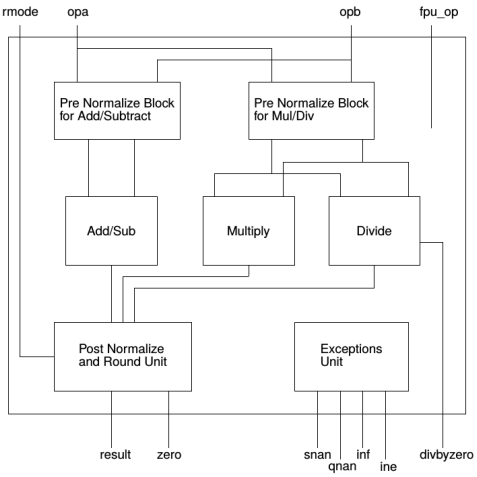

Figura 3.3: Diagrama de bloques del hardware *core* de Usselmann. Tomado de [7]

Por otro lado, de `opencores.org` se pueden encontrar el *core* de FPU de Lundgren (VHDL ,[33]), Marcus (VHDL, [34]) y Jidan (VHDL, [35]). Todas las anteriores tienen diferentes grados de verificación y referenciación en la literatura.

La unidad de Usselmann, al ser una de los primeros *cores* de código libre, es uno de los más representativos en la literatura. A continuación, se listan algunos trabajos que utilizan la FPU de Usselmann como punto de partida para sus diseños, o como punto de comparación:

- En [36, 37], los autores se basan en la FPU creada por Usselmann para crear sus propias mejoras sobre las etapas de operación sobre los significandos (Unidades rápidas de Carry Lookahead de [38], Codificación de Booth Raíz-4 (MBE) de [39], algoritmo para la raíz cuadrada [40, 41]) e implementar así su propia versión.
- En la implementación de FPU realizada en [42], se toma la unidad de Usselmann como referencia para el diseño.
- El multiplicador de [43] en coma flotante de precisión simple, sintetizado en la familia de FPGA Virtex 5 y en tecnología TSMC 180nm, también hace mención de la unidad de Usselmann.
- El acelerador de OpenGL creado en [44] hace utilización de la unidad de Usselmann, en conjunto con un procesador con un ISA VLIW, un *core* de PowerPC. El código fuente ejecutado sobre el procesador y la FPU se puede encontrar en `http://ece545.com/F15/reports/F09_OpenGL.pdf`.

3.3 Unidad de coma flotante con base alta

Esta unidad de coma flotante es presentada por Ehliar en [8]. Esta implementación del estándar de coma flotante realiza una implementación sobre una FPGA Kintex-7, y se basa en la utilización de un formato de base diferente a 2, para obtener una unidad aritmética de

mantisa (tal como un sumador o un multiplicador) simplificada, reduciendo la utilización de recursos sobre FPGA.

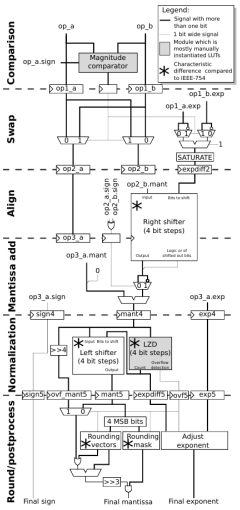

Figura 3.4: Implementación de la suma/resta

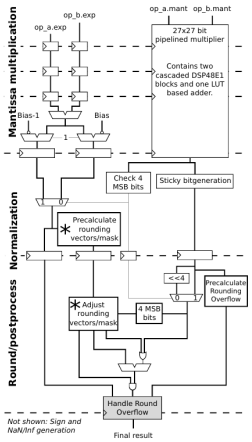

Figura 3.5: Implementación de la multiplicación

Figura 3.6: Diagramas de alto nivel de la implementación de Ehliar. Tomado de [8]

Las comparaciones realizadas con los operadores generador por Xilinx Logicore muestran una mejora en área en el caso del sumador.

El código fuente se puede descargar de [45] para una futura comparación con el presente trabajo.

3.4 FPU basada en bloques DSP48

Esta FPU con soporte para las operaciones de suma, resta y multiplicación descrita en [9] está diseñada específicamente para su funcionamiento con las celdas primitivas *DSP48E1* de la familia Virtex-6 y la serie 7 de FPGAs de Xilinx.

La implementación descrita en [9] comprende la implementación de un operador sumador y multiplicador en coma flotante con celdas LUT, y un operador sumador+multiplicador basado en primitivas DSP. Diagramas de los dos operandos basados en LUT se pueden apreciar en la figura 3.9. Estos fueron implementados como punto de referencia para la comparación con la implementación iterativa de la figura 3.10.

En la figura 3.9 se puede apreciar la arquitectura del operador suma+multiplicación en coma flotante basado en celdas DSP48. Este busca la baja utilización de lógica basada en LUTs.

El código de esta unidad es abierto, y se puede encontrar en `https://github.com/brosser/DSP48E1-FP`.

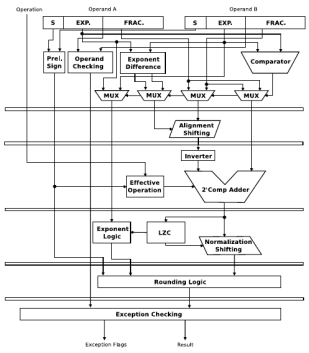

Figura 3.7: Implementación de la suma/resta

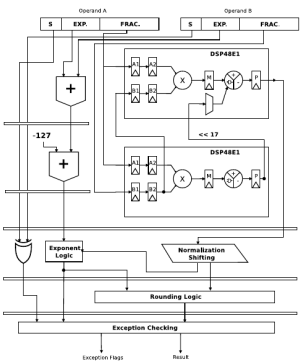

Figura 3.8: Implementación de la multiplicación

Figura 3.9: Diagramas de alto nivel de la implementación de [9].

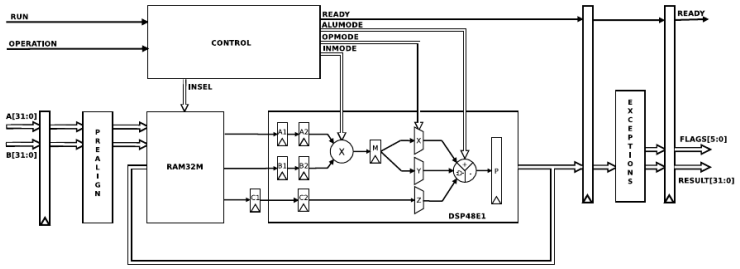

Figura 3.10: Diagrama de bloques de la implementación de FPU iterativa basada en bloques DSP

3.5 Bibliotecas de operadores en coma flotante

3.5.1 Xilinx FPU Core v6.0

LogiCORE IP Cores es una libreria de *cores* parametrizables ofrecidos por Xilinx. Xilinx diseña y mantiene el soporte de todos los *cores*. Estos *cores* corresponden a una implementación optimizada hacia la familia que se este utilizando del *core* que se este necesitando.

Cuando se genera un *core* por medio de la herramienta ISE, esta incluye lo siguiente [10]:

- Una implementación en netlist generada para la FPGA meta.
- Código en VHDL o Verilog para la instanciación del código.
- Un wrapper en VHDL o Verilog para la simulación correspondiente.
- Un simbolo para los esquemáticos.

Entre los cores ofrecidos por Xilinx, uno de particular importancia es el Xilinx Floating Point Operator [46]. El Xilinx FPU Core permite un rango de operaciones aritmeticas sobre FPGA. La operación es especificada al generar el *core*, y cada variante de operación tiene una interfaz común, AXI4 [47].

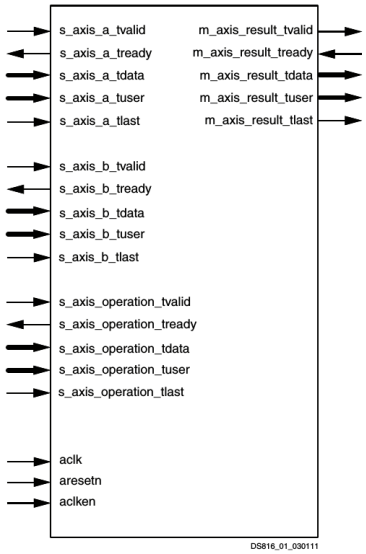

Figura 3.11: Interfaz del Xilinx FPU Core. Tomado de [10].

Este *core* de Xilinx tiene soporte para la multiplicación, suma/resta, división, raíz cuadrada, comparación, recíproco, recíproco de raíz cuadrada, y conversión de punto fijo a flotante y viceversa. Además, tiene un extenso manejo de excepciones y un redondeo del tipo al más cercano.

Un punto importante con respecto a esta FPU, es la flexibilidad de elegir diferentes versiones de un operador en especifico. Además, diferentes versiones de la misma FPU ha sido citada en [48, 49, 8].

Por otro lado, la disponibilidad de utilización de la unidad al mercado en general (habiendo comprado una) y la extensa documentación que caracteriza los productos ofrecidos por Xilinx, hacen que la reproducibilidad de un sistema de comparación de rendimiento sea alto.

3.5.2 `FloPoCo`

`FloPoCo` es un generador de *cores* aritméticos en coma flotante (Floating-Point Cores) para FPGAs.

Uno de los principios centrales de `FloPoCo` es que el procesamiento aritmético en FPGAs no debería asemejar el procesamiento de un procesador. Al diseñar operadores significativamente diferentes, se puede obtener resultados mucho más precisos con menos requerimientos de hardware y en menos tiempo. Esta filosofía es expandida en [50].

El segundo principio por el cual se rige `FloPoCo` es el habilitar el procesamiento *just right* (o adecuado). Esto se refiere a adecuar la precisión y otras características del mismo operador a las necesidades que se tenga en la presente aplicación. Según se comenta en [51], `FloPoCo` no es una biblioteca de operadores, sino, un generador de operadores codificado en C++. Para el usuario, `FloPoCo` es una herramienta de terminal al que se los requerimientos del operador, y este genera HDL completamente sintetizable.

FloPoCo ofrece más de 50 operadores en coma flotante y fijo, cada uno parametrizado en precisión y especificable su frecuencia final. Algunos de estos operadores pueden ser **open-ended** meta-operadores, tales como funciones universales aproximadoras y multiplicadores por constantes arbitrarias. Los operadores numéricos soportados por FloPoCo han sido bien documentados, y se pueden encontrar en ([52]).

```
flopoco -pipeline=no FPAdder 8 23 FPMultiplier 8 23 23
FPDiv 8 23 FPSqrt 8 23

Final report:
|---Entity FPAdder_8_23_uid2_RightShifter
|---Entity IntAdder_27_f400_uid7
|---Entity LZCShifter_28_to_28_counting_32_uid14
|---Entity IntAdder_34_f400_uid17
Entity FPAdder_8_23_uid2
Entity Compressor_2_2
Entity Compressor_3_2
|   |---Entity IntAdder_49_f400_uid39
|---Entity IntMultiplier_UsingDSP_24_24_48_unsigned_uid26
|---Entity IntAdder_33_f400_uid47
Entity FPMultiplier_8_23_8_23_8_23_uid24
Entity FPDiv_8_23
Entity FPSqrt_8_23
Output file: flopoco.vhdl
```

Figura 3.12: Ejemplo del resultado de la generación de una unidad de precisión simple con suma, multiplicación y un divisor sin segmentación. Fuente: Generación propia

Además de generar *cores*, FloPoCo genera **pruebas de banco** a partir de una resumida descripción matemática del funcionamiento esperado del operador. Este también maneja la generación de encauzados correctos por construcción, para que el diseñador se enfoque en la función combinacional. Finalmente, FloPoCo también ofrece la mayoría de los bloques necesitados para crear operadores personalizados.

El presente proyecto es de licencia abierta y código abierto, lo que permite analizar libremente el código, es de fácil manejo y esta especificamente pensado para necesidades personalizadas. La extensiva documentación y la cantidad de citación lo hacen un punto interesante de referencia para la comparación con el presente trabajo.

3.5.3 NEU FPU

En el 2002, el Laboratorio de Computación Reconfigurable lanza la primera versión de VFLOAT en [11] (también conocido en la literatura como NEU FPU), y se han expandido sus capacidades en [53, 54, 55, 56, 57, 58, 59]. La descripción se realizó utilizando VHDL.

Una característica de esta biblioteca de módulos de operaciones es la parametrización dinámica de los mismos módulos, para una implementación ajustada a las necesidades requeridas (potencias, área versus rango, precisión).

Por otro lado, esta biblioteca ha servido como punto de referencia en los trabajos de NoGAP [48], Sandia Labs FPU [60, 49] y [61], lo cual la vuelve un excelente punto de referencia para realizar una comparación de rendimiento en tiempo de ejecución, recursos, temporizado y

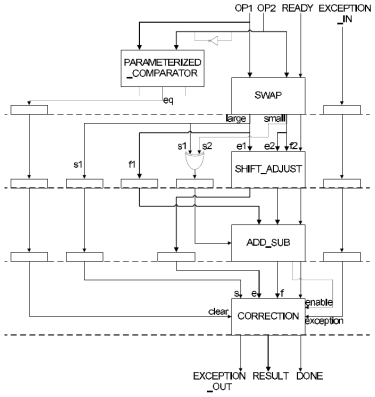

Figura 3.13: Implementación de la suma

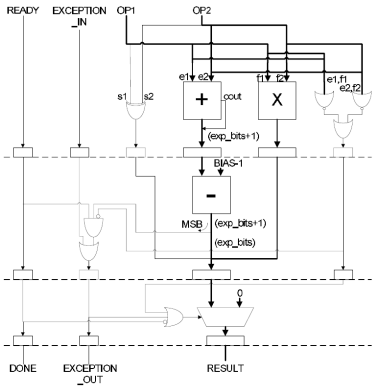

Figura 3.14: Implementación de la suma

Figura 3.15: Diagramas de alto nivel de la implementación de [11].

potencia.

Finalmente, diferentes operadores de la biblioteca VFLOAT se puede descargar en [62], y se utilizan bajo una licencia tipo GNU, la cual le permite al usuario la distribución, y/o modificación del mismo material.

3.6 Descripción histórica del desarrollo de la FPU en el DCILab

La FPU del DCILab, como ya se ha comentado, es parte del proyecto SiRPA. El objetivo de esta unidad es el ser parte de un ASP, el cual ejecuta sobre la arquitectura ISA RISC-V un algoritmo de DSP, los algoritmos Ocultos de Markov.

3.6.1 FPU SIRPA v1

La primera iteración de la presenta unidad fue implementada por Rodriguez en [17]. En esta, tuvo como objetivo la implementación de una FPU la cual tuviera un operador de suma/resta y un multiplicador, un redondeo de truncado, manejo de excepciones *sobredesborde* y *subdesborde*, y descrita en Verilog.

Ambos operadores cuentan con capacidades de redondeo al +Infinito, -Infinito, par próximo y hacia cero. No se cuenta con unidad de manejo de excepciones.

La síntesis y verificación de esta unidad se realizó en Xilinx ISE, para una unidad de FPGA de la familia Artix 7.

3.6.2 FPU SIRPA v2

La segunda iteración de la FPU del SiRPA fue implementada por López [13] y Quirós [5].

Por un lado, López se concentró en la unidad existente, realizando mejoras importantes sobre la unidad de control de la suma y la resta, reduciendo la latencia de ambas unidades. Además, se implementó el algoritmo no-recursivo de Karatsuba-Offman, comentado en la sección 2, una unidad de detección de zero (LZD) y una unidad de desplazadora de barril.

Por otro lado, Quirós se concentró en la creación de una unidad de operaciones trigonométricas basadas en la implementación del algoritmo CORDIC en coma flotante. Adicionalmente, se creó una unidad de detección de números no definidos (números NaN), según lo especificado por el formato IEEE 754.

La unidad de CORDIC utiliza un sumador en coma flotante para el cálculo de las variables X, Y y Z. Por esta razón, se hace utilización de la unidad de suma de López, y el CORDIC fue diseñado de tal forma que se le puede agregar un *core* sumador de coma flotante "multiciclo" cualquiera.

3.6.3 FPU SIRPA v3

En el presente trabajo se han realizado mejoras sobre lo realizado por [13, 5, 17]. Se tiene una unidad de coma flotante, esta contiene unicamente el operador de suma, resta y multiplicación [63].

La justificación detrás de la necesidad de las mejoras se basan en el requerimiento de baja utilización de pontencia del proyecto macro, el SiRPA. Esta restricción de potencia no se ha alcanzado de manera satisfactoria, lo que lleva a la necesidad de propuestas de mejoras en el rendimiento y el área de los diseños.

Los resultados expuestos en el presente trabajo se basan en la versión de la FPU que contiene el operador de suma, resta, multiplicación y el CORDIC. Una descripción detallada de estas mejoras se encuentra en el capítulo 5.

Tabla 3.1: Diferentes características de la FPU utilizada en el SiRPA y su respectiva latencia en el tiempo

Rasgos	Versión 1	Versión 2	Versión 3
FPADD/FPSUB	46	16-9	13(3/6)
FPMULT	47	14	14
FPCOS/FPSEN	-	437	138
Verificacion FPGA	✓	✓	✓
Verificacion 0.13μm	×	×	✓

3.7 Contraste de FPUs en la Literatura

En la tabla 3.2 se encuentra un listado de diferentes FPUs que se consideran representativas en la literatura. De esta tabla se puede observar que solamente las bibliotecas de VFLOAT,

Tabla 3.2: Tabla comparativa de los rasgos de diferentes FPUs

FPU	FPADD/ FPSUB	FPMULT	FPDIV	OTROS	Parametrizado	Código Libre
GRFPU [64]	✓	✓	✓			
ARM VFP9-S[64]	✓	✓	✓	✓		
ARM VFP11 [64]	✓	✓	✓	✓		
AMD K7 [64]	✓	✓	✓	✓		
MEIKO [64]	✓	✓	✓			
USC (MONARCH) [65]	✓	✓	✓	✓		
USC (DIVA) [66, 65]	✓	✓	✓	✓		
Sandia [60]	✓	✓				
NEU [11]	✓	✓	✓	✓	✓	✓
[61]		✓				
[49]	✓					
[67]		✓				
Base Alta [8]	✓	✓				✓
Milk [30, 31, 6]	✓	✓	✓	✓		✓
Xilinx Logicore [10]	✓	✓	✓	✓	✓	
NoGap [48]	✓					
FloPoCo [50]	✓	✓	✓	✓	✓	
DSP48 [9]	✓	✓				
[68]		✓				
[69]	✓	✓	✓	✓		
NTNU [70]	✓	✓	✓			✓

`FloPoCo` y Xilinx Logicore presentan diseños parametrizables. Además, puede observarse en la última columna de la misma tabla como las unidades de NEU, Base Alta, Milk, NTNU y las unidades de OpenCores son de código libre.

Por otro lado, para una efectiva comparación se requiere de un mismo marco de comparación, una misma familia de FPGA o una misma tecnología de síntesis. Es por esto que en la tabla 3.3 se listan la tecnología o familia de FPGA en dónde se reportan los resultados de rendimiento de cada unidad.

Finalmente, de la tabla 3.3 puede observarse que las plataformas más utilizadas a nivel de FPGA son las de Xilinx, especificamente la Virtex II y V.

Tabla 3.3: Análisis de plataformas donde se han reportado los resultados de diferentes FPUs

FPU	FPGA					ASIC		
	Altera	Virtex-II	Virtex-IV	Virtex-5	Virtex-6	0.18μm	0.13μm	90nm
GRFPU [64]		✓					✓	
ARM VFP9-S[64]						✓		
ARM VFP11 [64]								
AMD K7 [64]							✓	
MEIKO [64]						✓		
USC (MONARCH) [65]						✓		
USC (DIVA) [66, 65]		✓	✓			✓		
Sandia [60]		✓	✓					
NEU [11]								
[61]		✓	✓					
[49]		✓		✓				
[67]		✓	✓	✓				
Base Alta [8]								
Milk [30, 31, 6]	✓	✓					✓	✓
Xilinx Logicore [10]		✓	✓	✓	✓			
NoGap [48]		✓	✓					
FloPoCo [50]								
DSP48 [9]					✓			
[68]			✓					
[69]	✓			✓				
NTNU [70]	✓							

Capítulo 4

Diseño de una metodología para la medición del rendimiento computacional

En esta sección se comentará sobre la metodología que se debe utilizar para medir efectivamente el rendimiento de una FPU con especial importancia en la potencia utilizada.

En la primera sección se discutirán sobre algunos *benchmarks* que son estándar en la industria y otros que son código libre. De estos *benchmarks* presentados, se discutirá la elección de uno en específico al final de la misma sección, con base en la literatura y características de relevancia del mismo ASP.

Seguidamente, se comenta sobre los diferentes aspectos de rendimiento a medir del ASP, y cómo se han tratado en otros trabajos de la literatura.

Finalmente, se exponen los pasos a seguir y las métricas a utilizar para efectivamente comparar el rendimiento de el ASP del SiRPA y la FPU del mismo microprocesador.

4.1 *Benchmarks* computacionales

4.1.1 Generalidades

Un *benchmark* es "un estándar de medición o evaluación" [71]. Un *benchmark* computacional es típicamente un programa de computadora que realiza un conjunto de operaciones definidas, o *workload*, y retorna algún tipo de resultado, una *métrica*, describiendo como el sistema probado desempeño.

Las métricas de *benchmarks* computacionales usualmente miden la *velocidad*, qué tan rápido se completó el *workload*; o el *rendimiento*: cuantas unidades de *workloads* por unidad de tiempo fueron completadas.

El ejecutar el mismo *benchmark* en múltiples sistemas computacionales permite crear una

comparación entre los diferentes sistemas.

4.1.2 Tipos de programas *benchmark*

- **Programas de uso cotidiano**

  Estos son aquellos programas que son utilizados con regularidad por usuarios de estaciones compuacionales de propósito general. Estos tienen entradas, salidas y opciones fácilmente seleccionado por el usuario a la hora de la ejecución. Entre otros, se pueden mencionar compiladores de C, software de procesamiento de texto, y Photoshop.

- **Aplicaciones modificadas (o alteradas por *scripts*)**

  Estos son aquellos *benchmarks* compuestos por aplicaciones de uso cotidiano con importantes modificaciones o scripts que estimulan cierto sistema para medir su rendimiento a cargas computacionales comparables a las de un sistema en su ejecución normal.

- ***Benchmarks* de núcleo**

  Estos son piezas funcionalmente importantes de código de un programa, con los cuales se quiera medir el rendimiento de un sistema. Estos difieren con los *benchmark* de programas de uso cotidiano, en que un usuario no utilizaría programas de kernel para realizar una tarea del día a día.

- ***Toy benchmarks***

  Los *toy benchmarks* son pequeños programas computacionales entre las 10 y 100 líneas de código. Un punto importante de estos es que corren un algoritmo o procesamiento sobre el cual usualmente se conoce de antemano el resultado, y son fácilmente portables.

- ***Benchmarks* sintéticos**

  Los *benchmarks* sintéticos siguen la filosofía de los programas de núcleo al intentar igualar el número de operaciones y operadores de un conjunto de programas.

  Cabe mencionar, que este tipo de *benchmarks* intentan emular una carga computacional al igualar un perfil de ejecución computacional de un programa real, pero sin dejar de mostrar características fundamentalmente diferentes, lo que resulta en un sesgo con respecto al rendimiento real del sistema. Por otro lado, estos han encontrado un nicho al ayudar caracterizar la carga computacional de microprocesadores.

- ***Benchmarks* para sistemas empotrados**

  Este tipo de *benchmark* se preocupa por medir el rendimiento de aspectos importantes de sistemas computacionales empotrados. Aspectos como la utilización de memoria, el I/O, y la utilización energética del sistema son de particular importancia.

4.1.3 *Benchmark suites*

Estándares de la industria

En la industria del procesamiento computacional, la medición y comparación del rendimiento de diferentes sistemas es clave para obtener el sistema óptimo para la aplicación a mano, y desde que el momento que hubo variedad, se han necesitado *benchmarks* estándar.

Para este fin, se han creado consorcios de compañías interesadas en la creación de medio para la comparación de los sistemas computacionales. A continuación, se comentan algunos de los estándares de la industria:

`SPEC` : La Corporación de Estandarización de Evaluación del Desempeño (SPEC, por sus siglas en inglés), es una organización estadounidense sin ánimo de lucro que pretende, desde su nacimiento en 1988, el "producir, establecer, mantener y apoyar un conjunto estandarizado" de *benchmarks* de desempeño. Los *benchmarks* producidos por SPEC son altamente utilizados en la industria para la evaluación de sistemas computacionales [72].

Entre sus listas de *benchmarks*, el que más nos conviene utilizar para el fin de evaluar nuestra FPU es el SPECfp2006. Este comprende 17 programas computacionales escritos en C, Fortran y C++. Este benchmark se concentra en probar el rendimiento de sistemas computacionales en el contexto de computación de propósito general, al igual que aplicaciones científicas.

Para utilizar este benchmark se debe de obtener una licencia y su documentación se puede encontrar en `https://www.spec.org/cpu2006/docs/`.

`EEMBC` : `EDN Embedded Microprocessor Benchmark Consortium` (o *EEMBC*, pronunciado como *embassy*, en inglés) desarrolla *benchmarks* para ayudar a diseñadores de sistemas embebidos a elegir el procesador correcto para su aplicación específica. `EEMBC` tiene *suites* de *benchmarking* específicos para la nube y *big data* (`ScaleBench`), dispositivos móviles (teléfonos celulares y tabletas, `ANDEBench`), networking, microcontroladores de ultra baja potencia (`ULPBench`), el Internet de las Cosas (`IoT-CONNECT Benchmark`), medios digitales (`DenBench`), el sector automotriz (`AutoBench`), y en otras aplicaciones.

Por otro lado, `EEMBC` también tiene *benchmarks* para el análisis del rendimiento general de un sistema, incluyendo `Coremark`, `MultiBench` (multi-procesadores) y `FPMark` (coma flotante).

Finalmente, la utilización de cualquiera de los *benchmarks* comentados requieren el uso de una licencia propietaria. Documentación, reportes de rendimiento y contacto se puede obtener en la página del consorcio `http://www.eembc.org/`.

`BAPCo` : la Corporación para el Rendimiento de Aplicaciones de la Industria (`BAPCo`, por sus siglas en inglés), es un consorcio sin fin lucrativo con el objetivo de desarrollar

y distribuir un juego de *benchmarks* de rendimiento para sistemas computacionales personales, con énfasis en aplicaciones de software y sistemas operativos populares.

Entre sus *suites* de *benchmarking* más representativos, se pueden mencionar `MobileMark 2014` (dispositivos móviles), `TabletMark` (tabletas) y `Sysmark 2014` (Windows y iOS).

Entre los miembros de `BAPCo` se puede mencionar: Acer, ARCIntuition, ChinaByte, CNET, Compal Electronics, Dell, Hewlett-Packard, Hitachi, Intel, LC Future Center, Lenovo, Microsoft, Western Digital, Wistron, Samsung, Sony, Toshiba, Zol.

Finalmente, para encontrar más información sobre los resultados y los otros tipos de *suites* que ha desarrollado el consorcio, se puede accesar en `https://bapco.com/`.

`BDTI` : los productos y servicios de BDTI (`Berkeley Design Tecnology Inc.`) proponen el ayudar a ejecutivos, equipo de marketing, e ingenieros a crear y utilizar tecnología, software y herramientas para el procesamiento embebido. Entre los productos que ofrecen, se encuentra *benchmarking* de procesadores embebidos.

En el rango su repertorio de *suites*, se pueden encontrar *benchmarks* específicos para programás núcleo de DSP, Codificado Decodificado de Video, Flujo óptico, y más. [73].

Software libre

Uno de los problemas de los *suites* de *benchmark* estándar en la industria es la restricción intelectual sobre los mismos. Esto limita a la comunidad científica, por lo que se han realizado esfuerzos por crear estándares software libre de *benchmarking*, para diferentes sectores computacionales.

A continuación, se presentan algunos *benchmarks* dignos de mención para la presente aplicación (DSP con procesamiento de operaciones flotantes) y otros de referencia histórica:

`LINPACK` : la biblioteca `LINPACK` es otra manera más de medir la potencia computacional de un sistema de coma flotante. Introducido por Dongarra, Bunch, Moler y Stewart en 1979, `LINPACK` mide la rapidez con la cual el sistema resuelve una ecuación lineal de n por n, $Ax = b$.

Es importante notar que este *benchmark* está siendo actualmente utilizado por el *Top500*, que es un ranking de las super computadoras más poderosas del mundo. Una de las razones por las cuales se utiliza en el Top500, y por el cual es una excelente *benchmark* a utilizar, es debido a que es extensamente utilizado en la industria. Adicionalmente, las métricas de otros sistemas medidos con este *benchmark* son fácilmente encontradas en la red [74].

Uno de los principales problemas de `LINPACK` es la falta de una estandarización en la metodología de reportar los resultados, y un lugar centralizado para la visualización de los mismo resultados. Código fuente de la biblioteca `LINPACK` se puede encontrar en `https://www.math.utah.edu/software/linpack.html`.

`FBench & FFBench` : `FBench & FFBench` son un par de *benchmarks* núcleo codificados por John Walker. `FFBench` utiliza el cálculo de la transformada de Fourier sobre una matriz de 256x256 con una precisión doble, seguido por la transformada inversa. `FBench` es la implementación de un algoritmo de trazado de rayos óptico, una aplicación intensiva en procesamiento de cálculos en coma flotante.

En 1980 se hizo la introducción de ambos *benchmarks*, y se puede descargar el código fuente de ambos en [75].

`Whetstone` : este *benchmark* sintético es una mezcla de procesamiento de números enteros y en coma flotante, funciones trascendentales, saltos condicionales, llamadas a funciones, e indexación de arreglos. El código se puede encontrar en Fortran o C. Este *benchmark* es considerado un clásico y está diseñado para representar programas científicos comunes.

Este *benchmark* no se utiliza en los sistemas computacionales de uso general, ya que su tamaño, variabilidad de código y tipo lo han vuelto poco útil. Por otro lado, aún se puede encontrar su utilización en la medición del rendimiento de microcontroladores (MSP430 de TI [76]).

`MiBench` : Presentado en el 2001, `MiBench` [77] es un suite de programas que se asemeja el modelo de los *benchmarks* de `EEMBC` (dividido los sectores de Control Automotriz e Industria, Redes, Dispositivos Mobiles, Automatización en la Oficina y Telecomunicaciones), pero su creación viene a raíz de una necesidad de un *benchmark* suite que sea código abierto y con caracterización para *workloads* en aplicaciones embebidas.

Este *benchmark* analiza a profundidad los puntos importantes de un sistema embebido, tales como la distribución de instrucciones (dividiendo las instrucciones en 4 categorías: instrucciones de control, operaciones en números enteros, operaciones de números en coma flotante e instrucciones de la memoria) en el código del programa, análisis de saltos y el comportamiento de la memoria.

En código fuente de `MiBench` se puede encontrar sin costo alguno en `http://vhosts.eecs.umich.edu/mibench//`.

`DSPStone` : `DSPStone` es un conjunto de programas para el *benchmarking* de sistemas de procesamiento de señales digitales (DSP). Primera vez introducido en [12], este presenta una metodología para analizar el procesadores DSP y compiladores para los mismos.

Este *benchmark* está compuesto por 3 juegos de programas: ***Benchmarks* de aplicación** (Aplicaciones específicas a DSP), **DSP- *Kernel Benchmarks*** (*Kernels* de DSP como filtros FIR/IIR, FFTs, etc.) y **C- *Kernel Benchmarks*** (estructuras cotidianas de C).

Seguidamente, `DSPStone` propone el evaluar el compilador a ser utilizado al compilar los *suites* descritos anteriormente con respecto a un código en ensamblador (este es usualmente proveído por el proveedor de DSPs), como se muestra en 4.1.

La métrica para medir el rendimiento del compilador se basa la medición del tiempo de ejecución, número de ciclos de reloj, la utilización de la memoria del programa y de

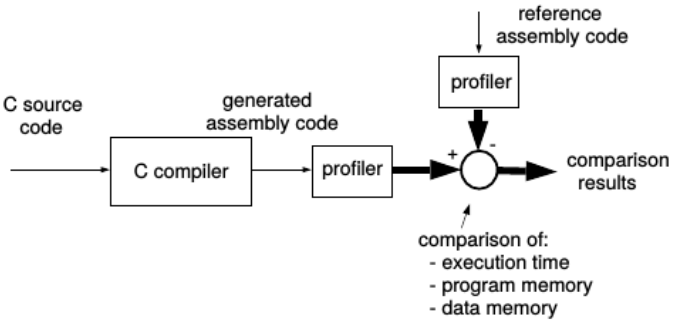

Figura 4.1: Evaluación del rendimiento del Compilador de C. Fuente: [12]

los datos del código generado por el compilador con respecto al código de referencia en ensamblador.

Código fuente de los diferentes *suites* se puede encontrar en [78].

`MediaBench` :

El *benchmark* de aplicaciones `Mediabench` tiene el objetivo especifico de probar el rendimiento en aplicaciones de multimedia, comunicaciones, y DSP (procesamiento de señales digitales) [79].

`MediaBench` fue introducido en un momento donde no existía una forma efectiva de evaluar sistemas dirigidos a sistemas embebidos o sistemas con una componente de DSP, por lo tanto, los objetivos principales de este son: el representar de manera precisa el *workload* de sistemas de multimedia y comunicación, la portabilidad de un set de programas de *benchmarking* y una evaluación que analiza tanto sistemas como la síntesis del mismo.

Las áreas en las que se concentra el `MediaBench` son: video (DCT/IDCT, *motion estimation*, codificación de la entropia), compresión de imagenes (DCT/IDCT, codificación de la entropia), audio (ADPCM), habla (compresión GSM, reconocimiento del habla), seguridad (MD5, IDEA, DES, RSA) y gráficos (renderización, etc). Un exhaustivo perfilado de cada uno de los programas del *benchmark* se pueden encontrar en [80].

`MediaBench` utiliza la ecuación (4.1) para evaluar el rendimiento/costo de un sistema embebido, tomando en cuenta el tamaño de la memoria necesaria para cada programa del *benchmark*. En estas prueba se recomienda la utilización de la media aritmética.

$$\frac{Rendimiento}{Costo} = \frac{\#\text{ de Instrucciones}}{(\text{AreaCore} + \text{AreaCache}) * (\#\text{ de Instrucciones} + \text{CacheMisses*MissPenalty})} \tag{4.1}$$

Finalmente, se encuentra bajo diseño una segunda versión del *benchmark*, `MediaBench II` [81]. El código fuente de los programas y las herramientas utilizadas para el análisis con `MediaBench I` se puede encontrar en [82].

4.1.4 Relevancia

A continuación, se analiza la relevancia científica normalizada a los años con respecto al número de resultados en Google, para obtener el número de menciones del *benchmark* en la Web (hits en la web). Seguidamente, se encuentra el número de resultados en Google Académico (hits académicos), esto con el objetivo de obtener un estimado de la relevancia del *benchmark* bajo análisis en libros, artículos y conferencias. Finalmente, se obtiene la relevancia científica en la web normalizando los hits en la web con los hits académicos con la fecha de publicación del mismo *benchmark*.

Finalmente, a este valor se le aplica el logaritmo natural en base dos, con el fin de eliminar el factor de escala que conllevan las variables que se basan en el algoritmo de búsqueda de Google.

$$\text{Relevancia} = log_2 \frac{\text{hits en la web} \times \text{hits academicos}}{\text{Año}_{Actual} - \text{Año}_{Publicacion}} \tag{4.2}$$

Con esta métrica en cuenta, en la tabla 4.1 se pueden encontrar los resultados de la relevancia científica de cada *benchmark*. Se estresa el hecho de que esta métrica una estimación, pero se puede corroborar esta estimación al observar la relevancia de LinPack, Dhrystone, Whetstone y SPECfp; todos ampliamente utilizados y aceptados en la industria.

Tabla 4.1: Número de resultados sobre buscadores comunes de cada *benchmark*

Benchmark	Hits en la Web	Hits Académicos	Relevancia
SPECfp	125000	106000	8,69
Linpack	87700	8350	7,20
Whetstone	202000	2080	7,02
Dhrystone	116000	2700	6,99
MiBench	26100	3890	6,83
MediaBench	22800	3170	6,58
CoreMark	12000	326	5,81
FBench	17500	63	4,61
DSPStone	3500	93	4,19
ParMiBench	798	71	3,98
FFBench	15100	8	3,65
FPMark	3700	8	3,63
OpenMPBench	425	38	3,43

Tabla 4.2: Rasgos de diferentes *benchmarks* computacionales.

Benchmark	Enfoque Embebido	Kernel	Toy/Sintético	HW Core	Restrictivo	Relevancia
Linpack			✓			7,20
OpenMPBench	✓					3,43
FPMark	✓	✓			✓	3,63
FFBench		✓	✓			3,65
FBench		✓	✓			4,61
Dhrystone			✓			6,99
Whetstone			✓			7,02
DSPStone	✓	✓		✓		4,19
MediaBench	✓	✓				6,58
SPECfp					✓	8,69
CoreMark		✓		✓	✓	5,81
ParMiBench	✓	✓				3,98
MiBench	✓	✓				6,83

4.2 Medición del rendimiento

4.2.1 CPU

Una de las formas más comunes de medir el rendimiento, es el reportar el tiempo total tomado para ejecutar cierto programa, generalmente referido como tiempo de ejecución en el CPU con multiples hilos y programas [2].

Una forma fácil de relacionar el tiempo de ejecución, con el número de ciclos que toma a cierto programa ejecutarse es por medio de las siguientes expresiones:

$$\begin{matrix}\text{Tiempo de ejecución}\\ \text{de un programa}\end{matrix} = \begin{matrix}\text{Ciclos de reloj}\\ \text{de un programa}\end{matrix} \times \begin{matrix}\text{Periodo de ciclo}\\ \text{de reloj}\end{matrix} \tag{4.3}$$

O, alternativamente:

$$\begin{matrix}\text{Tiempo de ejecución}\\ \text{de un programa}\end{matrix} = \frac{\begin{matrix}\text{Ciclos de reloj}\\ \text{de un programa}\end{matrix}}{\begin{matrix}\text{Frecuencia de}\\ \text{reloj}\end{matrix}} \tag{4.4}$$

Una métrica que ayuda a medir el rendimiento de las instrucciones procesadas por el compilador y ejecutadas en un procesador, es el número promedio de instrucciones ejecutadas por ciclo de reloj (CPI, por sus siglas en inglés). CPI provee a los diseñadores de una comparación del rendimiento de dos implementaciones con la misma arquitectura.

$$\text{Total de ciclos de reloj} = \text{Número de instrucciones del programa} \times \text{Promedio de ciclos de reloj por instrucción} \tag{4.5}$$

Por otro lado, el rendimiento también se puede describir tomando en cuenta el número de instrucciones del mismo programa (IC, por sus siglas en inglés).

$$\text{Tiempo de ejecución} = IC \times CPI \times \text{Periodo del ciclo de reloj} \tag{4.6}$$

El reto real de medir el rendimiento de un procesador, un conjunto de instrucciones, o un compilador es la profunda relación que tienen estos entre sí. Para entender esto mejor, se expone la tabla 4.3.

Tabla 4.3: Descripción de las señales del módulo de suma/resta en coma flotante.

Componente	Determinante
Algoritmo	IC, CPI
Lenguaje de programación	IC, CPI
Compilador	IC, CPI
ISA	IC, Frecuencia de reloj, CPI

4.3 Metodología para el rendimiento de una FPU de baja potencia en un ASP

Con los insumos encontrados en el capítulo 3, y con lo expuesto en el presente capítulo, se tiene el conocimiento necesario para exponer una metodología que de manera efectiva puede medir el rendimiento de las caracteristicas importantes del presente diseño de FPU y del ASP utilizado en el SiRPA.

En el contexto de la presente aplicación específica, se debe de tomar en cuenta lo siguiente:

- La aplicación del ASP se concentra en un contexto de bajo gasto energético.
- La aplicación del ASP es de un algoritmo de DSP, con una gran cantidad de operaciones de suma/restas y multiplicaciones.
- El ASP presenta un ISA apegado, a un nivel parcial, al estándar del RISC-V, ya que solamente se implementó la extensión M, F y D [3].
- El programa del HMM implementado en [3] supone la utilización de un programa pequeño.

4.3.1 Selección de *Benchmarks*

El ASP del SiRPA no se puede medir de manera fidedigna con cualquier *benchmark*, debido a que no todo el ISA fue implementado, y la aplicación es en baja potencia, con una aplicación DSP embebida. Por lo tanto, los *benchmarks* indicados son de tipo programa núcleo o *toy benchmark*, debido a que estos se asemejan en términos de tamaño y complejidad al HMM implementado en el ASP.

Adicionalmente, en el presente ASP no se ha implementado todo el estándar del ISA RISC-V, por lo que solamente programas con cierto nivel de complejidad se pueden ejecutar.

Observando la tabla 4.2, se puede observar como `MiBench`, `DSPStone`, `FFBench`, `FBench` y `MediaBench` cumplen con ambas necesidades mientras se tiene un cierto grado de relevancia dentro de la comunidad cientifica (>3).

Tomando en cuenta las similitudes entre el core Milk y Coffee comentados en la sección 3.1 con el ASP utilizado en el SiRPA, se recomienda utilizar los programas utilizados por `DSPStone` para el presente *benchmarking*. Adicionalmente, `MediaBench` cuenta con ciertos programas de DSP que se asemejan a la presente aplicación, por lo que también se recomiendan.

Finalmente, como parte del análisis del rendimiento de una arquitectura de un ASP en [83], se utiliza una serie de benchmarks, entre ellos `DSPStone` y `MediaBench`, por lo cual se recomienda la utilización de una combinación de estos para la presente metodología.

4.3.2 Propuesta de *benchmarking*

Método 1

Con el primer método de benchmarking, se propone realizar el siguiente procedimiento (ver figura 4.2) .

1. Utilizar una FPU de las propuestas en el capitulo 3 como punto de comparación. Se recomienda utilizar `FloPoCo` o `VFLOAT`, tomando ventaja de la variabilidad de palabra de ambas bibliotecas.

2. Crear un *wrapper* en HDL para la interfaz en las FPU, y el ASP. Verificar en RTL en funcionamiento en conjunto de FPU+ASP.

3. Sintetizar ambos diseños en una tecnología comercial.

4. Utilizar un *benchmark* de núcleo para la comparación del tiempo de ejecución de algoritmos como el filtro FIR de dos dimensiones (fir2dim) el filtro IIR (ambos de DSPStone) en conjunto con Whetstone y el mismo algoritmo del SiRPA, todo en simulación post-síntesis.

5. Obtener un estimado de la potencia total utilizada al ejecutar diferentes programas.

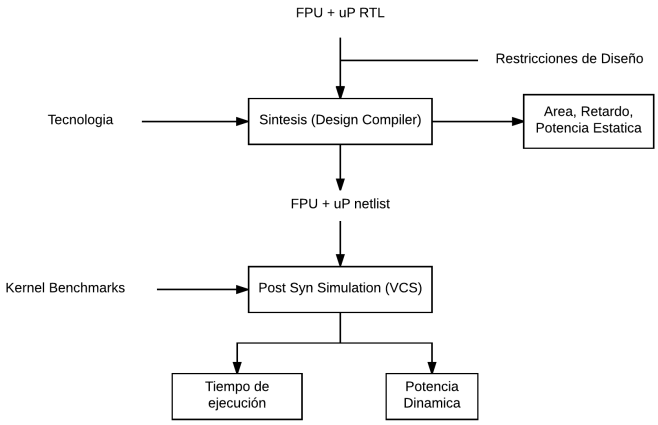

Figura 4.2: Propuesta metodológica para encontrar el rendimiento de un ASP, con énfasis en la potencia. Fuente: Generación propia

Para comparar el rendimiento de ambas implementaciones, se propone usar la *Densidad Computacional* descrita en la ecuación (3.1), basado en la descrita por [79].

$$DensidadComputacional = \frac{f_{clk}}{\#ciclos \times AreaCore} \tag{4.7}$$

Por otro lado, se requiere una medida que ayude a caracterizar el comportamiento de la potencia de un diseño con respecto a otro en la ejecución de un cierto programa. En consecuencia, se presenta el Rendimiento de Potencia como métrica para aliviar esta necesidad. Se notará que la frecuencia a la cual es ejecutado el programa se eleva al cuadrado, esto se hace para normalizar el efecto que tiene la frecuencia sobre la potencia.

$$\text{Rendimiento de Potencia} = \frac{{f_{clk}}^2}{\#ciclos \times \text{ÁreaCore} \times \text{Potencia Dinámica}} \tag{4.8}$$

Método 2

El segundo método propuesto para el análisis del rendimiento de la presente unidad se basa en la ejecución de la FPU sobre un *soft-processor*. Estos son procesadores creados Xilinx específicamente para ser ejecutados sobre FPGAs de Xilinx. Los pasos son los siguientes:

1. Utilizar el Xilinx FPU como punto de referencia. Este ha sido utilizado en varios trabajos como [60] y [9].

2. Adaptar la actual implementación a la interfaz utilizada por el Soft-Processor a ser utilizado. Se recomienda PicoBlaze y la familia Virtex 4.

3. Adaptar el juego de instrucciones del microprocesador en cuestión para trabajar con la unidad de coma flotante.

4. Ejecutar *benchmarks* de núcleo de `DSPStone` y `MediaBench` sobre el procesador en la FPGA.

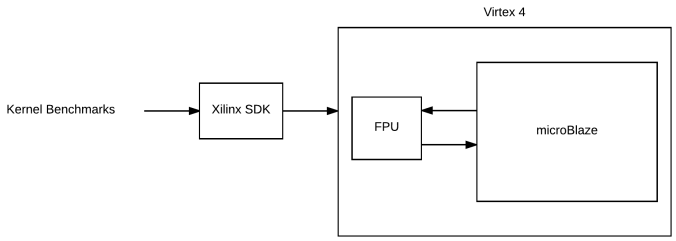

Figura 4.3: Diagrama ejemplificando el Método 2. Fuente: Generación propia

Discusión

El primer procedimiento permite realizar un análisis de aspectos importantes del sistema como área y potencia dinámica, con unidades de puntos flotantes utilizadas y aceptadas en la literatura al ejecutar programas de benchmark de núcleo aceptados en la industria. Las métricas *Densidad Computacional* y *Rendimiento de Potencia* son métricas que ayudan a comparar ambas implementaciones.

Por otro lado, el segundo método se basa de un diseño ya existente y fácilmente maleable por medio de Xilinx SDK. Este método permite obtener un estimado preliminar de la posible utilización de recursos y potencia dinámica de una FPU más un procesador sobre una FPGA. Las métricas de *Densidad Computacional* y *Rendimiento de Potencia* pueden ser utilizadas de una misma forma, modificando la variable de Área por recursos utilizados en FPGA.

Finalmente, esta metodología es propuesta con el fin de ser ejecutada en un futuro trabajo. Esto debido a que en el presente trabajo no se ha podido ejecutar la medición del rendimiento, ya que el trabajo previo a realizar la ejecución del `benchmark` ha sido cuantioso con respecto al resto de objetivos.

Esto significa que el objetivo correspondiente estaba sobredimensionado.

A continuación, se listarán los insumos obtenidos con respecto al *benchmarking*:

- Se hizo un análisis exhaustivo de benchmarks estándar y de software libre con énfasis en sistemas empotrados y de DSP en la literatura. Se realizó un repositorio en Github de estos.
- Se realizó una investigación de las unidades de coma flotantes estándar y de software libre con las cuales se puede realizar una comparación efectiva con la unidad actual. Se tiene un repositorio en Github de estos.
- Se presentaron dos métricas y dos metodologías distintas con las cuales se puede realizar una efectiva medición del rendimiento de la FPU.

Capítulo 5

Análisis de resultados

Como se mencionó en la sección 1.1, se cuenta con las implementaciones en RTL de Rodriguez [17], Quirós [5] y López [13]. Basado en estos diseños, se generan 3 propuestas de mejoras que serán descritas en las siguientes secciones.

Seguidamente, se discuten también los resultados de potencia, área, utilización de recursos y temporizado obtenidos para la familia FPGA Artix 7 en `Vivado 2016.2` y para la síntesis con una tecnología 0.13μm en `Design Compiler` versión L-2016.03-SP3 con el software de interfaz gráfica `Design Vision` para dichas implementaciones basadas en las mejoras propuestas.

La verificación funcional de las mismas mejoras se realizaron utilizando el simulador ISim que acompaña el `Vivado`, en el caso del diseño para FPGA, y `VCS` (utilizando `VDC` como interfaz gráfica) para el caso del diseño en la tecnología 0.13μm.

5.1 Implementación del algoritmo de Karatsuba recursivo

En el presente trabajo, se toma como punto de partida la mejora realizada por López sobre el multiplicador con la implementación del algoritmo simple de Karatsuba (KOA), y se propuso la utilización del algoritmo recursivo de Karatsuba (RKOA) para una FPGA (basado en el trabajo de [23]) con el fundamento expuesto en la subsección 2.2.1.

A continuación, se presentan los resultados correspondientes a la presente implementación de algoritmo recursivo de Karatsuba.

5.1.1 Descripción

Primeramente, se comenzará por realizar una descripción en alto nivel de la unidad (ver Figura 5.1 para detalle de entradas y salidas). Se procura la mantener una serie de puertos

similares a los de anteriores implementaciones, para mantener la modularidad de la implementación.

Figura 5.1: Diagrama de entradas y salidas del multiplicador recursivo de Karatsuba. Fuente: Generación propia

Como se ha comentado en la sección 2.2.1, el algoritmo de Karatsuba-Offman se presta para una implementación recursiva, y los trabajos de [84, 85, 13] muestran mejoras en tamaño y latencia para una implementación recursiva.

En la figura 5.3 se puede observar la representación gráfica de la implementación recursiva del algoritmo de Karatsuba recursivo, visto desde el tamaño de palabra del operador. En la literatura se pueden encontrar diferentes formas de implementar la recursividad del algoritmo, entre ellas, la recursión binaria, de relleno y la general. En esta implementación se utiliza la recursión simple.

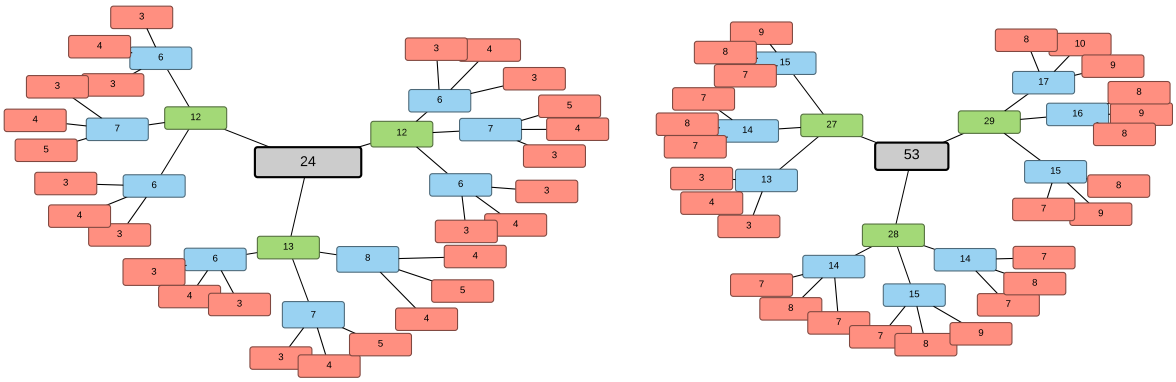

Figura 5.3: Representación gráfica de los bloques de multiplicación de Karatsuba-Offman recursivos, y su tamaño de palabra correspondientes en precisión simple y doble. Fuente: Generación propia.

Finalmente, en el presente trabajo se segmentó con registros la implementación recursiva y simple del multiplicador de Karatsuba para obtener un multiplicador de una (KOA1, RKOA1) y dos etapas (KOA2, RKOA2).

Como punto de comparación, también se implementa el multiplicador disponible en la biblioteca `DesignWare` de `Synopsys`. Este multiplicador es elegido por el mismo sintetizador para obtener la implementación más optima, tomando en cuenta área, tamaño de palabra, retardo, etc [86].

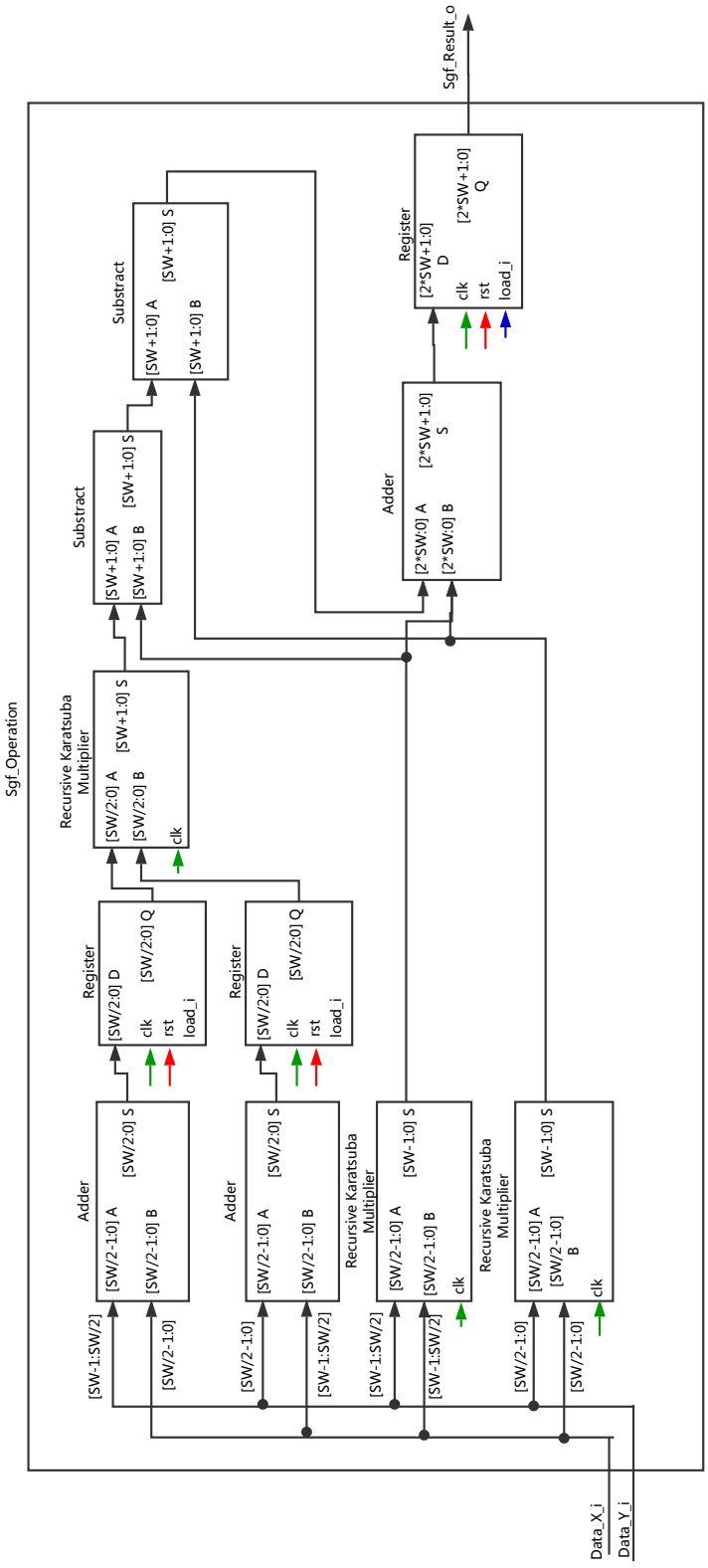

Figura 5.2: Diagrama de bloques segmentado RTL del algoritmo recursivo de Karatsuba implementado. Fuente: Modificado de [13]

5.2 Unidad de sumado en coma flotante segmentado

En esta sección se discutirá sobre los aspectos importantes a la re-implementación del algoritmo de suma/resta en coma flotante, según lo realizado por López en [13].

Los fundamentos importantes que son pertinentes a esta sección se pueden encontrar en la subsección 2.2.2. Se debe siempre tener en cuenta, que en esta subsección se realizó una descripción de las ventajas y desventajas de pasar de una arquitectura a la otra, siempre haciendo analogía entre un procesador (con la operación de la instrucciones) y los operandos de aritmética en coma flotante (con la ejecución del algoritmo de la operación correspondiente, en este caso la suma/resta).

5.2.1 Descripción general

El acercamiento realizado por López comprendía el procedimiento algorítmico comentado en la subsección 2.1.4. Esta implementación hace uso de una arquitectura multiciclo, como la que se menciona en la subsección 2.2.2.

En otras palabras, la implementación de López recibe un par de operandos, y después de 9 - 18 ciclos de reloj, se recibe un resultado (ver Figura 5.10). En contraste, la presente implementación puede procesar tres operandos en 13 ciclos de reloj debido a la arquitectura segmentada que se implementó.

Seguidamente, en la Figura 5.4 se puede observar un diagrama general de las entradas y salidas del módulo para las operaciones de suma/resta en coma flotante. En este se puede observar como, en contraste con la unidad de López ([13]), se elimina la señal de acuse (ack_FSM), el bus de entrada para el modo de redondeo (RMODE) y se agrega la señal para la sincronización de los datos secuenciales (busy). En la tabla 5.1 se detallan las señales del módulo.

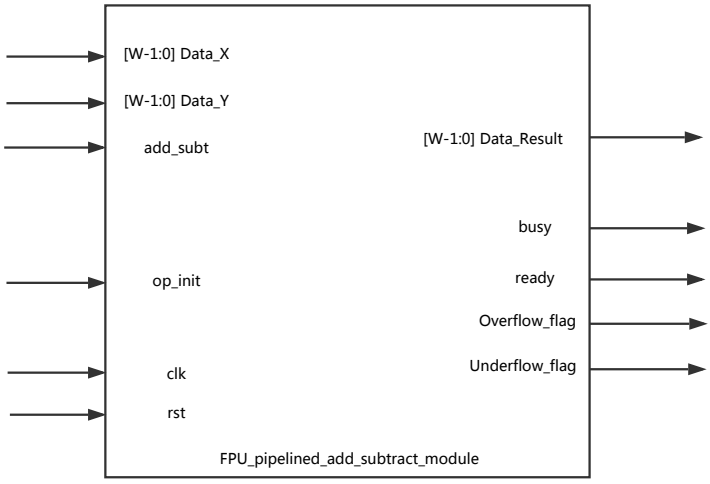

Figura 5.4: Diagrama de entradas y salidas del sumador/restador en coma flotante segmentado. Fuente: Generación propia.

Tabla 5.1: Descripción de las señales del módulo de suma/resta en coma flotante.

Nombre	Tipo	Ancho de Palabra	Descripción
clk	Entrada	1	Señal global de reloj
rst	Entrada	1	Señal global de reset
init_OP	Entrada	1	Señal para el inicio de la operación suma/resta
Data_X, Data_Y	Entrada	W	Operandos en formato IEEE 754
add_subt	Entrada	1	Señal que especifica la operación (1: SUMA; 0: RESTA).
busy	Salida	1	Señal que indica la disponibilidad de la unidad de ingresar operandos para ser operados.
Overflow_flag	Salida	1	La bandera de sobre-desborde (overflow).
underflow_flag	Salida	1	La bandera de sub-desborde (underflow).
zero_flag	Salida	1	Bandera que indica un resultado de cero.
ready	Salida	1	Bandera que indica la disponibilidad del resultado.
Data_Result	Salida	W	Resultado en formato IEEE 754.

En la figura 5.5 se pueden observar los módulos internos que comprenden la unidad realizada. A continuación, se listan las secciones segmentadas, con una descripción la sección combinacional que las precede:

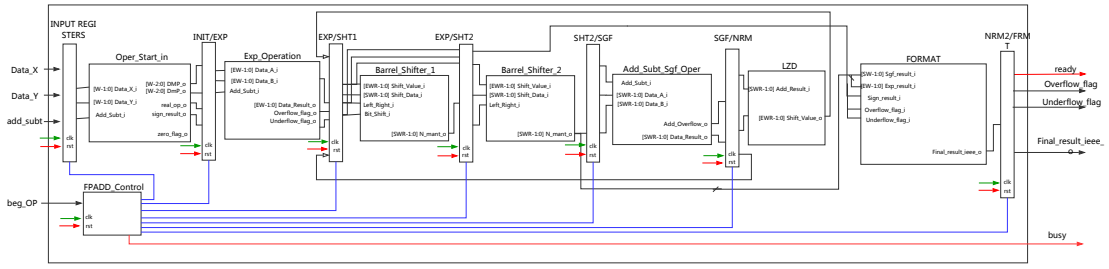

Figura 5.5: Diagrama general del sumador/restador en coma flotante segmentado. Fuente: Generación propia.

1. **INIT/EXP**: aquí se determina el operando de mayor y menor valor, y encontrase necesario, se intercambian de posición. Es conocido en la literatura como etapa *SWAP*.

2. **EXP/SHT1**: realiza la operación de resta entre los operandos para determinar el número corrimientos que se deben de realizar sobre el operando menor.

3. **SHT1/SHT2**: contiene la primera mitad del Desplazador de Barril, una etapa de rotación y un par de arreglos de multiplexores que realizan los corrimientos logarítmicos.

4. **SHT2/SGF**: es la segunda mitad del Desplazador de Barril, tres arreglos de multiplexores que realizan los corrimientos logarítmicos y la rotación final.

5. **SGF/NRM**: la presente sección contiene la ALU que realiza la operación sobre el signficando. Además, el resultado de esta operación puede tener como resultado un sobredesborde (ADD_OVRFLW), y por lo tanto, se trasmite también este valor a la siguiente etapa.

6. **NRM/NRM2**: aquí se efectúa la normalización sobre el valor del resultado entre las mantisas. Para realizar esto, se hace utilización del módulo LZD (Leading Zero Detector, o Anticipador de Ceros) y la primera sección del módulo Desplazador de Barril.

7. **NRM/NRM2**: aquí se utiliza la segunda sección del Desplazador de Barril, y la unidad de formato, que tiene como objetivo el formateo final del signo, el exponente y la mantisa en el formato IEEE 754.

El procedimiento que se ha listado anteriormente se puede analizar directamente con el algoritmo 2.1, excepto por el procedimiento de redondeo, el cual se ha determinado no necesita ser implementado para el proyecto SiRPA, tomando en cuenta el área y potencia extra consumida versus la ganancia en precisión y rango del mismo redondeo.

5.2.2 Unidad de control del operador de suma/resta en coma flotante

En la figura 5.6 se pueden observar los módulos para el manejo del flujo de datos de la unidad de suma/resta. Esta unidad se encarga del ingreso de los datos para su respectiva operación, y es de vital importancia, ya que evita la corrupción de datos en cada etapa.

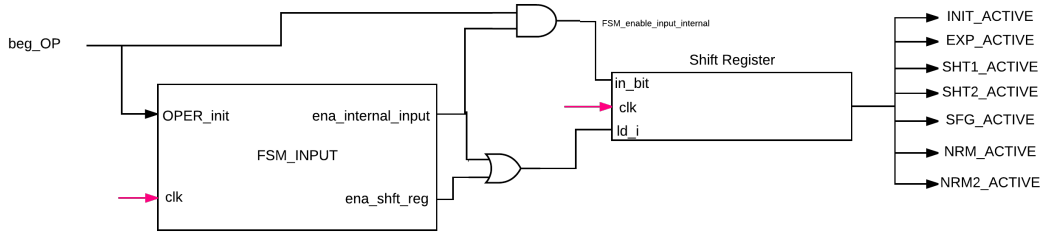

Figura 5.6: Unidades para el control de la suma/resta en coma flotante segmentada. Fuente: Generación propia.

Primeramente, para entender efectivamente el flujo de los datos, se debe analizar el diagrama de tiempos que se muestra en la figura 5.10. Se denota en la parte superior la etapa de segmentación, y se puede observar a la izquierda de la figura el ciclo de reloj correspondiente.

En este diagrama, se observa el flujo que llevan los operandos en el tiempo, y es de vital importancia observar en el tiempo $t = 6T$ los operandos de $Data1$, ya que estos hacen reutilización de la primera y la segunda sección del Desplazador de Barril. Esto se debe a que

la mantisa debe ser re-normalizada. La re-utilización de esta etapa significa el comprometer el rendimiento de procesamiento de la misma unidad, pero se ahorra en área (y la potencia estática correspondiente) al evitar utilizar un segundo desplazador.

Se puede observar en la figura 5.9 la implementación de la FSM (Maquina de Estados) realizada por López en la versión anterior del sumado, y la presente. Entre los cambios más significativos, se puede mencionar la utilización de un registro de corrimiento externo a la FSM para el control de las etapas segmentadas, y a la eliminación del bucle utilizado para el redondeo (estados 9 - 7 - 10 - 11 - 12).

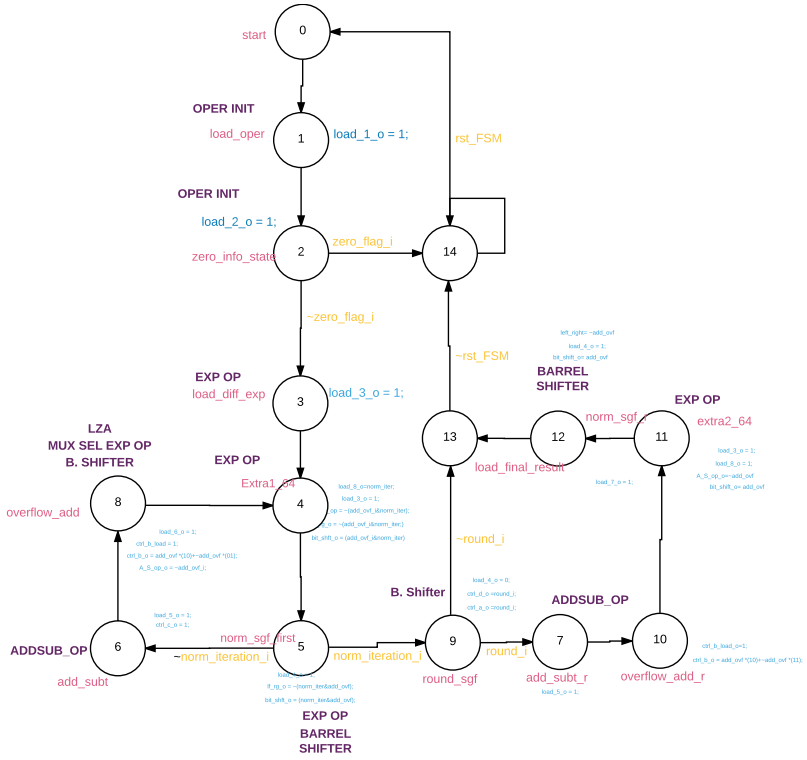

Figura 5.7: Implementación de López [13].

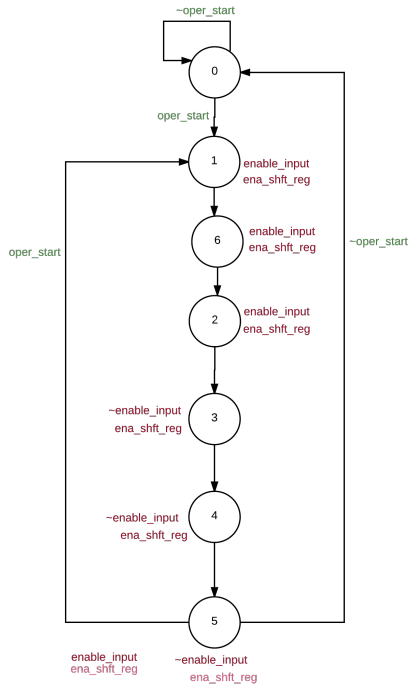

Figura 5.8: Presente implementación

Figura 5.9: Diagramas de las máquinas de estados para las versiones de sumador

La utilización del presente esquema permite obtener una FSM simple y limpia, y un registro de corrimiento externo permite evitar *glitches* o estados inesperados en el control de la ruta de datos. Además, la presente implementación permite el agregar una etapa final de redondeo sin cambios significativos sobre la arquitectura.

Tabla 5.2: Señales de entrada y salida para la FSM de la suma/resta en coma flotante

Nombre	Tipo	Descripción	Dirección
clk	Entrada	Señal global de reloj	Proviene del reloj del sistema externo
rst	Entrada	Señal global de *reset*	Proviene del *reset* del sistema externo
init_OP	Entrada	Señal de inicio de la operación suma/resta	Proviene del sistema externo
ena_internal_i	Salida	Señal a ser ingresada al bit de entrada del registro de corrimiento de control	Registro de corrimiento de control
ena_Pipe_i	Salida	Señal de habilitación de la primera etapa de registros de la unidad de suma/resta.	Registros de la Etapa INIT (INPUT_STAGE_OPERANDX, INPUT_STAGE_OPERANDY, INPUT_STAGE_FLAGS)
ena_shift_reg	Salida	Señal de habilitación del registro de corrimiento	Registro de corrimiento de control

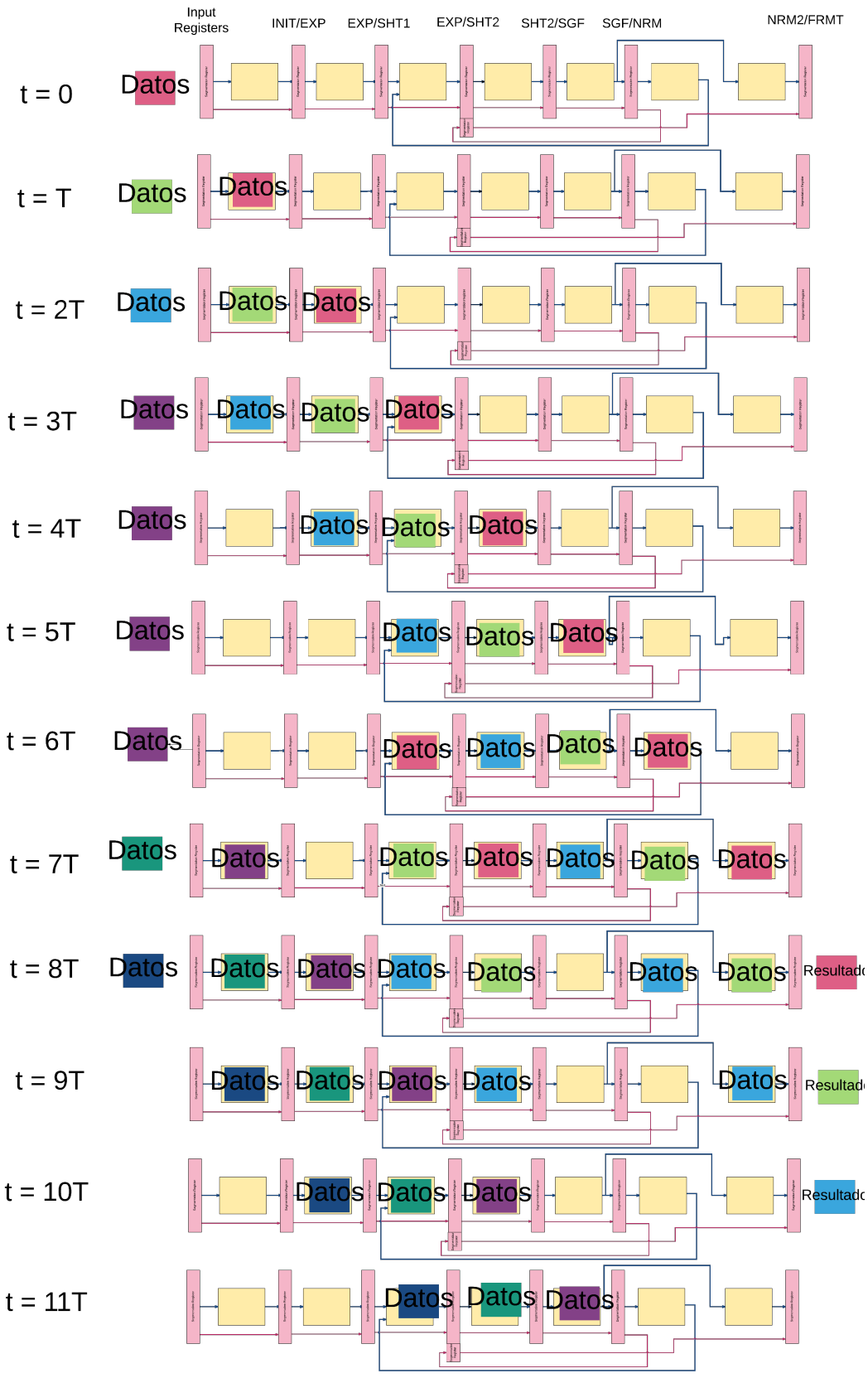

Figura 5.10: Diagrama de tiempos con respecto a las etapas segmentadas de la unidad de suma/resta en coma flotante. Fuente: Generación propia.

5.3 CORDIC con unidad de suma segmentada

Como se comentó en la sección 3.6.2, la unidad de CORDIC fue creada de tal forma que se le puede agregar una unidad de suma de coma flotante. Por lo tanto, se decidió utilizar esta unidad para probar la suma presentada en la sección anterior y al mismo tiempo mejorar el rendimiento de la misma.

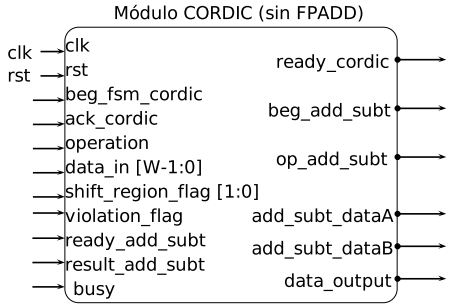

Figura 5.11: Interfaz de la unidad CORDIC para unidad de suma externa.

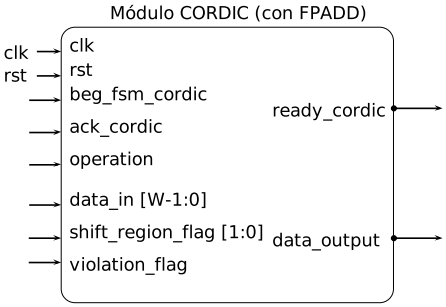

Figura 5.12: Interfaz de la unidad CORDIC con sumado integrado

Figura 5.13: Interfaz de la unidad CORDIC. Basado en [5]

En la figura 5.13 se puede encontrar la interfaz de la implementación de CORDIC con la suma en coma flotante integrada dentro de la unidad, y la interfaz con los puertos listos para interactuar con una unidad de sumado flotante externa.

5.3.1 Modificación de la unidad de control

Se debe de modificar la versión del operador CORDIC entregado por Quirós, y realizar una integración de la nueva unidad de sumado. Primeramente, se debe analizar la lógica combinacional que se encuentra dentro de la máquina de estados.

Esto es analizado y sacado fuera de la Unidad de Control e introducido a la ruta de datos como el módulo region_sel.

Seguidamente, también se elimina la utilización de algunos registros que complementaban el trabajo de la unidad de control dentro de la ruta de datos. Estas modificaciones se pueden ver en la figura 5.14

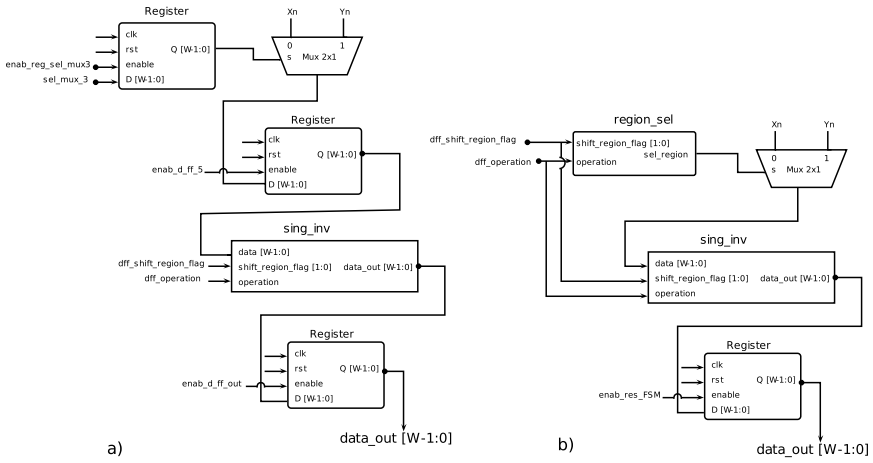

Figura 5.14: Final de la ruta de datos para implementación de CORDIC. a) Implementación de Quirós [5]. b) Presente solución.

Seguidamente, se procede a eliminar la lógica de espera del resultado de la suma de la variable temporal (X_n, Y_n o Z_n). Se debe de recordar que la versión de la suma de López utiliza una arquitectura la cual tiene que esperar de 9 a 13 ciclos de reloj para dar cada resultado.

Por otro lado, con la presente versión de la suma, al recibir un resulto de las tres operaciones en fila, se debe adecuar la arquitectura de Quirós.

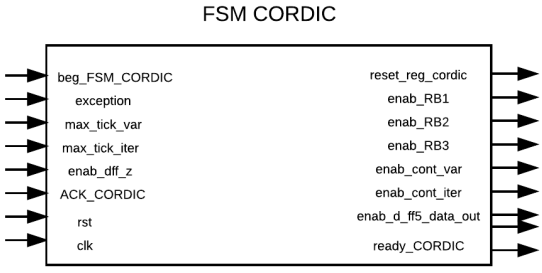

Figura 5.15: Interfaz de la unidad de control utilizada en la presente solución de CORDIC

5.3.2 Manejo del sumador

Para ingresar los valores X, Y y Z del algoritmo, y recibir los resultados de los mismos de la unidad de sumado, se hace utilización de un contador digital de 2 bits y un decodificador de prioridad.

El contador ayuda a ingresar de manera secuencial ingresar los operandos a ser controlados. El decodificador de proridad se encarga de seleccionar la señal de habilitación del los registros donde serán guardados los resultados de la operación, y la máquina de estados se encarga de manejar ambas unidades. La figura 5.16 muestra un diagrama de bloques que ejemplifica lo discutido.

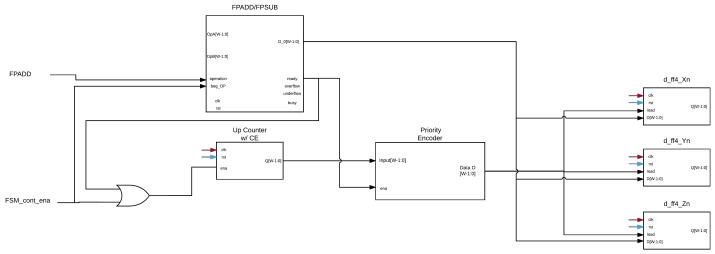

Figura 5.16: Unidad de manejo de la suma/resta en la unidad CORDIC en la presente implementación

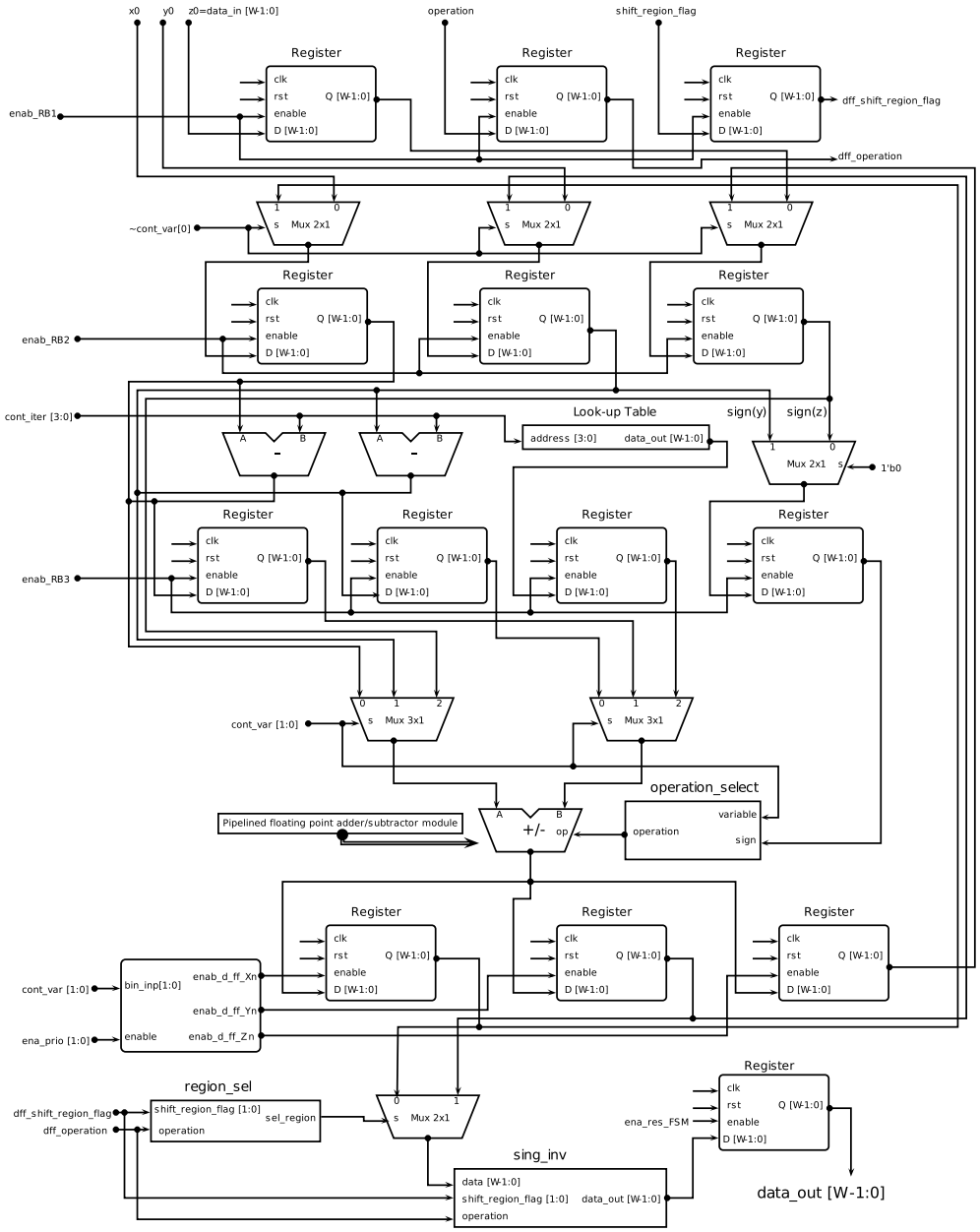

Figura 5.17: Diagrama completo del algoritmo de CORDIC con arquitectura bit-paralela iterativa utilizando suma segmentada

5.4 Resultados

5.4.1 Descripción

A continuación, se muestran los resultados de sintesis sobre un FPGA de una familia Artix 7 y sobre la tecnología comercial de 0.13μm. Los resultados corresponden a las diferentes implementaciones de multiplicadores y las unidades de coma flotante de los multiplicadores correspondientes, el sumador de coma flotante anterior y el actual, las diferentes versiones de CORDIC sin el sumador de coma flotante, y la presente FPU. Adicionalmente, todas estas unidades han sido verificadas con pruebas de banco utilizando vectores de punto de referencia dorada.

Siguiendo la metodologia de López y Quirós para medir el rendimiento de los operadores de coma flotante implementados, se ha medido el tiempo de ejecución al introducir 1024 operandos sobre las unidades de multiplicación, suma/resta, y el CORDIC. Los tiempos de ejecución con respecto a anteriores implementaciones se pueden observar en las tablas 5.8 y 5.6.

Por otro lado, con el fin de comprar en términos de retardo las diferentes implementaciones, se encontraron las diferentes frecuencias máximas a las que pueden operar los operadores, estas son presentadas en las tablas 5.4, 5.7 y 5.11.

Tabla 5.3: Utilización de recursos en FPGA para multiplicadores de Karatsuba simple y recursivo de 24 bits

	KOA1	KOA2	RKOA1	RKOA2	Δ	Δ
LUT	622	612	678	667	+56(%7)	+55(%7.2)
FF	46	118	46	162	0	+44(%37)

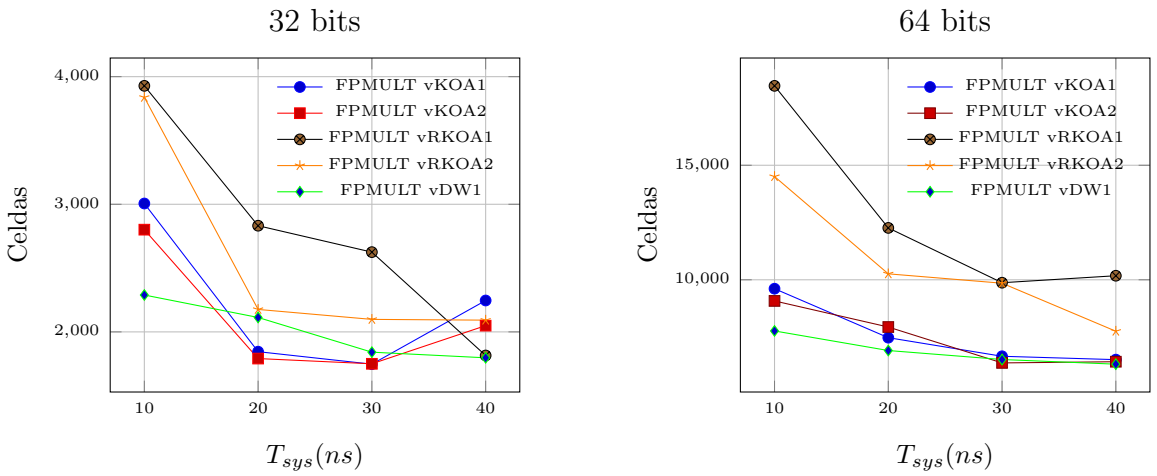

Figura 5.18: Número total de celdas utilizadas en función del periodo de reloj del sistema para diferentes tipos de multiplicadores en precisión simple y doble.

Tabla 5.4: Frecuencia máxima (en MHz) que pueden alcanzar diferentes multiplicadores en una tecnología de 0.13μm

Multiplicador	32 bits	64 bits
DW	255.75 (+%26.23)	199.60 (-%1.18)
KOA1	202.02	158.48 (-%21.78)
KOA2	243.31 (+%20.29)	202.03
RKOA1	160.51 (-%20.79)	109.05 (-%20.79)
RKOA2	225.73 (+%11.74)	120.77 (-%46.03)

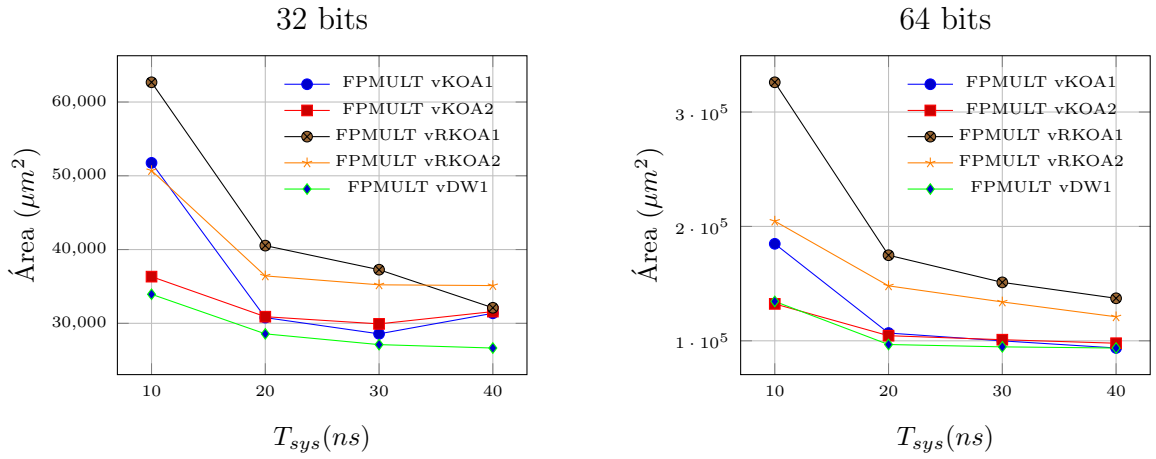

Figura 5.19: Área del diseño en función del periodo de reloj del sistema para diferentes tipos de multiplicadores en precisión simple y doble.

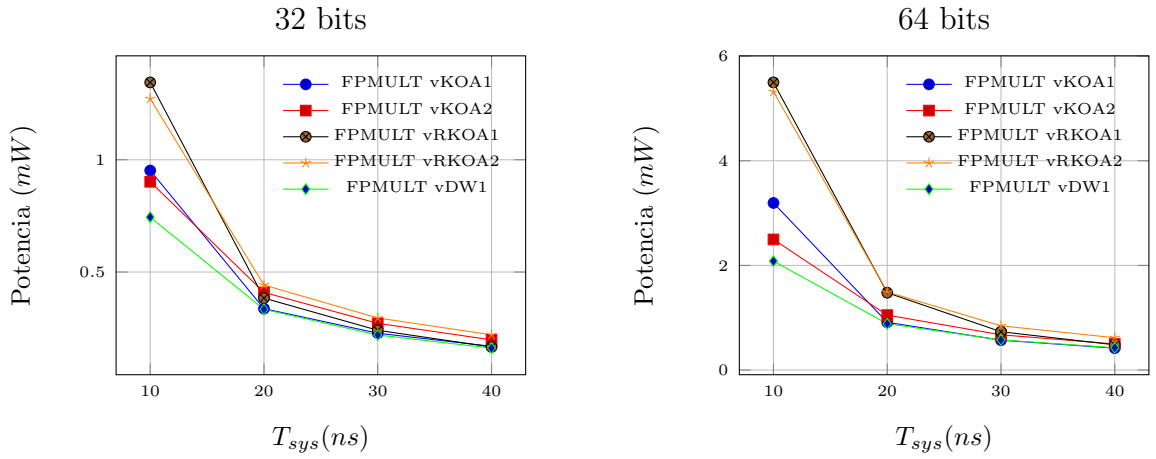

Figura 5.20: Estimado de potencia total en función del periodo de reloj del sistema para diferentes tipos de multiplicadores en precisión simple y doble.

Tabla 5.5: Comparación de los recursos lógicos utilizados en la FPGA entre las versiones de la FPU en la operación suma/resta en precisión de 32 y 64 bits

Diseño	Recursos lógico	32 bits	64 bits
Primera versión	LUT	326	571
	FF	467	912
Diseño con LZA	LUT	433	924
	FF	206	573
Diseño con LZD	LUT	310	682
	FF	263	514
Diseño Actual	LUT	303	638
	FF	413	792
Δ	LUT	-7(2.5%)	-44(6.4%)
	FF	+150(57%)	+278(54%)

Tabla 5.6: Tiempo de ejecución de la operación suma/resta para 1024 valores en precisión de 32 y 64 bits

Arquitectura	32 bits	64 bits
Primera versión	463,97	455,69
Segunda versión	174,02	174,02
Versión diseñada	20,54	20,54
Δ	153,48	153,48
Speedup	8,47	8,47

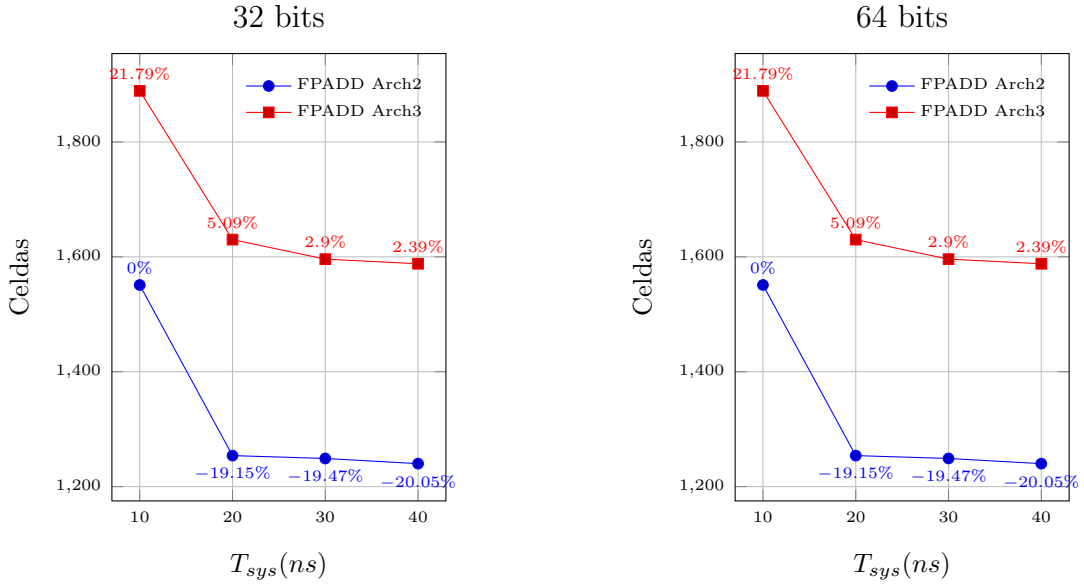

Figura 5.21: Número total de celdas utilizadas en función del periodo de reloj del sistema para la versión de López y la actual en precisión simple y doble.

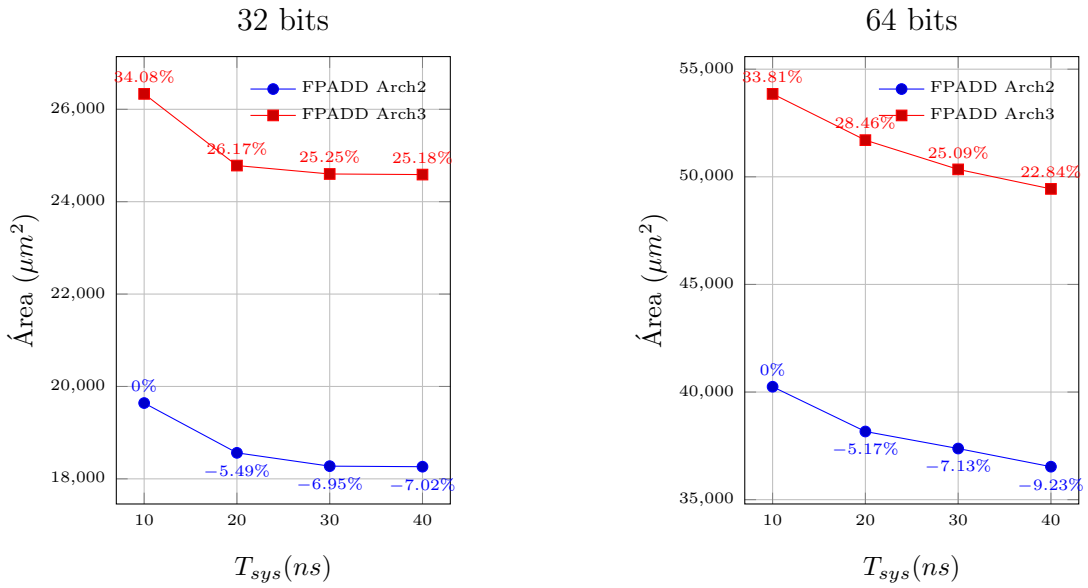

Figura 5.22: Área del diseño en función del periodo de reloj del sistema para la versión de López y la actual en precisión simple y doble.

z

Tabla 5.7: Frecuencia máxima (en MHz) que pueden alcanzar ambas implementanciones en una tecnología de 0.13μm

	32 bits	64 bits
Segunda Versión	341	303
Presente Versión	350 (+%2.36)	318 (+%4.95)

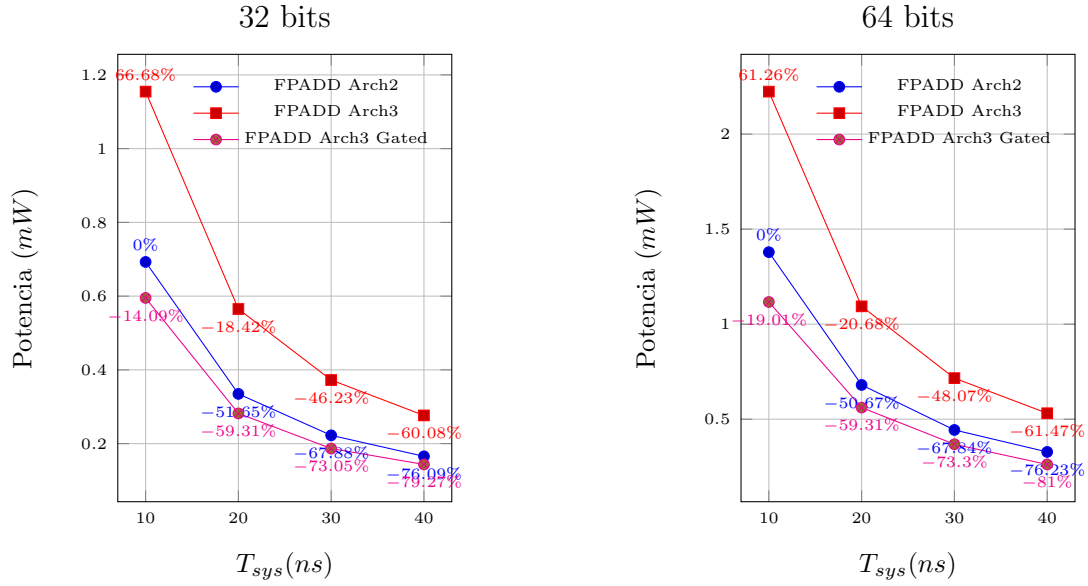

Figura 5.23: Estimado de potencia total en función del periodo de reloj del sistema para la versión de López y la actual con y sin *clock gating* en precisión simple y doble.

Tabla 5.8: Rendimiento al procesar 1024 operadores con las diferentes versiones del CORDIC en precisión simple y doble

Arquitectura	32 bits	64 bits
Primera versión	7754 ns	7754 ns
Versión diseñada	2639 ns	2639 ns
Speedup	2.93	2.93

Tabla 5.9: Comparación en la utilización de recursos sobre la unidad CORDIC en FPGA para precisión simple y doble

Diseño	Recursos lógico	32 bits	64 bits
Primera versión	LUT	192	347
	FF	391	771
Diseño Actual	LUT	229	433
	FF	359	709
Δ	LUT	+37(%16.16)	+86(%19.86)
	FF	-32(%8.91)	-62(%8.74)

Tabla 5.10: Comparación en la utilización de potencia (en mW)sobre la unidad CORDIC en FPGA para precisión simple y doble

Secciones	Primera Versión		Diseño Actual	
On-chip	32 bits	64 bits	32 bits	64 bits
Clock	2	4	4	6
Logic	3	5	4	7
Signals	2	5	4	10
Static	97	97	97	97
Total	104	111	109	120

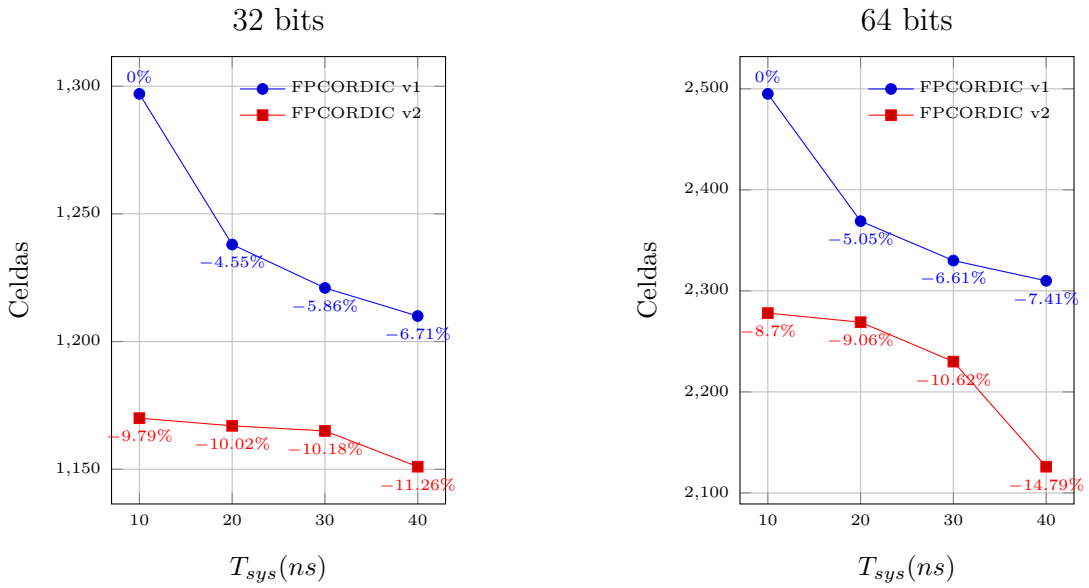

Figura 5.24: Número total de celdas utilizadas en función del periodo de reloj del sistema para la versión de Quirós y la actual en precisión simple y doble.

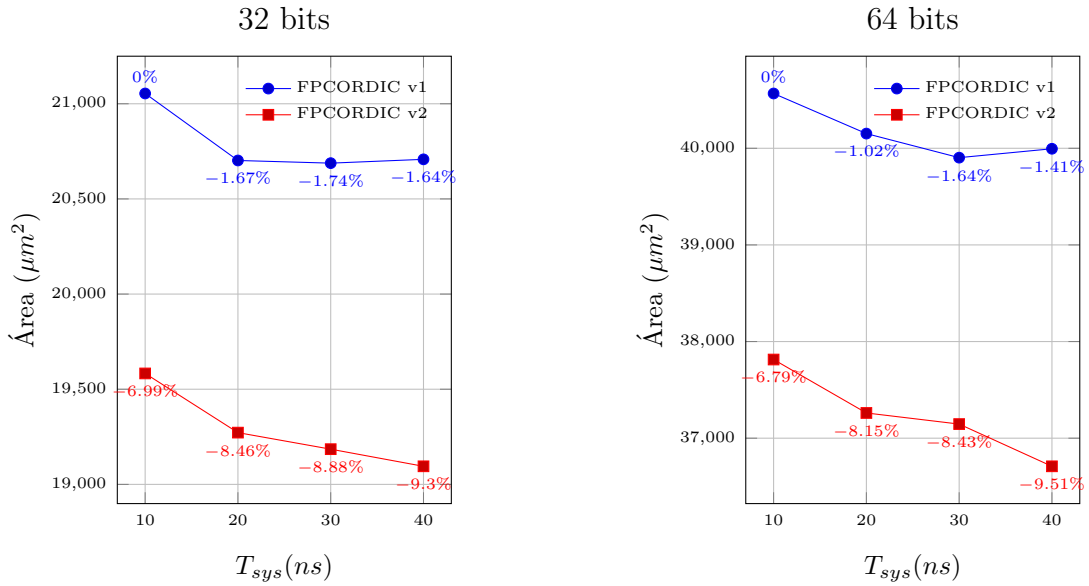

Figura 5.25: Área total del diseño en función del periodo de reloj del sistema para la versión de Quirós y la actual del CORDIC en precisión simple y doble.

Tabla 5.11: Frecuencia máxima (en MHz) que pueden alcanzar ambas implementaciones del CORDIC en una tecnología de 0.13μm

	32 bits	64 bits
Segunda Versión	487	426
Presente Versión	537 (+%10.26)	446 (+%4.69)

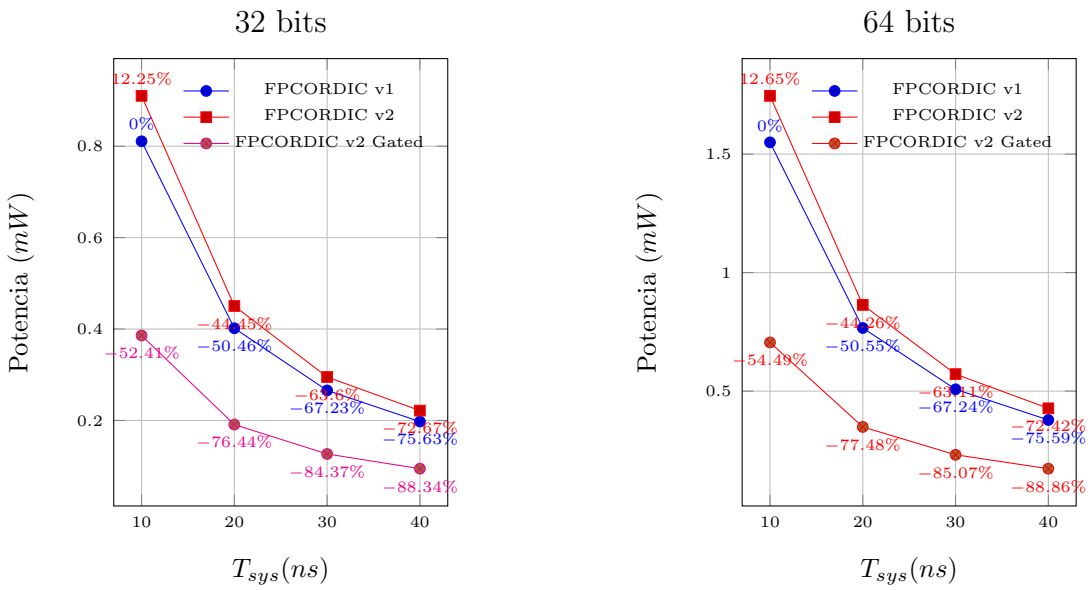

Figura 5.26: Estimado de potencia total en función del periodo de reloj del sistema para el CORDIC de Quirós y la presente solución con y sin *clock gating* en precisión simple y doble.

Tabla 5.12: Utilización de recursos sobre FPGA para la unidad de coma flotante.s

Diseño	Recursos lógico	32 bits	64 bits
Segunda versión	LUT	790	1819
	FF	933	1945
	DSP	4	12
Diseño Actual	LUT	809	1884
	FF	1045	2154
	DSP	4	12
Δ	LUT	19(+1.04%)	65(+3.45%)
	FF	112(+5.76%)	209(+9.70%)
	DSP	0	0

Tabla 5.13: Consumo de potencia para la unidad de coma flotante.

Secciones	Primera Versión		Solución	
On-chip	32 bits	64 bits	32 bits	64 bits
Clock	7	14	13	16
Signals	6	17	7	18
Logic	4	11	5	13
DSP	3	7	3	10
Static	104	104	104	105
Total	124	153	132	162

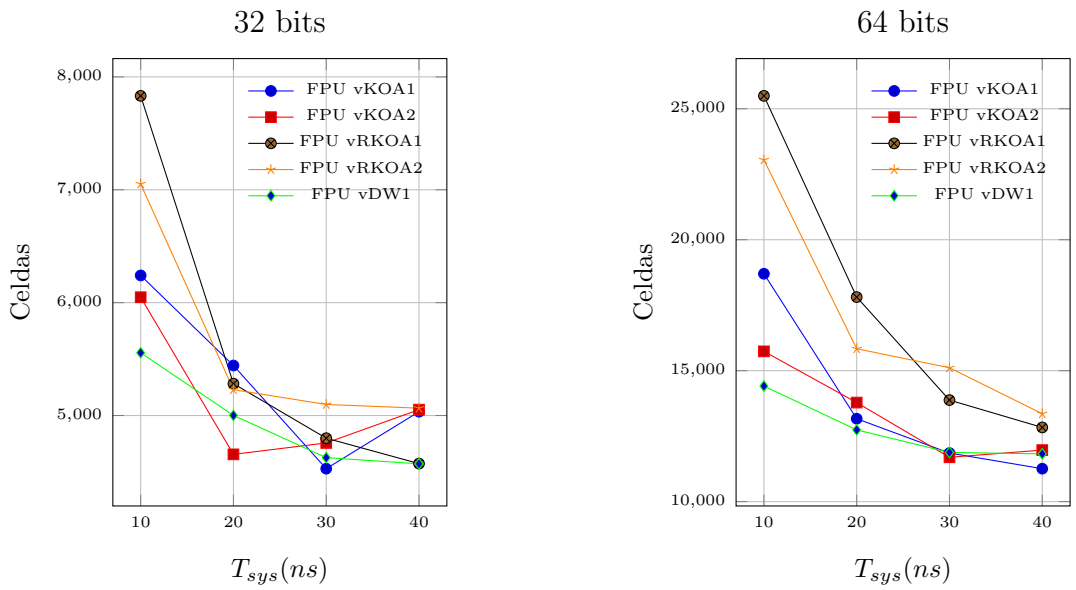

Figura 5.27: Número total de celdas utilizadas en función del periodo de reloj del sistema de la solución actual de FPU en precisión simple y doble.

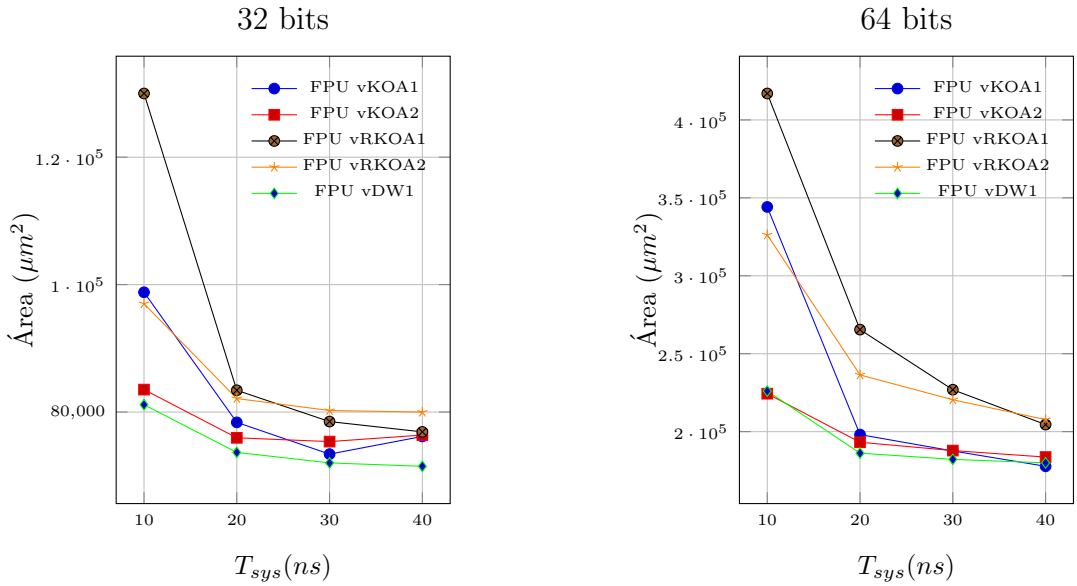

Figura 5.28: Área total del diseño en función del periodo de reloj del sistema de la solución actual de FPU en precisión simple y doble.

Tabla 5.14: Resumen sobre las mejoras implementadas en términos de potencia consumida (mW)

Multiplicador de Karatsuba		Suma en coma flotante		Algoritmo de CORDIC	
Versión anterior	Mejora 1	Versión anterior	Mejora 2	Versión anterior	Mejora 3
0.9522	0.9020	0.6927	0.5957	0.8108	0.4064

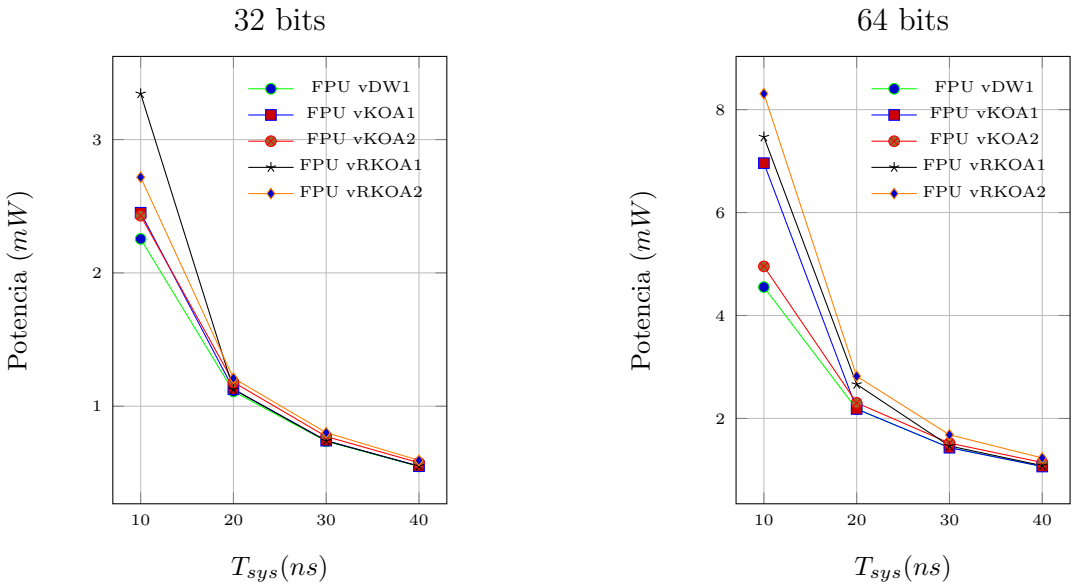

Figura 5.29: Estimado de potencia en función del periodo de reloj del sistema de la solución actual de FPU en precisión simple y doble.

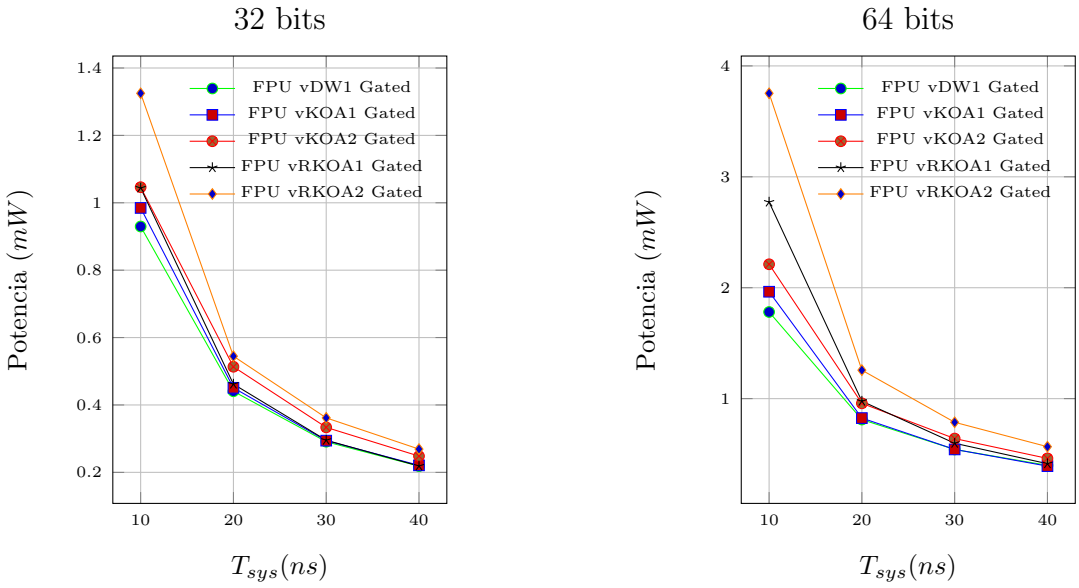

Figura 5.30: Estimado de potencia total en función del periodo de reloj del sistema de la solución actual de FPU en precisión simple y doble utilizando *clock gating*

5.5 Análisis de los resultados

Primeramente, se puede observar en las Figuras 5.18, 5.19, 5.20, 5.21, 5.22, 5.23, 5.25, 5.26, 5.27, 5.28, 5.30 el decrecimiento en término de área, utilización de celdas, y potencia, al disminuir la restricción del reloj al que se ejecuta el sistema. En el caso de la potencia, la potencia dinámica constituye casi más de un 70% de la potencia total consumida, y debido a que la potencia dinámica es directamente proporcional al cuadrado de la frecuencia del sistema ([18]), se justifica así el decrecimiento de la potencia.

Por otro lado, al relajar la restricción temporal del sistema se pueden utilizar celdas lógicas con menos esfuerzo lógico (ya que no se necesita más corriente para que ocurra la conmutación a la salida del transistor), que a su vez tienen una menor área, lo que explica el decrecimiento en el área de los diseños a menos frecuencias de operación [18].

Se encuentra que la implementación del algoritmo simple de Karatsuba tiene una mejor utilización de las celdas DSP48, al utilizar 4 DSPs. En este caso, se ha obligado al sintetizador de Vivado el no utilizar estas celdas, con el fin de realizar una comparación efectiva entre ambas unidades. Se nota en la tabla 5.3, cómo la implementación de Karatsuba simple de una etapa (KOA1, la de López) es la más efectiva en términos de utilización de recursos sobre la FPGA. Esto se debe a que la implementación del multiplicador de Karatsuba recursivo es más eficiente a mayor largo de palabra.

Se puede notar de la Figura 5.20 cómo la implementación de multiplicador de la biblioteca de *cores* de `DesignWare` es la más efectiva en términos de potencia para ambas precisiones. La implementación de Karatsuba simple de dos etapas es la segunda más efectiva en términos de potencia.

En términos de retardo (ver tabla 5.4), la implementación del Karatsuba recursivo de dos etapas en precisión simple es un 11% más veloz que la implementación de López. Por otro lado, la implementación de `DesignWare` se muestra la más veloz, al alcanzar un 26% más de frecuencia máxima, con respecto a la implementación de López.

Seguidamente, los resultados obtenidos de la unidad de sumado actual en FPGA (ver tabla 5.5) muestran un aumento en registros con respecto a la implementación de López, esto gracias a la transferencia de estados inherente a una implementación segmentada. El aumento es de un 54% a un 57%. Por otro lado, se encuentra una reducción en la utilización de LUTs de un 2.5% para la precisión simple y de un 6.4% para la precisión simple. Esto se puede adjudicar a la simplificación en el control de las diferentes etapas del operador, y la eliminación de la unidad de redondeo.

La síntesis sobre la tecnología 0.13μm muestra un aumento en potencia, área y celdas de la presente implementación con respecto a la implementación de López para una frecuencia de 100MHz (ver Figuras 5.21, 5.22, 5.23). Esto es de esperar, debido al aumento en las unidades secuenciales.

No obstante, la presente solución presenta un *speedup* de 8.5 con respecto a la implementación de López (ver la tabla 5.6). Este aumento en rendimiento de la presente unidad permite

el disminuir la frecuencia de reloj del sistema, sin afectar la ejecución del programa del ASP. Permitiendo así, a una velocidad de 25MHz y en precisión simple, una disminución del área consumida de un 6%, una disminución de la utilización de celdas de un 16% y una disminución del 76% en la potencia total estimada con respecto a la velocidad de 100MHz.

Adicionalmente, la disminución de la potencia de la presente solución, en precisión simple, a una frecuencia de 25MHz ($0.27mW$) (ver Figura 5.23) con respecto a la implementación de López, a una velocidad de 100MHz ($0.693mW$), es de un 66% y al realizar *clock gating* sobre el presente diseño, esto se vuelve una disminución del 75%.

Adicionalmente, la presente unidad de sumado es más rápida que la anterior, teniendo una velocidad máxima de operación de 350MHz y 318MHz para una precisión simple y doble, respectivamente (ver tabla 5.7).

Con respecto a la unidad de CORDIC, se puede notar en la tabla 5.9 como la implementación de FPGA presenta un ligero aumento de la utilización de LUTs (16% en precisión simple y 19.86% en precisión doble). Esto se puede adjudicar a decodificador de prioridad que ayuda a controlar la unidad de sumado en coma flotante. Asimismo, se encuentra una disminución de las unidades secuenciales (9% en precisión simple y 8 en precisión doble). Esta disminución de las unidades secuenciales se puede atribuir a la simplificación sobre la FSM y las unidades secuenciales que se eliminaron de la ruta de los datos.

Por otro lado, la solución actual muestra una mejora en la utilización del área y de las celdas (ver Figuras 5.24 y 5.25)de un 9.8% y 8.7% en la tecnología 0.13μm, respectivamente, con respecto a la solución anterior. Estas mejoras son atribuibles a las simplificaciones sobre la arquitectura del CORDIC.

En contraste, a una frecuencia de 100MHz, la implementación actual tiene un consumo de potencia ($0.9101mW$) de un 12.34% más que la solución anterior ($0.8108mW$) (ver Figura 5.26). No obstante, el rendimiento de la presente implementación muestra un *speedup* de 3 al procesar 1024 ángulos, en comparación con la solución anterior (ver tabla 5.8). Esto significa que se puede disminuir la frecuencia del sistema de 100MHz hasta 33MHz sin afectar el procesamiento actual del ASP, en términos de operaciones trigonométricas.

El aumento en rendimiento significa una disminución del 84.3% en la potencia consumida y 25% en área utilizada de la presente solución en precisión simple, con *clock gating*, y ejecutando a una frecuencia de 33MHz, con respecto a la solución anterior en 100MHz, lo que la vuelve más eficiente en términos de potencia y área a un mismo procesamiento de datos (basado en la Figura 5.26).

Además, la velocidad máxima de la presente versión alcanza los 537MHz en precisión simple y 446MHz en precisión doble. Un aumento del 10% y el 5% , respectivamente, con respecto a la solución anterior (ver tabla 5.11).

Por otro lado, sobre la tecnología 0.13μm, se puede observar en la Figura 5.28 que la implementación del multiplicador de Karatsuba simple es favorable, junto a la implementación de la biblioteca de `DesignWare` en términos de área. Adicionalmente, la implementación de la presente FPU con *clock gating* ejecutando a una frecuencia de 33MHz consume una potencia

estimada de $0.3mW$, utilizando el multiplicador de Karatsuba recursivo implementado (ver la Figura 5.29).

Finalmente, se puede analizar en la tabla 5.14 un resumen de los parámetros que se proponen mejorar en el presente trabajo.

Capítulo 6

Conclusiones

Se concluye, de las tablas 3.2 y 3.3, que existen FPUs de código libre disponibles en la literatura con las cuales se pueden realizar comparaciones en términos de variación de la palabra de los operandos, área, temporizado y potencia.

Adicionalmente, también se encuentra en la sección 4.3 que con el movimiento del software libre, y la necesidad de comparar sistemas embebidos de alto rendimiento, se han creado *benchmarks* de código abierto como `DSPStone` y `MediaBench`, los cuales son puntos de referencia utilizados ampliamente en la literatura.

Es posible realizar una medición y comparación de una FPU por medio de una metodología de *benchmarking*, pero en este trabajo la dimensión de los objetivos imposibilitaron el cumplimiento de la realización de la medición mismo, como se discutió en la sección 4.3.2.

Se determina, de las Figuras 5.19 y 5.20, que el multiplicador recursivo de Karatsuba para largos de palabra pequeños es menos eficiente en términos de área y potencia que la implementación simple.

Además, se concluye que el presente sumador tiene un speedup de 8 con respecto a la implementación anterior, y una frecuencia de operación máxima de 350MHz en precisión simple y 318MHz en precisión doble (ver Tablas 5.7 y 5.6). Esto a cambio de un aumento en la potencia y el área del diseño, los cuales se pueden manejar al disminuir la frecuencia de reloj del sistema y utilizar *clock gating* (ver Figuras 5.22 y 5.23).

Por otro lado, se determinó que la presente unidad de CORDIC es 3 veces más rápida (ver Tabla 5.11), una disminución en el área utilizado y las celdas utilizadas con respecto a la solución anterior (Figuras 5.25 y 5.24). En constraste, la presente solución sin *clock gating* presenta un ligero aumento en la potencia utilizada (ver Figura 5.26).

Finalmente, se concluye que la presente FPU con el multiplicador de Karatsuba recursivo puede llegar a consumir $0.3mW$ siendo ejecutado a una frecuencia de reloj de 33MHz y utilizando métodos de *clock gating* (ver Figura 5.30).

Capítulo 7

Recomendaciones

Como se menciona en [13], los módulos centrales para la normalización son el Contador de Ceros y el Desplazador de Barril. Adicionalmente, estas unidades con el operador de las mantisas, constituyen las secciones más costosas en términos de potencia y área consumida de los operadores de suma y multiplicación en coma flotante [87, 31, 88].

Es por esto que se recomienda el llevar a cabo las variaciones de Contadores de Ceros presentados en [89] para encontrar la variación del detector de ceros que sea más beneficiosa para la aplicación de baja potencia que se tiene en el presente proyecto, en una tecnología comercial.

En [58] se discute el efecto que el tamaño del ancho de palabra que la FPU tiene sobre la potencia consumida y el área utilizada. Es con el fin de reducir la potencia disipada, que se recomienda también el analizar el efecto que el reducir el ancho de palabra de la FPU puede tener sobre la clasificación efectiva de patrones acústicos que realiza el algoritmo ejecutado sobre el ASP. Para esta tarea, se recomienda el utilizar las bibliotecas `VFLOAT` y `FloPoCo` como punto de partida.

Siempre con el fin de reducir la potencia consumida, se recomienda la utilización de circuitos aproximados en los operadores de las mantisas. Unidades aritméticas de sumado y multiplicación aproximada presentan grandes ganancias en la reducción del gasto energético (hasta un 50%). En [90] se presenta la implementación de 3 tipos de sumadores aproximados en un operador de suma/resta en coma flotante y sus correspondientes resultados.

Finalmente, con el fin de reducir la utilización de recursos lógicos en las etapas de normalización y redondeo en los operadores de suma/resta y multiplicación en coma flotante, se recomienda el analizar la integración de la unidad de suma segmentada con la multiplicación, realizando pruebas sobre las posibles ganancias en el consumo energético de tal implementación con el algoritmo de HMM y tomando en cuenta la perdida del paralelismo que se tiene con ambas unidades separadas (basado en [91]).

Bibliografía

[1] P. Alvarado and E. Salas, "Implementación de un banco de filtros digitales multitasa para la estimación energética espectral en una aplicación de protección ambiental."

[2] D. A. Patterson, J. L. Hennessy, and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann series in computer architecture and design, Morgan Kaufmann, 4th ed ed.

[3] C. Salazar, "Implementación de un microprocesador de aplicación especifica para la ejecución del algoritmo de modelos ocultos de Markov en el reconocimiento de patrones acusticos.."

[4] T. Adam, "How to use the cordic algorithm in your fpga design — ee times."

[5] J. Quiros, "Diseño, implementación y mejoras sobre una unidad de suma (FPU) con operaciones no lineales.."

[6] C. Brunelli, F. Campi, C. Mucci, D. Rossi, T. Ahonen, J. Kylliäinen, F. Garzia, and J. Nurmi, "Design space exploration of an open-source, IP-reusable, scalable floating-point engine for embedded applications," vol. 54, no. 12, pp. 1143–1154.

[7] R. Usselmann, "Floating Point Unit :: Overview :: OpenCores."

[8] A. Ehliar, "Area efficient floating-point adder and multiplier with IEEE-754 compatible semantics," in *Field-Programmable Technology (FPT), 2014 International Conference on*, pp. 131–138, IEEE.

[9] F. Brosser, H. Y. Cheah, and S. A. Fahmy, "Iterative floating point computation using FPGA DSP blocks," in *2013 23rd International Conference on Field Programmable Logic and Applications*, pp. 1–6.

[10] "Xilinx Core Generator."

[11] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *International Conference on Field Programmable Logic and Applications*, pp. 657–666, Springer.

[12] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 715–720.

[13] F. Lopez, "Diseño e implementación de hardware para optimizar la Unidad Aritmetica de Coma Flotante de un procesador de aplicación especifica."

[14] D. Goldberg, "What every computer scientist should know about floating-point arithmetic."

[15] A. Hidalgo, "Implementación en hardware del Sistema de Reconocimiento de Patrones Acustico (SiRPA)."

[16] A. Chacon and J. P. Alvarado, "Sistema electrónico integrado en chip (SoC) para el reconocimiento de patrones de disparos y motosierras en una red inalámbrica de sensores para la protección ambiental,"

[17] D. Rodriguez, "Diseño e implementación de una unidad aritmético-lógica de coma flotante para un procesador de aplicación especifica.."

[18] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 4th ed ed. OCLC: ocn473447233.

[19] W. Kahan, "IEEE 754: Standard for Binary Floating-Point Arithmetic."

[20] C. Paar, "Generalizations of the Karatsuba Algorithm for Polynomial Multiplication."

[21] C. Rebeiro and D. Mukhpodhyay, "Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier," pp. 706–711, IEEE.

[22] H. Fan, J. Sun, M. Gu, and K.-y. Lam, "Overlap-free Karatsuba-Ofman Polynomial Multiplication Algorithms for Hardware Implementations,"

[23] R. K. Kodali, S. K. Gundabathula, and L. Boppana, "FPGA implementation of IEEE-754 floating point Karatsuba multiplier," in *Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2014 International Conference on*, pp. 300–304, IEEE.

[24] J. E. Volder, "The CORDIC Trigonometric Computing Tecnique."

[25] A. Tang, L. Yu, F. Han, and Z. Zhang, "CORDIC-based FFT real-time processing design and FPGA implementation," in *2016 IEEE 12th International Colloquium on Signal Processing Its Applications (CSPA)*, pp. 233–236.

[26] N. Jain and B. Mishra, "DCT and CORDIC on a novel configurable hardware," in *2015 IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia)*, pp. 51–56.

[27] T. T. Hoang, H. T. Nguyen, X. T. Nguyen, C. K. Pham, and D. H. Le, "High-performance DCT architecture based on angle recoding CORDIC and Scale-Free Factor," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pp. 199–204.

[28] N. Jarray, M. Elhaji, and A. Zitouni, "Low complexity and efficient architecture of 1D-DCT based Cordic-Loeffler for wireless endoscopy capsule," in *2015 12th International Multi-Conference on Systems, Signals Devices (SSD)*, pp. 1–5.

[29] P. K. Meher, J. Valls, T. B. Juang, K. Sridharan, and K. Maharatna, "50 Years of CORDIC: Algorithms, Architectures, and Applications," vol. 56, no. 9, pp. 1893–1907.

[30] C. Brunelli, F. Garzia, J. Nurmi, C. Mucci, F. Campi, and D. Rossi, "A FPGA implementation of an open-source floating-point computation system," in *2005 International Symposium on System-on-Chip*, pp. 29–32, IEEE.

[31] "Proceedings 2001 / Design Automation Conference: Las Vegas Convetion Center, Las Vegas, NV, June 18 - 22, 2001." OCLC: 247993469.

[32] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 330–335, IEEE Computer Society.

[33] D. Lundgren, "FPU Double VHDL :: Overview :: OpenCores."

[34] G. Marcus, "Floating Point Adder and Multiplier :: Overview :: OpenCores."

[35] J. Al-Eryani, "FPU :: Overview :: OpenCores."

[36] U. G, R. Dhanabal, and S. K. Sahoo, "Implementation of a High Speed Single Precision Floating Point Unit using Verilog," vol. NCVES, no. 1, pp. 32–36.

[37] C. Bhaskar, "Implementation of Area Efficient IEEE -754 Double Precision Floating Point Arithmetic Unit Using Verilog," vol. 2, pp. 15–21.

[38] Y.-T. Pai and Y.-K. Chen, "The Fastest Carry Lookahead Adder.," in *DELTA*, pp. 434–436.

[39] S. Smith, *Digital Signal Processing-A Practical guide for Engineers and Scientists*. Elsevier Science.

[40] J. Bannur and A. Varma, "The VLSI implementation of a square root algorithm," in *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*, pp. 159–165.

[41] K. C. Johnson, "Algorithm 650: Efficient Square Root Implementation on the 68000," vol. 13, no. 2, pp. 138–151.

[42] A. Adams, S. D. Nemirovsky, M. Sagreras, and E. Daniello, "Una aproximación al diseño de un Sumador de Punto FLotante en Doble Precisión basado en el Estandar IEEE 754."

[43] J. Chaitanya and R. Rama Koteswara, "Implementation of Single Precision Floating Point Multiplier," vol. 2, no. 9, pp. 1373–1377.

[44] C. Tom, E. Rippey, and A. Li, "OpenGL Accelerator."

[45] A. Ehliar, "High Radix Floating Point Adder and Multiplier (HRFP_16)."

[46] "LogiCORE IP Floating-Point Operator v6.1: Product Specification."

[47] "Axi Reference Guide."

[48] P. Karlstrom, W. Zhou, and D. Liu, "Implementation of a Floating Point Adder and Subtracter in NoGAP, A Comparative Case Study," pp. 68–72, IEEE.

[49] M. K. Jaiswal and R. C. Cheung, "High Performance FPGA Implementation of Double Precision Floating Point Adder/Subtractor," vol. 4, no. 4, pp. 71–80.

[50] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors." http://prunel.ccsd.cnrs.fr/ensl-00174627.

[51] F. de Dinechin, "Welcome to the FloPoCo project."

[52] F. de Dinechin, "HiPEAC 2013 - FloPoCo tutorial."

[53] M. Leeser, "A Library of Parameterized Hardware Modules for Floating Point Arithmetic and Its Use."

[54] X. Wang, M. Leeser, and H. Yu, "A parameterized floating-point library applied to Multispectral image clustering," in *Proceedings of the 7th Annual MAPLD International Conference*.

[55] M. Leeser and X. Wang, "Variable precision floating point division and square root."

[56] X. Wang, S. Braganza, and M. Leeser, "Advanced Components in the Variable Precision Floating-Point Library," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '06, pp. 249–258, IEEE Computer Society.

[57] X. Wang and M. Leeser, "A Truly Two-dimensional Systolic Array FPGA Implementation of QR Decomposition," vol. 9, no. 1, pp. 3:1–3:17.

[58] X. Wang and M. Leeser, "VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware," vol. 3, no. 3, pp. 16:1–16:34.

[59] X. Fang and M. Leeser, "Vendor agnostic, high performance, double precision Floating Point division for FPGAs," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pp. 1–5, IEEE.

[60] K. S. Hemmert and K. D. Underwood, "Open source high performance floating-point modules," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 349–350, IEEE.

[61] M. K. Jaiswal and N. Chandrachoodan, "Efficient implementation of IEEE double precision floating-point multiplier on FPGA," in *2008 IEEE Region 10 and the Third International Conference on Industrial and Information Systems*, pp. 1–4, IEEE.

[62] "RCL Floating Point Library."

[63] A. Chacon-Rodriguez, C. Meza, C. Salazar-Garcia, D. Rodriguez-Valverde, J. Quiros, and A. Cervantes, "Implementation of an open core IEEE 754-based FPU with non-linear arithmetic support,"

[64] E. Catovic, "GRFPU - High Performance IEEE-754 Floating-Point Unit."

[65] T.-J. Kwon, J. Sondeen, and J. Draper, "Design trade-offs in floating-point unit implementation for embedded and processing-in-memory systems," in *2005 IEEE International Symposium on Circuits and Systems*, pp. 3331–3334, IEEE.

[66] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 149, IEEE.

[67] M. K. Jaiswal and R. C. Cheung, "Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support," vol. 44, no. 5, pp. 421–430.

[68] C. Minchola and G. Sutter, "A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier," pp. 59–64, IEEE.

[69] J. Joven, P. Strict, D. Castells-Rufas, A. Bagdia, G. De Micheli, and J. Carrabina, "Hw-sw implementation of a decoupled fpu for arm-based cortex-m1 socs in fpgas," in *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pp. 1–8, IEEE.

[70] D. Hornaes, "Low-Cost FPU: Specification, Implementation and Verification,"

[71] "Dictionary and Thesaurus — Merriam-Webster."

[72] "Standard Performance Evaluation Corporation." Page Version ID: 741345987.

[73] "List of All BDTI Benchmarks — www.bdti.com."

[74] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*. SIAM. Google-Books-ID: YTXzIy1dTfsC.

[75] J. Walker, "FFBENCH: Fast Fourier Transform Benchmark."

[76] W. Goh and K. Venkat, "Application Report: MSP430 Competitive Benchmarking."

[77] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop on Workload Characterization, 2001. WWC-4*, pp. 3–14.

[78] "ICE - Institute for Communication Technologies and Embedded Systems: Entry."

[79] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *, Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997. Proceedings*, pp. 330–335.

[80] B. Bishop, T. Kelliher, and M. Irwin, "A detailed analysis of MediaBench," pp. 448–455, IEEE.

[81] "MediaBench II."

[82] "MiBench version 1."

[83] Kingshuk Karuri and Rainer Leupers, *Application Analysis Tools for ASIP Design Application Profiling and Instruction-Set Customization*. Springer Science + Business Media.

[84] C. Rebeiro and D. Mukhpodhyay, "Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier," pp. 706–711, IEEE.

[85] M. Machhout, M. Zeghid, B. Bouallegue, R. Tourki, and others, "Efficient hardware architecture of recursive karatsuba-ofman multiplier," in *Design and Technology of Integrated Systems in Nanoscale Era, 2008. DTIS 2008. 3rd International Conference on*, pp. 1–6, IEEE.

[86] "DesignWare Library - Datapath and Building Block IP."

[87] Woo-Chan Park, Tack-Don Han, and Shin-Dug Kim, "Efficient simultaneous rounding method removing sticky-bit from critical path for floating point addition," pp. 223–226, IEEE.

[88] T.-J. Kwon, J.-S. Moon, J. Sondeen, and J. Draper, "A 0,18 um implementation of a floating-point unit for a processing-in-memory system," in *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on*, vol. 2, pp. II–453, IEEE.

[89] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, "Low-Power Leading-Zero Counting and Anticipation Logic for High-Speed Floating Point Units," vol. 16, no. 7, pp. 837–850.

[90] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *European Solid-State Circuits Conference, ESSCIRC Conference 2016: 42nd*, pp. 465–468, IEEE.

[91] A. Mathur and S. Saluja, "Improved merging of datapath operators using information content and required precision analysis," in *Design Automation Conference, 2001. Proceedings*, pp. 462–467, IEEE.

Apéndice A

CORDIC

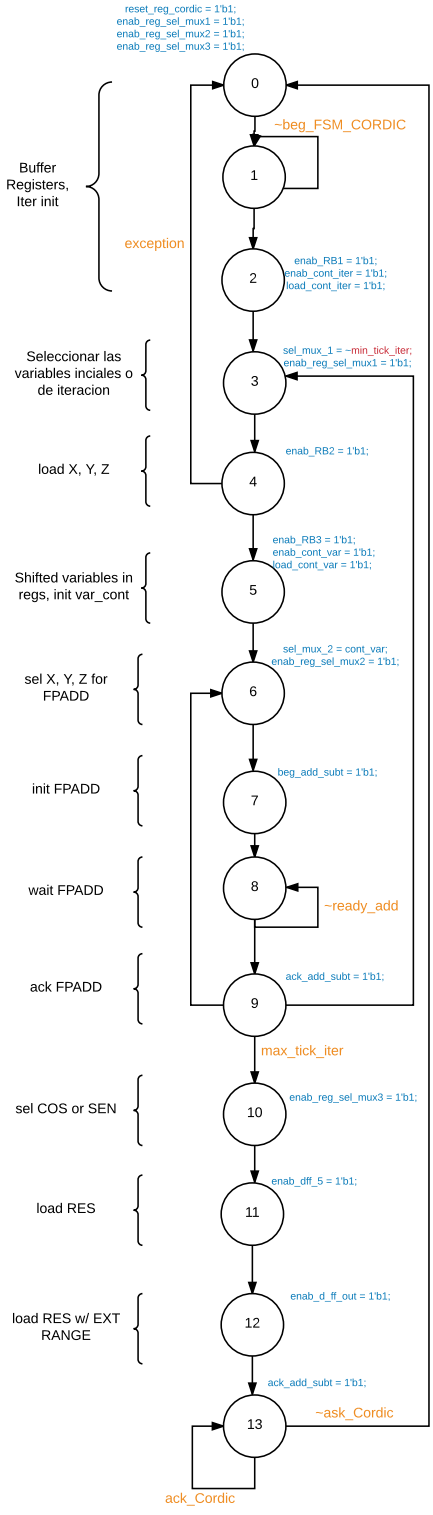

Figura A.1: Implementación de Quiros

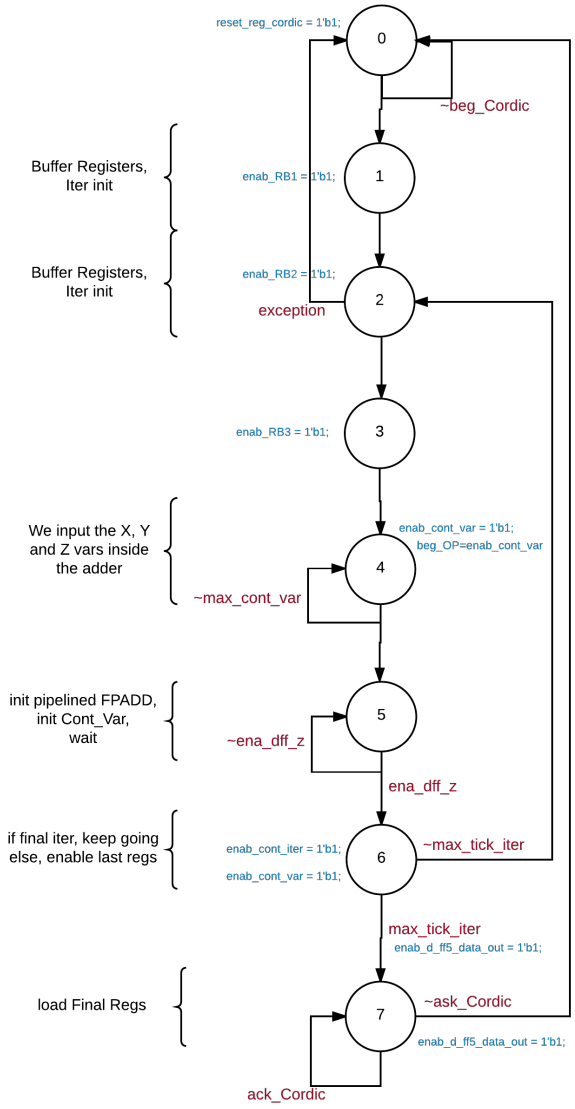

Figura A.2: Presente implementación

Figura A.3: Diagramas de las máquinas de estados para las versiones del CORDIC

Tabla A.1: Descripción de la interfaz de la unidad de control del operador CORDIC

Nombre	Tipo	Descripción
clk	Entrada	Señal global de reloj
rst	Entrada	Señal global de reset
beg_fsm- CORDIC	Entrada	Señal para el inicio de la operación seno/coseno
ACK_FSM CORDIC	Entrada	Señal proveniente del modulo que recibe el resultado, indicado que el dato ha sido recibido.
exception	Entrada	Señal que indica la entrada de un par de números no representables por el formato IEEE 754.
max_tick_iter	Entrada	Señal que indica la máxima cuenta, en el contador de iteraciones.
max_tick_var	Entrada	Señal que indica la máxima cuenta, en el contador de variables.
enab_dff_z	Entrada	Señal que confirma la carga de las variables Xn, Yn, y Zn provenientes de la suma/resta en coma flotante.
reset_reg_cordic	Salida	Reinicia el valor de las diferentes etapas secuenciales utilizadas en operaciones anteriores.
ready_CORDIC	Salida	Indica la existencia de un nuevo resultado a la salida de la unidad. La señal es de lógica positiva.
beg_add_subt	Salida	Señala a la unidad de suma/resta el comienzo de una nueva operación.
enab_cont_iter	Salida	Señal que habilita y comienza la operación del contador de iteraciones.
enab_cont_var	Salida	Señal que habilita y comienza la operación del contador de variables.
enab_RB1, enab_RB2, enab_RB3	Salida	Señales que habilitan los registros de las diferentes etapas de la actual arquitectura.
enab_d_ff5- data_out	Salida	Señal que habilita el registro de salida de la operación seno/coseno.