Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño e implementación de hardware para optimizar la Unidad Aritmética de Coma Flotante de un procesador de aplicación específica

Informe de Proyecto de Graduación para optar por el título de

Ingeniero en Electrónica con el grado académico de Licenciatura

Francis Alexander López Montero

Borrador de 8 de junio de 2016

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 hardware para optimizar la Unidad Aritmética de Coma Flotante de un procesador de aplicación específica, realizado por el señor Francis Alexander López Montero 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. Roberto Molina Robles

Profesor lector

Ing. Leonardo Rivas Arce

Profesor lector

Ing. Alfonso Chacón Rodríguez

Profesor asesor

Cartago, 16 de junio, 2016

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Francis Alexander López Montero

Cartago, 8 de junio de 2016

Céd: 1-1480-0347

Resumen

Este documento trata sobre la optimización operacional de una Unidad Aritmética de Coma Flotante (FPU) para arquitecturas de 32 y 64 bits según el estándar IEEE 754 y con tres operaciones básicas: Suma, Resta y Multiplicación. En ella se implementan unidades de hardware encontradas en la literatura (desplazador de barril, detector de ceros precedentes, y multiplicador de Karatsuba) con el fin de optimizar el tiempo de operación y los recursos lógicos requeridos. La unidad es verificada sobre una FPGA.

Palabras clave: Punto Flotante, Desplazador de barril, multiplicador de Karatsuba, Operación aritmética, Detector de ceros

Abstract

This paper discusses the design of a Floating Point Arithmetic Unit (FPU) for 32-bit and 64-bit architectures using the IEEE 754 standard with the three basic operations: addition, subtraction and multiplication. This hardware uses units proposed in the literature (Barrel Shifter, Leading Zero Detector, Karatsuba's multiplication) in order to optimize the operating time and the required logical resources of the FPU, which is verified on a FPGA.

Keywords: Floating Point Arithmetic, barrel shifter, Karatsuba's multplication, leading zero detector

A mi madre, le estaré eternamente agradecido

Agradecimientos

Primero agradecer a mi familia que me ha dado un cariño incondicional en todos mis años de vida, en especial a mi madre Maribell Quesada Montero ya que ha sacrificado mucho tiempo y esfuerzo para yo poder lograr lograr estar graduándome y formar parte de mi desarrollo integral como persona.

Agradecimientos a los profesores, el Dr. Alfonso Chacón Rodríguez y al M.Sc. Carlos Salazar García, por la guía en todas las etapas del proyecto, así como en confiar en mi persona para la realización del mismo. También se le agradece al ingeniero, el Lic. Diego Rodríguez Valverde por su voto de confianza para continuar su trabajo.

También un agradecimiento a los compañeros del DCILab, en especial a Jeffry Quirós Fallas que ha sido un excelente colega y un gran amigo durante todos estos años de carrera universitaria.

Por ultimo se les agradece a todas las personas que, tanto en la universidad como en la vida, han aportado su apoyo y cariño. Gracias a sus palabras de aliento me han motivado a continuar en esta etapa de la vida, y por ello ha sido posible la realización de este proyecto.

Francis Alexander López Montero

Cartago, 8 de junio de 2016

Índice general

Índice de figuras

Índice de tablas

Revisar

Capítulo 1

Introducción

Una Unidad Aritmética De Punto Flotante (FPU por Floating Point Unit en inglés) ha sido diseñada como parte de un proyecto que está realizando el Laboratorio de Diseño De Circuitos Integrados (DCILab) del Instituto Tecnológico de Costa Rica: un Sistema integrado en Chip (SoC) llamado Sistema de Reconocimiento de Patrones Acústicos (SiRPA) [15] , el cual está orientado hacia la protección de zonas ambientales protegidas. Este proyecto pretende desarrollar un sistema que reconozca el sonido de armas de fuego y motosierras mediante el uso de técnicas como los Modelos Ocultos de Márkov (HMM) los que a partir de parámetros observables, se pueden encontrar parámetros desconocidos y con ello realizar un análisis y toma de decisiones.

Algunos sistemas y estructuras desarrollados en este proyecto han sido ya comprobados exitosamente en [16] , [8] , [10] , incluyendo además ya alguna experiencia en el desarrollo de unidades de coma flotante [7]. Estos resultados, no obstante, motivaron la necesidad de un procesador de aplicación específica con capacidad de aritmética en coma flotante, para cumplir con la precisión y eficiencia buscados [15], [12].

El problema actual presentado, y objetivo principal en la solución del proyecto, es que la FPU diseñada hasta el momento [12] es presentada como una prueba de concepto en que es posible, a partir de un lenguaje HDL, tener una FPU basada en el estándar IEEE 754, para 32 y 64 bits, con 3 operaciones básicas (suma, resta y multiplicación). Pero el diseño tiene deficiencias en los siguientes puntos:

- Mal manejo de recursos lógicos (varios módulos que cumplen con las mismas características y funciones).
- Estados extras e innecesarios en la Máquina de Estados Finitos (FSM).
- Módulos que no tienen dependencia en la ruta de datos pero se vuelven dependientes por el diseño de la FSM.
- Iteraciones en los procesos de normalización para operandos y resultado que requieren de muchos ciclos de reloj.
- En la arquitectura diseñada de la multiplicación para precisión doble, no se puede ejecutar a una frecuencia de 100 MHZ según las pruebas sobre FPGA, lo que lo vuelve el cuello de botella de la FPU.

Estas falencias se exacerbaron cuando se intentaron implementar otras funciones aritméticas que, para sus cálculos, requería de muchos ciclos de reloj para resolver y daban problemas en simulaciones de *timing closure* como consecuencia del mismo.

La solución que se propone en este proyecto es la eliminación de la parte secuencial en la normalización de los valores de operandos y resultado mediante módulos de hardware que se mencionarán en el documento, el rediseño de la FSM y módulos requeridos para una mejora en la utilización de recursos lógicos. Se realizará un diseño a nivel de RTL de la nueva arquitectura para que con menos unidades se puedan calcular más rápido el resultado de la operación.

1.1 Objetivos

1.1.1 Objetivo General

Optimizar las unidades de hardware de una unidad aritmética de coma flotante verificada en una FPGA (Field Progammable Gate Array), para reducir tanto el tiempo de ejecución de la misma, así como en la cantidad de recursos del sistema.

1.1.2 Objetivos Específicos

- Evaluar las soluciones propuestas existentes en la literatura para encontrar una solución computacional óptima para las operaciones más críticas de la FPU bajo desarrollo.
- Implementar en HDL los algoritmos escogidos en el objetivo anterior, integrándolo dentro de la FPU.
- Verificar la funcionalidad y optimización sobre FPGA en ahorro de recursos lógicos y velocidad en la unidad completa de la FPU.

Capítulo 2

Marco teórico

En esta sección se explicarán acerca de los fundamentos teóricos en la resolución del problema. Primero se profundizará acerca del estándar IEEE 754 y la representación de los números de tipo flotante en formato binario para 32 y 64 bits. En el segundo punto se dará la descripción de los algoritmos para realizar las operaciones aritméticas de suma, resta y multiplicación en este estándar. Por último, se explicará sobre los módulos en hardware de la FPU seleccionados para la solución planteada, y su papel en cada operación aritmética.

2.1 Estándar IEEE 754

2.1.1 Descripciones generales

El estándar IEEE 754 [4], es usado actualmente en muchos procesadores y computadoras, ya que permite una notación científica de valores fraccionarios en formato binario, y con ello, se logra representar una gran gama de valores (dependiente de su precisión determinada por la cantidad de bits).

La representación para cada valor binario se basa en que todo valor real, puede ser representado en una notación científica como se muestra en la ecuación (2.1) donde **s** es el signo del valor, **b** la base de la notación científica, **e** el exponente del valor y **m** significando o mantisa, que representa la parte fraccionaria del valor.

$$F = (-1)^s * b^e * 1.m \tag{2.1}$$

Este estándar toma en cuenta las siguiente consideraciones para su representación en binario:

- Para no tener que desperdiciar un bit en representar al exponente en complemento a dos, se le suma un valor de escala (también conocido como *bias* o sesgo) que depende de la cantidad de bits que se representa el valor. Este nuevo exponente se le conoce como *exponente normalizado*.
- Para representar valores binarios, $b = 2$

- Para el segmento del significando se evita almacenar el bit más significativo de la parte fraccionaria, ya que siempre debe ser uno (1).
- El bit de signo, tomará valor de cero (0) para valores positivo y uno para valores negativos.

De esta forma la trama de bits para representar cada valor se muestra en la figura 2.1 La cantidad de bits para representar cada segmento depende del formato de precisión, que este proyecto se escogió fuera de precisión simple (32 bits) y precisión doble (64 bits). En la tabla 2.1 se menciona la cantidad de bits para representar cada trama según el estándar IEEE 754 [1], así como el valor de *bias* al que se debe sumar al exponente normalizado.

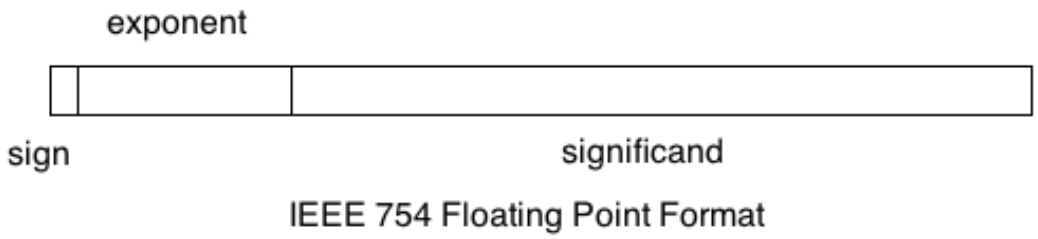

Figura 2.1: Formato de trama de bits para el estándar IEEE 754. Imagen tomada de [**?**]

Tabla 2.1: Parámetros del estándar IEEE 754 para las la representación de valores en precisión simple y doble

Precisión	Simple	Doble
Exponente	8 bits	11 bits
Mantisa	23 bits	52 bits
Bias	127	1023

2.1.2 Valores característicos y límites

La interpretación de los valores reales en el estándar IEEE 754 está limitado por la máxima y mínima representación del exponente. De no conocer cuales son estos límites se pueden presentar errores en los cálculos aritméticos por una mal manejo de los datos a la hora de normalizar el resultado de la operación. En la tabla 2.2 se muestran la máxima y mínima representación de los valores tanto decimal como en binario.

Tabla 2.2: Valores máximos y mínimos representables en el estándar IEEE 754

Precisión	Límite	Representación IEEE 754	Valor decimal
Simple	Mínimo	00800000_h	1.175494351E-38 $_d$
	Máximo	$7f7fffff_h$	3.402823466E+38 $_d$
Doble	Mínimo	0010000000000000_h	$2.2250738585072014E\text{-}308_d$
	Máximo	$7fefffffffffffff_h$	$1797693134862315 7E+308_d$

En los casos en que se superen los límites, se tienen representaciones para valores especiales que se muestran en la tabla 2.3. Los valores mostraron se muestran en el resultado de salida en cada caso cumpliendo la condición dependiente de cada operación a realizar.

Tabla 2.3: Valores para representar casos excepcionales en el estándar IEEE 754

Precisión	Valor característico	Representación IEEE 754
Simple	Sobre-desborde (*Overflow*) ($+\infty$)	$7f800000_h$
	Sub-desborde (*Underflow*) ($-\infty$)	$ff800000_h$
	Cero (*Zero*)	00000000_h
Doble	*Overflow* ($+\infty$)	$7ff0000000000000_h$
	Underflow ($-\infty$)	$fff0000000000000_h$
	Zero	0000000000000000_h

También se encuentran en estos casos los valores denormalizados. Estos son valores que se encuentran en un rango entre el valor mínimo representable y cero. Estos valores no son posible representarlos mediante la ecuación 2.1 por el límite en el exponente mínimo y requieren de una lógica dedicada aparte para realizar operaciones aritméticas. Estos valores no entran en las condiciones del proyecto.

2.2 Operaciones aritméticas en el estándar IEEE 754

El manejo de los valores reales en punto flotante requieren de una mayor complejidad a la hora de realizar cálculos matemáticos, ya que cada segmento de los valores requiere de su propia lógica y se debe normalizar los datos para que cumplan con el formato usado.

Para esta sección se explicarán los algoritmos para realizar las operaciones realizadas en este proyecto: suma, resta y multiplicación, así también los casos especiales de resultados.

2.2.1 Operandos

Para realizar operaciones aritméticas en punto flotante, los operandos deben ser representados como lo indica el estándar IEEE 754. Solo que, para la parte fraccionaria, se requiere añadir el bit implícito durante el cálculo de la operación; de no ser así conlleva a que el valor de mantisa durante la operación no sea igual al representado, dando errores en el resultado.

También se añade en el caso de la suma/resta, dos bits extra que representan el bit de guardia y el bit de redondeo en las posiciones menos significativas de la mantisa, que serán usados para determinar si el resultado de la operación requiere o no ser ajustado por redondeo. Esto mejora la precisión del resultado, ya que hay un error intrínseco entre la representación en el estándar IEEE 754 y el valor representado en decimal.

En el caso de la multiplicación se usa la trama menos significativa del resultado de la multiplicación para la evaluación del redondeo en la mantisa, por lo que no requiere de estos bits de redondeo.

2.2.2 Operación suma/resta

Algoritmo

La operación suma y la resta funcionan igual para el estándar IEEE 754 en cuestión de los pasos a realizar la operación, ya que la operación a realizar depende del signo de la operación, pero este puede cambiar según los signos de los operandos. A partir de este punto se consideran como una misma operación a nivel de hardware y a nivel de algoritmo.

Los pasos para realizar esta operación son [2]:

1. Pasar el valor de punto flotante a la representación en notación científica. Agregar el bit implícito y los bits de redondeo y guardia en la trama de la mantisa.

2. Calcular la diferencia entre los exponentes de ambos operandos. Este valor será la cantidad de desplazamientos a la derecha que se le deberán aplicar al operando de menor magnitud para ser normalizado.

3. Normalizar el operando menor para igualarlos con respecto a los exponentes.

4. Calcular la operación suma/resta correspondiente entre las mantisas de ambos operandos.

5. En caso de que la mantisa de resultado no cumpla con el formato correspondiente, deberá ser normalizada con los desplazamientos que se requieran para tener un uno en el bit más significativo de la mantisa. También se debe ajustar el exponente de resultado por cada desplazamiento según sea desplazamiento a la izquierda (decrecimiento del exponente) o a la derecha (crecimiento del exponente).

6. Calcular el bit de signo de resultado, que será dependiente tanto del signo de los operandos como de la operación que se está realizando.

7. Realizar el redondeo del resultado según los valores de bit de redondeo y guardia. Volver a normalizar si es requerido.

8. Representar el resultado en notación de punto flotante, truncando el bit más significativo de la mantisa (bit implícito) y los bits de redondeo y de guardia.

Excepciones

En la tabla 2.4 se muestran los casos en los cuales el resultado no es representable ya sea por un sobre-desbordamiento del valor representable (*overflow*) o un sub-desbordamiento del valor representable (*underflow*) al momento de la normalización del resultado. También se mencionan los casos en que puede haber un error en la operación, dependiendo de los operandos de entrada.

Tabla 2.4: Casos en los que el resultado de la Suma/resta se representa con valor excepcional

Descripción	Resultado de salida
El resultado en el exponente es mayor al máximo número representable según el formato de precisión	*Overflow*
El resultado en el exponente es menor al mínimo número representable según el formato de precisión	*Underflow*
La operación a realizar es una resta y los 2 operandos son iguales	*Zero*
Uno de los operandos es infinito	*Overflow* / *Underflow* (dependiendo del signo del operando infinito)
Uno de los operandos es cero	Operando $\neq 0$
Ambos operandos son infinito	Valor indefinido (*NaN*)

2.2.3 Operación multiplicación

Algoritmo

El algoritmo de la multiplicación en el estándar IEEE 754 requiere menos pasos para ejecutarse que en el caso de la suma/resta, pues no es necesaria la normalización de los operandos antes de ejecutar la operación. Los pasos para realizar la operación son los siguientes[3]:

1. Pasar el valor de punto flotante a la representación en notación científica. Agregar el bit implícito.

2. Para calcular el exponente de resultado se debe primero sumar ambos exponentes de los operandos y restarle a este, el valor de *bias* correspondiente a cada formato de precisión. Esto ya que en la suma se agrega dos veces el valor de *bias*, lo cual no da el resultado correcto de exponente normalizado.

3. Calcular la multiplicación de la mantisa de los operandos.

4. En caso de que la mantisa de resultado no cumpla con la representación correspondiente, deberá ser normalizada con los desplazamientos que se requieran. También se debe ajustar el exponente de resultado por cada desplazamiento.

5. Calcular el bit de signo de resultado, que será dependiente del signo de los operandos.

6. Realizar el redondeo del resultado y volver a normalizar si es requerido.

7. Representar el resultado en notación de punto flotante, truncando en la mantisa el bit implícito y los bits de redondeo y de guardia.

Excepciones

También en la multiplicación se deben considerar los casos para los cuales el resultado no puede representarse o casos en específico que se deben tomar en cuenta para tener el resultado correcto. Los casos mencionados se muestran en la tabla 2.5

Tabla 2.5: Casos en los que el resultado de la multiplicación se representa con valor excepcional

Descripción	Resultado de salida
El resultado en el exponente es mayor al máximo número representable según el formato de precisión	*Overflow*
La suma en los exponentes de los operandos es menor al *bias* según el formato de precisión	*Underflow*
Uno de los operandos es cero	*Zero*
Uno de los operandos es infinito	*Overflow* / *Underflow* (dependiendo del signo del operando infinito)
Multiplicar $0 \times \infty$	Valor indefinido (NaN)

2.3 Mejoras planteadas a nivel de hardware

En este punto se explicará sobre el funcionamiento que tiene cada uno de los nuevos módulos de hardware que se usaron para la solución de este proyecto. Se tomó la decisión de usarlos basado en los cambios en la lógica secuencial planteada en la primera versión de la FPU por una combinacional, para mejorar el tiempo de respuesta de la misma así como la reducción de recursos lógicos y de ciclos de reloj.

2.3.1 Desplazadores

Descripción

Un desplazador o *shifter* [6] se trata de una lógica combinacional en la que los bits del valor de entrada se les desplazan de posición según un valor constante o variable y que posee la misma cantidad de salidas que de entrada. Al ser desplazados, se deben agregar nuevos valores en los espacios que van quedando luego del desplazamiento, que dependen de la lógica combinacional diseñada; estos desplazadores se conocen como:

- Desplazadores lógicos: los bits que se van agregando, ya sea desplazamiento a la izquierda o derecha, son ceros.
- Desplazadores aritméticos: para el caso de un desplazamiento a la derecha, los bits que se ocupan dependen del bit de signo del valor de entrada.
- Desplazadores rotacionales: los bits que se se encuentran en las últimas posiciones, se desplazan a los nuevos espacios.

Hay diferentes diseños para crear un desplazador, pero en este proyecto se usó el conocido como desplazador de barril logarítmico. Este consiste en arreglos por nivel de multiplexores 2x1, donde cada entrada está conectada según los desplazamientos que correspondan cada nivel. En la figura 2.2 se puede ver un ejemplo de un desplazador de barril de ocho (8) bits en el que la entrada D1 se conecta según la ecuación $D1 = 2^j + i$ donde j es el nivel del arreglo j=0,1,...,7; e i es la entrada D0.

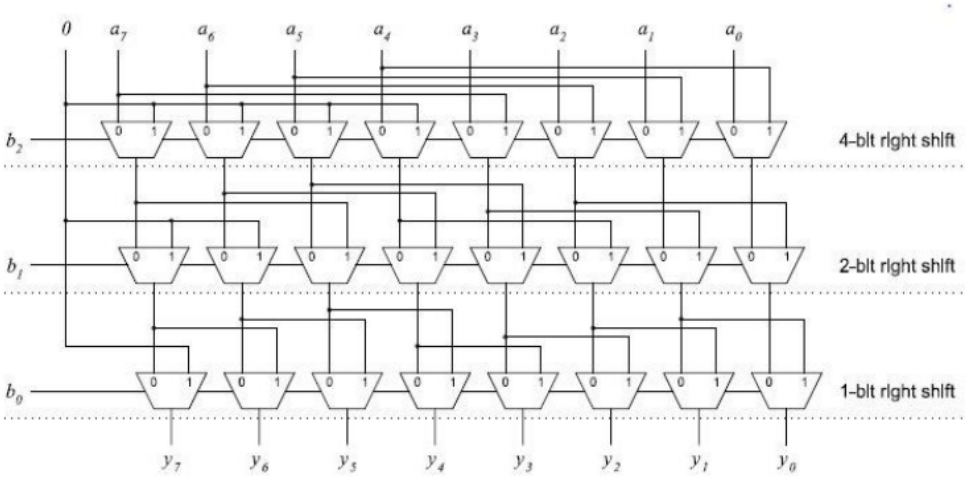

Figura 2.2: Desplazador de barril lógico en arquitectura logarítmica de ocho bits de ancho de datos. Imagen tomada de [14]

Desplazamientos bidireccionales

En la anterior figura se mostró la lógica de un desplazador de barril con la capacidad para una sola dirección, pero también existen diseños los cuales permiten que se logre un funcionamiento bidireccional, dependiente de una señal de control [13]. Esto añadirá una entrada extra a la lógica para la selección izquierda/derecha del desplazamiento.

- Bidireccional en serie: consta de 2 desplazadores unidireccionales de distintas direcciones, conectados en serie. Estos desplazadores funcionan de tal forma que, según la señal de control, uno de los desplazadores cumple con el desplazamiento requerido y el otro deja pasar el dato a la salida.
- Bidireccional en paralelo: también posee 2 desplazadores de distintas direcciones, solo que se conectan en paralelo y trabajan simultáneamente. Se le añade un arreglo extra de multiplexores para direccionar el resultado de uno de los desplazadores a la salida, según la señal de control de dirección.
- Bidireccional con reversión de dato: este diseño es más reducido a los anteriores ya que solamente requiere de 1 desplazador. Para lograr que realice desplazamientos bidireccionales, se le añaden arreglos de multiplexores extra en la entrada y salida del desplazador, conectados de tal forma que se hace una reversión entre los bits

más significativos con los menos significativos y viceversa. Al invertirle los datos, se ejecuta el desplazamiento "en reversa", dando como resultado que se pueda realizar desplazamientos a la izquierda y derecha. En la figura 2.3 se puede ver un ejemplo de este diseño.

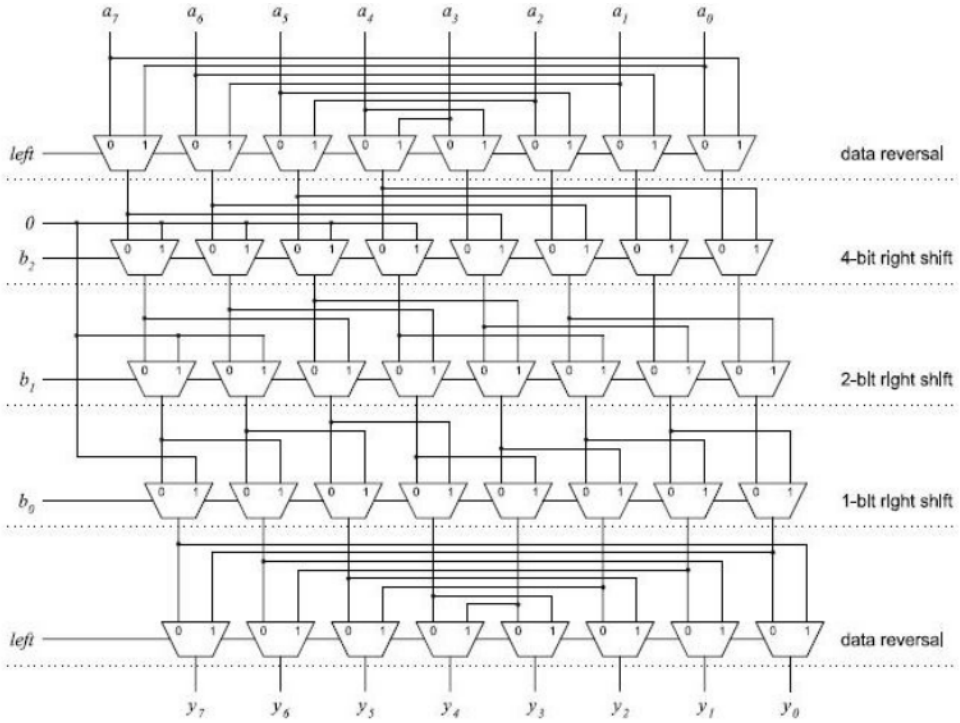

Figura 2.3: Desplazador de barril bidireccional con reversión de dato en arquitectura logarítmica de ocho bits de ancho de datos. Imagen tomada de [14]

2.3.2 Unidades para cálculos de normalización

Contrario con la normalización del menor operando, la cantidad de desplazamientos que requiere el resultado no se puede determinar a partir de los exponentes. En el diseño anterior, la normalización en el resultado se realiza mediante desplazamientos secuenciales e iterativos hasta cumplir con encontrar el primer uno en el bit más significativo.

Por ello, para este nuevo diseño, se planea utilizar una lógica combinacional la cual a partir de los datos de entrada, permita obtener el valor exacto de los desplazamientos que requiere la mantisa de resultado.

De la literatura investigada se encontraron dos soluciones:

- Unidad de anticipación de ceros precedentes (*LZA, leading zero anticipation*) [9]: consiste en realizar este cálculo a partir de los valores de entrada del sumador/restador, permitiendo que se pueda ejecutar en paralelo con el cálculo de la operación (figura 2.4).

- Unidad de detección de ceros precedentes (*LZD, leading zero detector*) [11]: calcula a partir del resultado de la operación suma/resta, el número de ceros posteriores al primer bit de valor uno. Este módulo trabaja en serie con el módulo de suma/resta (figura 2.5).

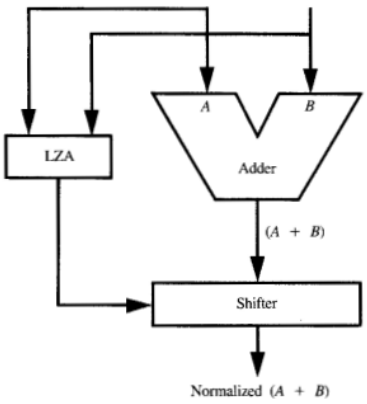

Figura 2.4: Arquitectura para la normalización del resultado de una suma utilizando un *LZA*. Imagen tomada de [9]

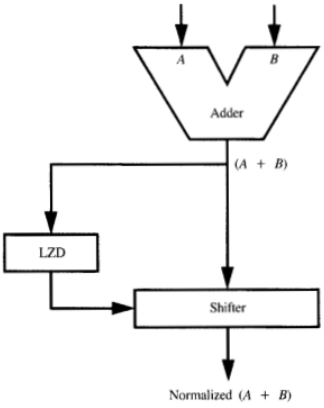

Figura 2.5: Arquitectura para la normalización del resultado de una suma utilizando un *LZD*. Imagen tomada de [9]

En una suma entre dos valores A y B, se puede tener los siguientes casos:

- $A > 0; B > 0; A + B > 0$
- $A < 0; B < 0; A + B < 0$
- $A > 0; B < 0; A + B > 0$
- $A > 0; B < 0; A + B < 0$

De los mencionados casos, en el estándar IEEE 754, solo se cumplen los casos en que el resultado es positivo. La mantisa es un valor de magnitud por lo que los operandos siempre son positivos, y según el algoritmo de suma, siempre se cumple que $|A| > |B|$, por lo que no existen resultados con complemento a 2 y no habrán casos en los que el resultado de la operación sea negativo. Esto reduce la cantidad de casos para los que se necesita el LZA.

La LZD es una unidad bastante más sencilla en los casos en que se cumplen los resultados descritos. Al no utilizarse complemento a a dos, solamente se requiere calcular cuántos bits de valor cero hay antes del primer bit de valor uno, con lo que el cálculo de la cantidad de desplazamientos requiera de menos lógica.

En el caso de la multiplicación, la posición correspondiente al bit implícito de resultado siempre se encuentra entre los primeros dos bits más significativos, por lo que para esta operación no se requiere de una lógica de detección de ceros precedentes.

2.3.3 Algoritmo de Karatsuba para multiplicación

El proceso de multiplicación a nivel de hardware requiere de bastantes recursos lógicos y llega a ser de las operaciones aritméticas que requieren bastante tiempo por el retardo como resultado del camino más crítico para la obtención del resultado. Por ello, existen múltiples algoritmos computacionales que tratan de acelerar esta operación.

Aquí se ha propuesto usar el algoritmo de Karatsuba [17]. El procedimiento consiste en segmentar la trama de los datos de entrada por la mitad y realizar operaciones parciales más pequeñas respecto al tamaño del dato para obtener el resultado, como se expresa en la ecuación (2.2). Estos cálculos reducen la carga de la ruta crítica, pues el hardware puede segmentarse por registros y ejecutarse en más de un ciclo de reloj [5].

$$X.Y = 2^n.X_l.Y_l + X_r.Y_r + 2^{\frac{n}{2}}.((X_l + X_r).(Y_l + Y_r) - X_l.Y_l - X_r.Y_r) \tag{2.2}$$

Donde:

- X,Y: Operandos de la multiplicación de n bits de datos.
- X_l, Y_l: Segmento más significativo del dato de los operandos.
- X_r, Y_r: Segmento menos significativo del dato de los operandos.
- $2^n, 2^{\frac{n}{2}}$: Cantidad de desplazamientos a la derecha requeridos.

Para explicar la solución más adelante, se utilizará la siguiente notación (ecuaciones 2.3,2.4,2.5 y 2.6) para definir cada expresión de la ecuación (2.2).

$$\alpha = X_l * Y_l \tag{2.3}$$

$$\beta = X_r * Y_r \tag{2.4}$$

$$\gamma = X_l + X_r \tag{2.5}$$

$$\theta = Y_l + Y_r \tag{2.6}$$

con la cual se representa en la ecuación (2.7).

$$X.Y = 2^n.\alpha + \beta + 2^{\frac{n}{2}}.(\gamma * \theta - \alpha - \beta) \tag{2.7}$$

Se busca que a nivel de hardware se describa mediante un modulo parametrizado tanto para datos con cantidad de bits pares como impares, porque la división de datos debe ser simétrica entre ambas partes.

Capítulo 3

Solución propuesta

A partir de los problemas mencionados en el capítulo 1, además de la adición de los módulos aritméticos seleccionados (*Barrel Shifter*, Multiplicador de Karatsuba) y determinar la selección del módulo para la normalización de resultado (*LZA* o *LZD*), se diseñará una nueva arquitectura de la FPU y un nuevo diseño en la máquina de estados (*FSM*) con el fin de reducir la cantidad de estados.

Estos puntos permitirán una mejora en el tiempo de respuesta que requiere la FPU para obtener una solución y la cantidad de recursos lógicos en la FPGA. Además, para la multiplicación, se plantea solucionar el problema del timing en la arquitectura para 64 bits, y que la operación pueda trabajar a una frecuencia de 100MHz.

Esta sección esta dividida de acuerdo con cada objetivo específico mencionado en el capítulo 1, mostrando los resultados respectivos y un análisis del cumplimiento de los objetivos.

3.1 Módulos de hardware

3.1.1 Descripción

Barrel Shifter

En la figura 3.1 se muestra lo que es el diagrama de bloques del módulo Barrel Shifter, que es usado para los procesos de normalización del operando menor de la suma/resta, así como para el resultado de las operaciones realizadas en el proyecto. En las entrada se tienen multiplexores para seleccionar los datos a normalizar, así como el valor calculado para la cantidad de desplazamientos del mismo.

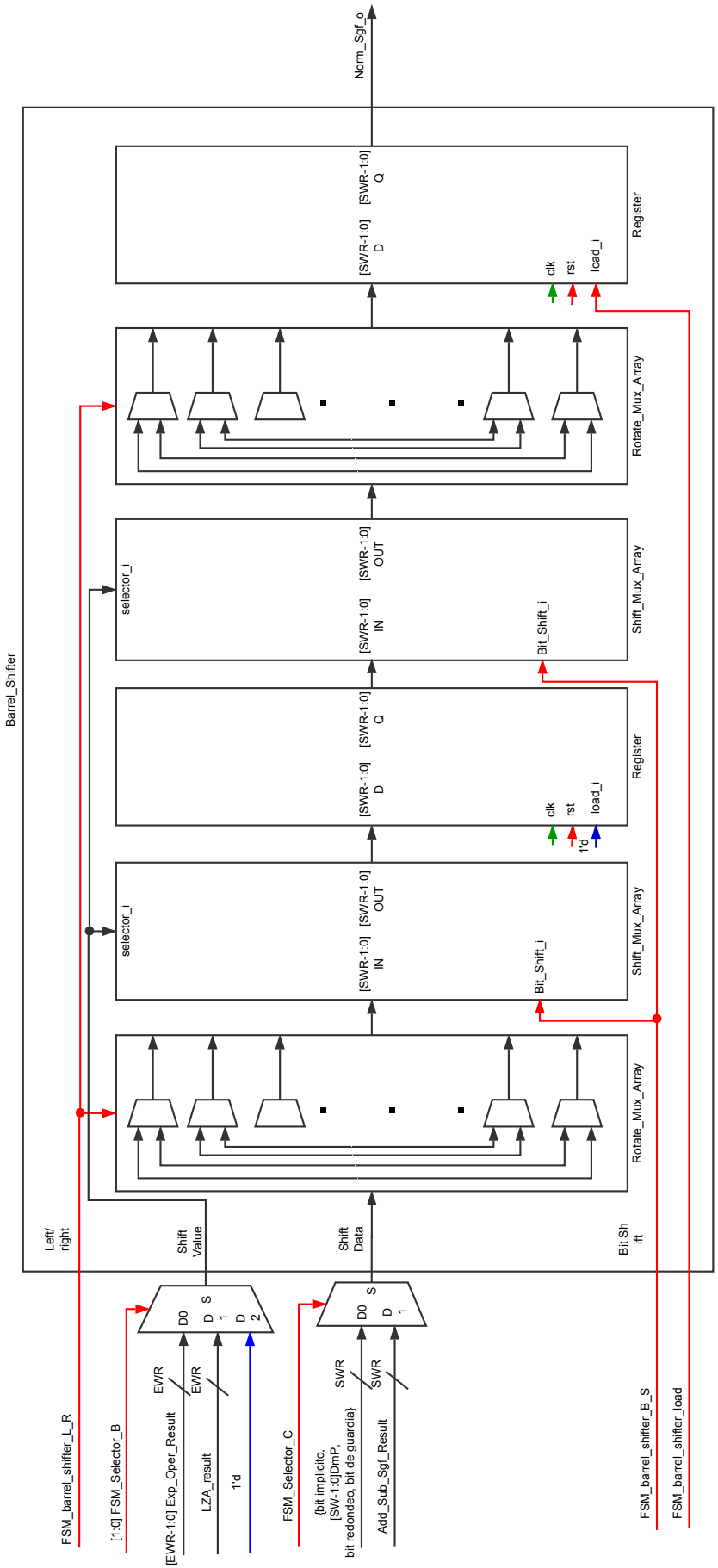

Figura 3.1: Diagrama de bloques del barrel shifter diseñado para la normalización de datos en el módulo de suma/resta de la FPU

En la tabla 3.1 se muestra una lista de las entradas/salidas para este módulo, así como una descripción de cada una de ellas.

Tabla 3.1: Listas de entradas/salidas del módulo barrel shifter diseñado para la normalización de datos en el módulo de suma/resta de la FPU

Nombre	I/O	Descripción
clk	Entrada	Señal de reloj
rst	Entrada	Señal de reset
load_i	Entrada	Señal de control para carga en registros
Shift_Value_i	Entrada	Valor binario para cantidad de desplazamientos que se le realizaran a la mantisa.
Shift_Data_i	Entrada	Dato de entrada (mantisa) el cual se realizarán los desplazamientos.
Left_Right_i	Entrada	Señal para controlar la dirección del desplazamiento.
Bit_Shift_i	Entrada	Señal de bit que se va añadiendo en los desplazamientos.
N_mant_o	Salida	Dato de salida desplazado.

El diseño en hardware consiste en el uso de la función *generate* junto a la función de iteración *for* del lenguaje de descripción de hardware *Verilog*, para generar el arreglo de multiplexores siendo la cantidad de bits de Shift_Data_i, el número de multiplexores que se genera por cada fila mientras que la cantidad de bits de Shift_Value_i, el número de niveles en cada desplazamiento.

La conexión entre multiplexores depende de la posición del dato y del nivel en que se encuentra, en donde la entrada D0 será el dato de bit que no produce desplazamiento y D1 es el dato el cual será desplazado. Se toma como límite máximo de cableado de entrada, el bit más significativo y si el cálculo del bit de conexión supera ese límite, se conecta con la señal Bit_Shift_i para ir añadiendo los bits que se deben compensar en los desplazamientos.

Al ser un diseño logarítmico, cada bit del Shift_Value_i es conectado con la entrada de selección de cada arreglo de multiplexores, según corresponda los desplazamientos que se realicen en cada nivel. Para los arreglos de multiplexores para reversión de datos se usa la señal Left_Right_i para la selección de desplazamiento izquierda/derecha.

Cálculo para la normalización de resultado

Para el *LZA* que se muestra en la figura 3.2 se plantea un diseño nuevo para los cálculos basado en el estándar IEEE 754, esto para reducir la cantidad de lógica que se requiere para casos en que no se van a cumplir en la operación suma/resta. Este módulo consiste en una lógica combinacional para determinar en que posición se cumplen alguno de los casos para encontrar el primer uno, y un codificador de prioridad para dar a la salida el valor binario para la normalización.

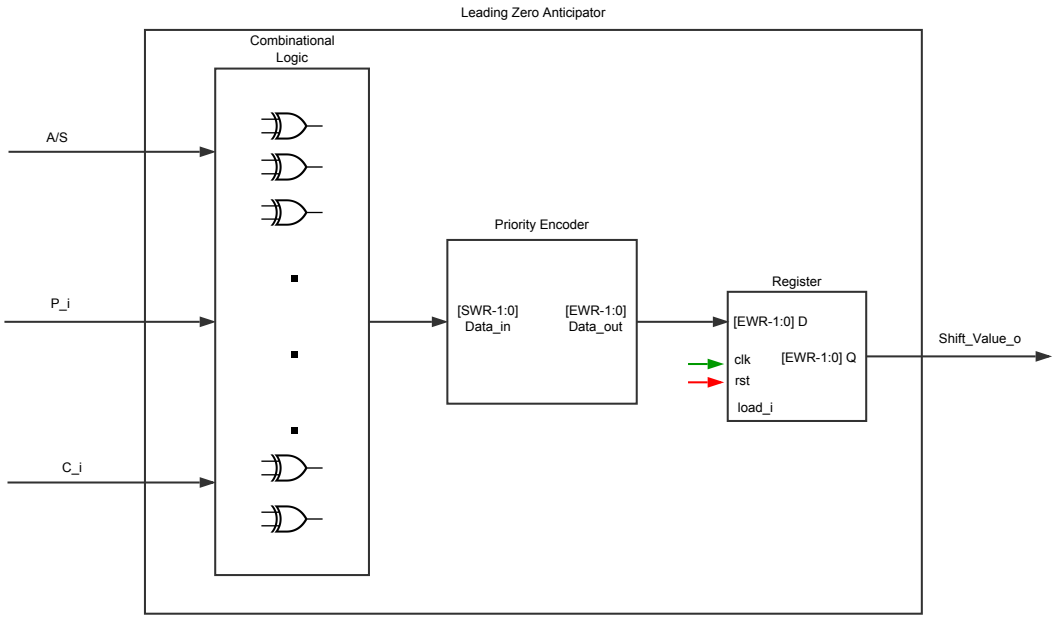

Figura 3.2: Diagrama de bloques del Anticipador de Ceros

En la tabla 3.2 se muestran el listado de entradas y salidas para este módulo.

Tabla 3.2: Listas de entradas/salidas del módulo LZA

Nombre	I/O	Descripción
clk	Entrada	Señal de reloj
rst	Entrada	Señal de reset
load_i	Entrada	Señal de control para carga en registros
P_i	Entrada	Dato propagate del sumador
C_i	Entrada	Dato carry del sumador
A_S_i	Entrada	Señal de control de la operación a realizar (C[0])
Shift_Value_o	Salida	Dato de salida para la normalización.

El algoritmo pensado en la FPU, para una operación $A \pm B$, no da como resultado valores en complemento a dos, por lo que se reducen la cantidad de posibilidades para encontrar el primer uno. Se hicieron diferentes casos de sumas y restas en los que el resultado fuese positivo, con ello se tienen los siguientes casos en los que se puede encontrar el primer uno en la posición i del resultado:

- $A + B; A[i] \neq B[i]; C[i] = 0$
- $A + B; A[i] = B[i] = 0; C[i] = 1$
- $A - B; A[i] = B[i] = 0; C[i] = 1$
- $A - B; A[i] = B[i] = 1; C[i] = 1$

en donde C[i] es el *carry* resultante de la suma $A[i-1] \pm B[i-1]$.

Con estos datos se calculó mediante una tabla de verdad, el circuito combinacional resultante. El resultado es una XNOR entre C[i] y la señal P (*propagate*) que es una señal propia del sumador, dando como resultado la ecuación 3.1 .

$$\overline{\mathrm{Y[i]}} = \overline{(\mathrm{C[i]} \oplus (\mathrm{A[i]} \oplus \mathrm{B[i]}))} \tag{3.1}$$

Existen otros casos en donde para una posición k puede tener un uno en el resultado, pero para esos casos también se da que hay un uno, al menos, en la posición siguiente más significativa por lo que ya k deja de ser el bit más significativo. Para ello se utiliza el codificador, el cual da prioridad desde el bit más significativo y da el valor i en binario, para normalizar tanto en el exponente como en la mantisa y así eliminando la parte iterativa de los desplazamientos.

Multiplicación por algoritmo de Karatsuba

En la figura 3.3 se muestra el diagrama de bloques del módulo de multiplicación mediante el algoritmo de Karatsuba y la tabla 3.5 con la descripción de entradas/salidas.

El módulo consiste en la descripción en hardware de la ecuación (2.7.) en donde se calculan las expresiones α, β ,γ y θ a partir de los datos de entrada (Data_A_i y Data_B_i).

Luego se tienen los módulos para el cálculo de la expresión que requiere un desplazamiento a la derecha de *(n/2)* bits ($\gamma * \theta - \alpha - \beta$). Ya que el resultado de esta operación no debe representarse en complemento a dos, se deben utilizar restadores en serie, calculando por pasos la expresión.

Ya que el ancho del dato de las expresiones es constante, para la suma de la expresión final se usará un solo sumador de ancho (*2*n*), en donde el primer sumando será un bus de las expresiones α y β mientras que el segundo sumando será la expresión desplazada (*n/2*) bits (figura 3.4) donde m es un valor que varía dependiendo de si n es par o impar.

Esta suma se puede hacer ya que en la ecuación (2.7.), la expresión $2^n.\alpha + \beta$ resulta en un dato de ancho (*2*n*) bits, el cual se compone del dato α desplazado n bits a la derecha (*[2n-1:n]*) y el dato β (*[n-1:0]*).

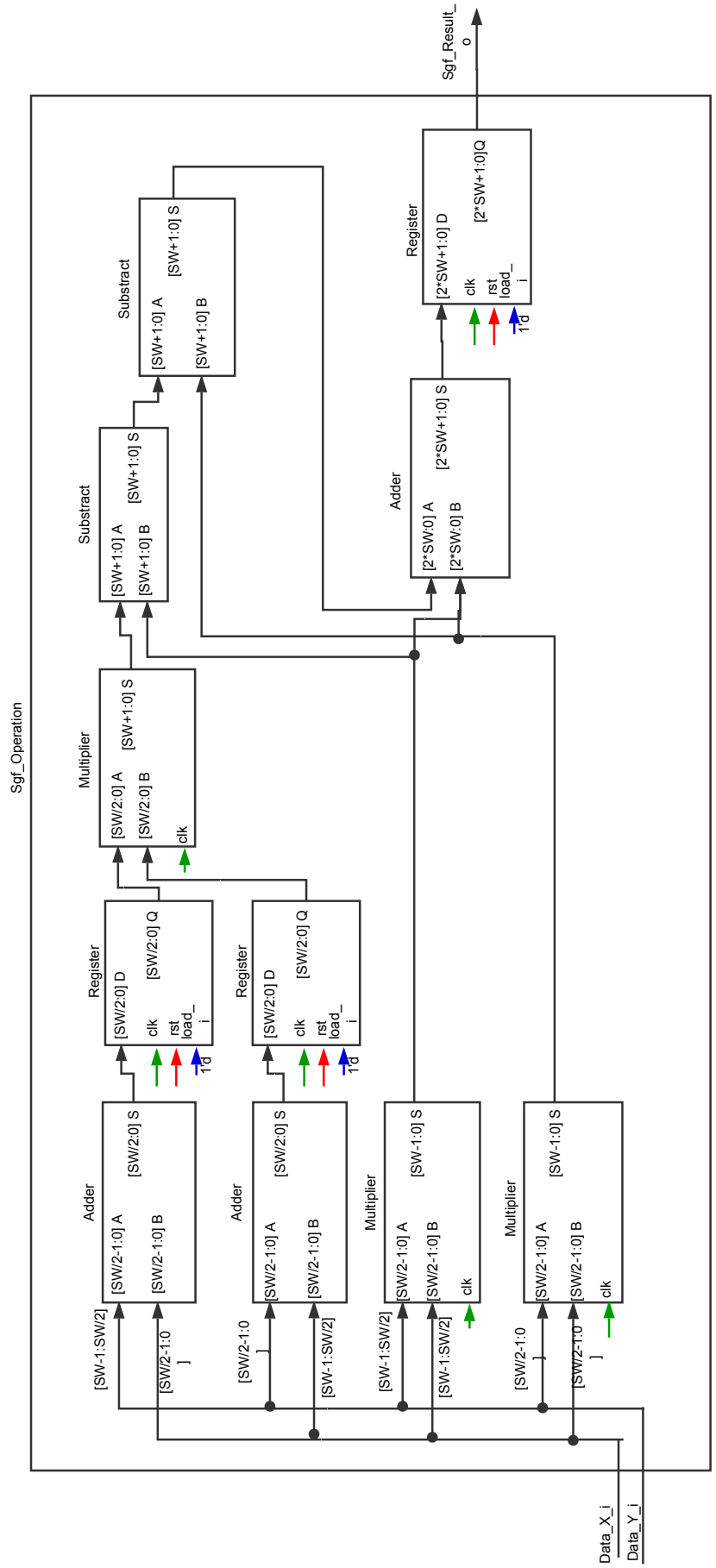

Figura 3.3: Diagrama de bloques del multiplicador de Karatsuba, el cual describe en hardware la ecuación (2.7.)

Tabla 3.3: Listas de entradas/salidas del módulo Sgf_Operation

Nombre	I/O	Descripción
clk	Entrada	Señal de reloj
rst	Entrada	Señal de reset
load_i	Entrada	Señal de control para carga en registros
Data_A_i	Entrada	Operando X
Data_B_i	Entrada	Operando Y
Sgf_Result_o	Salida	Dato de salida de resultado.

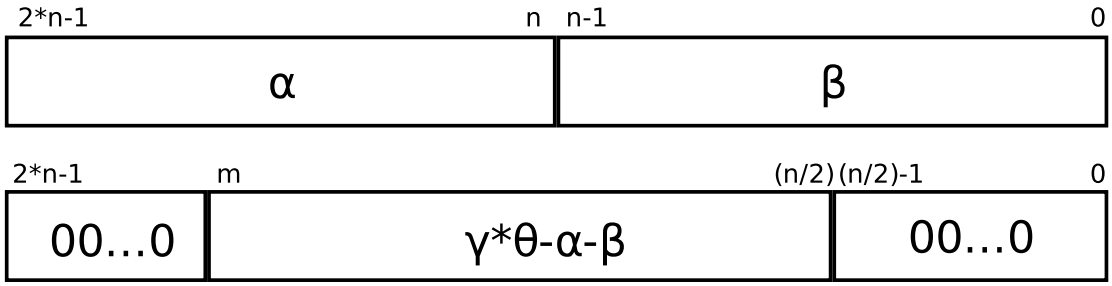

Figura 3.4: Representación de la trama de datos de los operandos para la suma final del multiplicador de Karatsuba

3.1.2 Resultados

Normalizador

El barrel shifter diseñado debe manejar el dato de la mantisa, así como el bit implícito y los bits de redondeo y guardia, mientras que la cantidad de bits para la cantidad de desplazamientos debe representar el valor de la cantidad de bits del valor de entrada. En la tabla 3.4 muestran los parámetros del barrel shifter para precisión simple y doble.

Tabla 3.4: Parámetros de entradas/salidas para el módulo *Barrel shifter*

Arquitectura	32 bits	64 bits
Shift_Value_i	5 bits	6 bits
Shift_Data_i	26 bits	55 bits
N_mant_o	26 bits	55 bits

El tiempo que se requiere para obtener un resultado es de un ciclo de reloj para precisión simple y dos para precisión doble. Ello debido a errores de temporizado que no permiten alcanzar los 100 MHz en la ruta crítica que aparece en esta estructura, al añadirse el arreglo de multiplexores extra. La ruta se optimiza entonces añadiendo un registro de segmentación entre los multiplexores.

Se diseñó el LZA pensando en utilizar la lógica del sumador/restador para los cálculos y reducir el área, pero de acuerdo con la ecuación (3.1) , el resultado es igual al resultado de la operación del módulo suma/resta, por lo que la idea de reducir los casos da como resultado la utilización de un LZD (figura 3.5) conectado a la salida del sumador/restador.

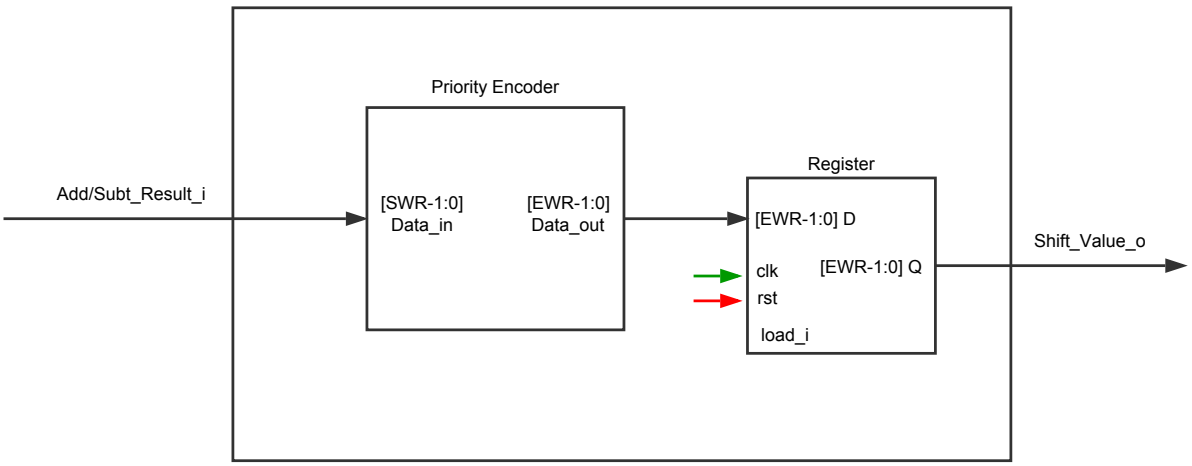

Figura 3.5: Módulo LZD utilizado para el cálculo de la cantidad de desplazamientos que requiere la mantisa de resultado para ser normalizada en una operación suma/resta

Multiplicación

En la tabla 3.5 se muestran los parámetros para el multiplicador de Karatsuba utilizado en la solución, así como los recursos que requiere el mismo.

Tabla 3.5: Parámetros de entradas/salidas del módulo *Sgf Operation*

Arquitectura	32 bits	64 bits
Data_A/B_i	24 bits	53 bits
Sgf_Result_o	48 bits	106 bits

Este módulo requiere de dos ciclos de reloj en un cálculo para precisión simple, lo que resulta aceptable ya que este cálculo se realiza en paralelo con el cálculo del exponente que también requiere de 2 ciclos de reloj por lo cual no produce retrasos en la solución de la operación.

Para precisión doble, el *delay* juega un papel importante en el cálculo de $\gamma * \theta$, en donde la multiplicación es entre valores de 27 bits, ya que la ruta crítica en toda la expresión no cumple con el *timing* para 100 MHz. Esto hace que se le deba añadir registros de segmentación entre los sumadores de γ y θ, y el multiplicador para cumplir con el *timing*, creándose un requerimiento de latencia de tres ciclos de reloj para obtener un resultado.

3.1.3 Análisis de resultados

El *barrel shifter* diseñado, con la configuración de reversión de dato, reduce bastante el área de uso con la desventaja del *delay* que hay entre el arreglo de multiplexores que, junto con los multiplexores de entrada para seleccionar el dato a normalizar, no logra que se realice toda la operación en un ciclo de reloj en la arquitectura de 64 bits. Esto podría mejorarse en una posible futura implementación VLSI al optimizar estos multiplexores a nivel de transistores de compuertas de paso, lo que podría también reducir el área.

El módulo que permite una verdadera mejora para el cálculo de desplazamientos en el resultado de la suma/resta en el estándar IEEE 754 es el LZD, ya que los casos mencionados para el resultado en la mantisa solamente se necesita de una detección de la cantidad de ceros que hay antes del primer bit significativo del resultado. Aunque este módulo agrega un ciclo de reloj extra para su cálculo,resulta mucho más eficiente que la versión anterior que requería un algoritmo iterativo [12]. Se explicará más adelante en el diseño de la máquina de estados, que este cálculo no se vuelve crítico ya que se realizan otras operaciones en paralelo con él.

El uso del algoritmo de Karatsuba a nivel de hardware, mejora el *timing* en el cálculo de la multiplicación entre operandos de datos grandes. Se deberá tomar en cuenta que conforme aumenta la cantidad de bits que se utilizan en la multiplicación, aumentará el *slack* del módulo. Por lo que si se quiere usar arquitecturas mayores a 64 bits, se deberá aplicar el mismo algoritmo de Karatsuba para la multiplicaciones parciales de las expresiones.

3.2 Reducción de recursos lógicos

En esta sección se hará un listado de las modificaciones que se tuvieron que realizar a nivel de hardware, además de la adición de los nuevos módulos, con el fin de reducir el uso de recursos lógicos y estructurar la arquitectura de la FSM para mejorar el tiempo de respuesta en las operaciones.

3.2.1 Suma/Resta

Arquitectura

En la figura 3.6 se muestra el diagrama de bloques para la operación suma/resta diseñada y solución de la operación. Cada uno de los módulos usados se explican en la tabla 3.6

El primer cambio que se implementó fue reducir la cantidad de módulos repetidos que eran usados en diferentes etapas de la arquitectura y que poseen las mismas características. Esto también aumentaba la cantidad de registros para almacenar el resultado en cada etapa. Se menciona en la siguiente lista los módulos encontrados:

- Cálculos en el exponente: se usaban un par sumador/restador que realizaban los procesos de diferencia entre exponentes para la normalización del menor operando, normalización del exponente post-operación en mantisa y normalización del exponente post-redondeo. Lo anterior implicaba un total de tres sumadores y dos restadores.
- Cálculos de mantisa: se usaban un par sumador/restador que realizaban los procesos de la suma entre las mantisas de los operandos y otro sumador para adición para el proceso de redondeo.
- Módulos de desplazamiento: se usaba uno combinacional para la normalización del operando menor y un desplazador secuencial para la normalización de resultado.
- Comparadores de magnitud repetidos para los resultados de *overflow*. Se usaban dos, cada uno en las etapas de verificación post-operación en mantisa y en post-redondeo.

Para optimizar estos recursos, se utilizaron multiplexores en las entradas de los módulos que se requieren para más de una etapa de la resolución de la operación aritmética, los cuales direccionan el flujo de datos y que son controlados por la FSM que fue diseñada con respecto al algoritmo original de suma y resta y que se puede resumir en la figura 3.7.

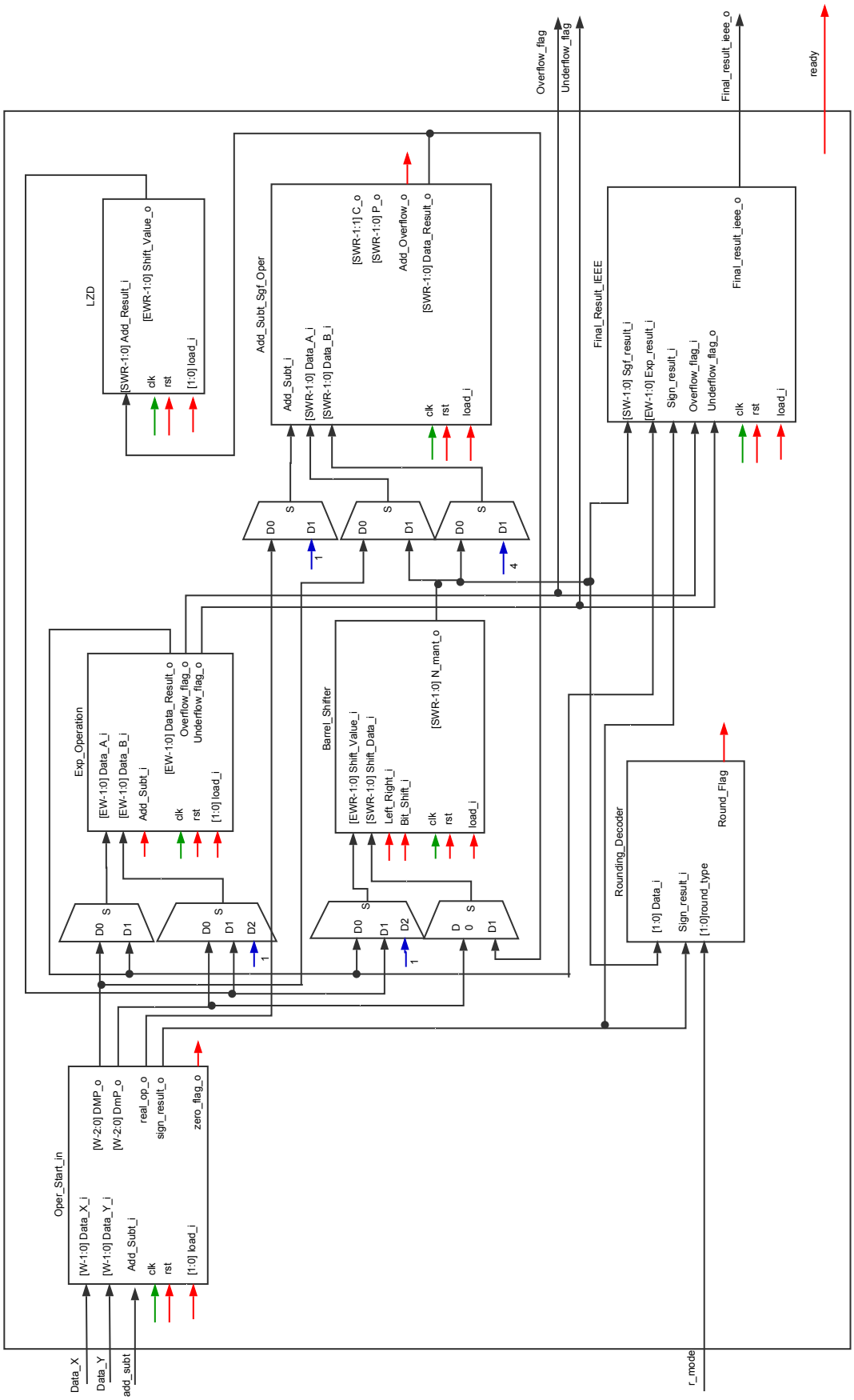

Figura 3.6: Diagrama de bloques para la operación suma/resta

Tabla 3.6: Descripción de los módulos que conforman la operación suma/resta

Nombre	Descripción
Oper_start_i (Figura A.2.)	• Carga de datos a la entrada y selección del valor mayor/menor • Calculo de la operación real a realizar según sea la señal de operación de entrada y los signos de los valores de entrada. • Cálculo de bandera de resultado *zero*. • Cálculo del bit de signo de resultado
Exp_Operation (Figura A.3.)	Se calcula: • La diferencia entre los exponentes de los operandos. • La compensación en el exponente de resultado para cada normalización de la mantisa. También se compara el exponente resultado con los límites máximo y mínimo para los casos de *overflow/underflow*.
Barrel_shifter (Figura 3.1)	Sección del normalizador para los desplazamiento de los bits de datos para: • La mantisa del bit menos significativo. • La mantisa del resultado.
LZD (Figura 3.5)	Segmento del normalizador para calcular la cantidad de desplazamientos que requiere el dato de resultado en caso de requerir desplazamientos a la izquierda.
Add_Subt_sgf (Figura A.4.)	Es el módulo donde se realizan las operaciones de suma/resta en la mantisa: • Entre los operandos de entrada y la señal de control para seleccionar la operación a realizar es la operación real calculado en Oper_start_i. • Para los casos de redondeo, que requiere una suma entre la mantisa de resultado + uno en el bit menos significativo.
Round_sgf_Dec	Decodificador que determina los casos de redondeo existente en el estándar IEEE 754
Final_Result_IEEE (Figura A.5.)	Módulo que selecciona los resultados de la operación según se la excepción encontrada: • *Zero* • *Overflow/Underflow* • Resultado representable.

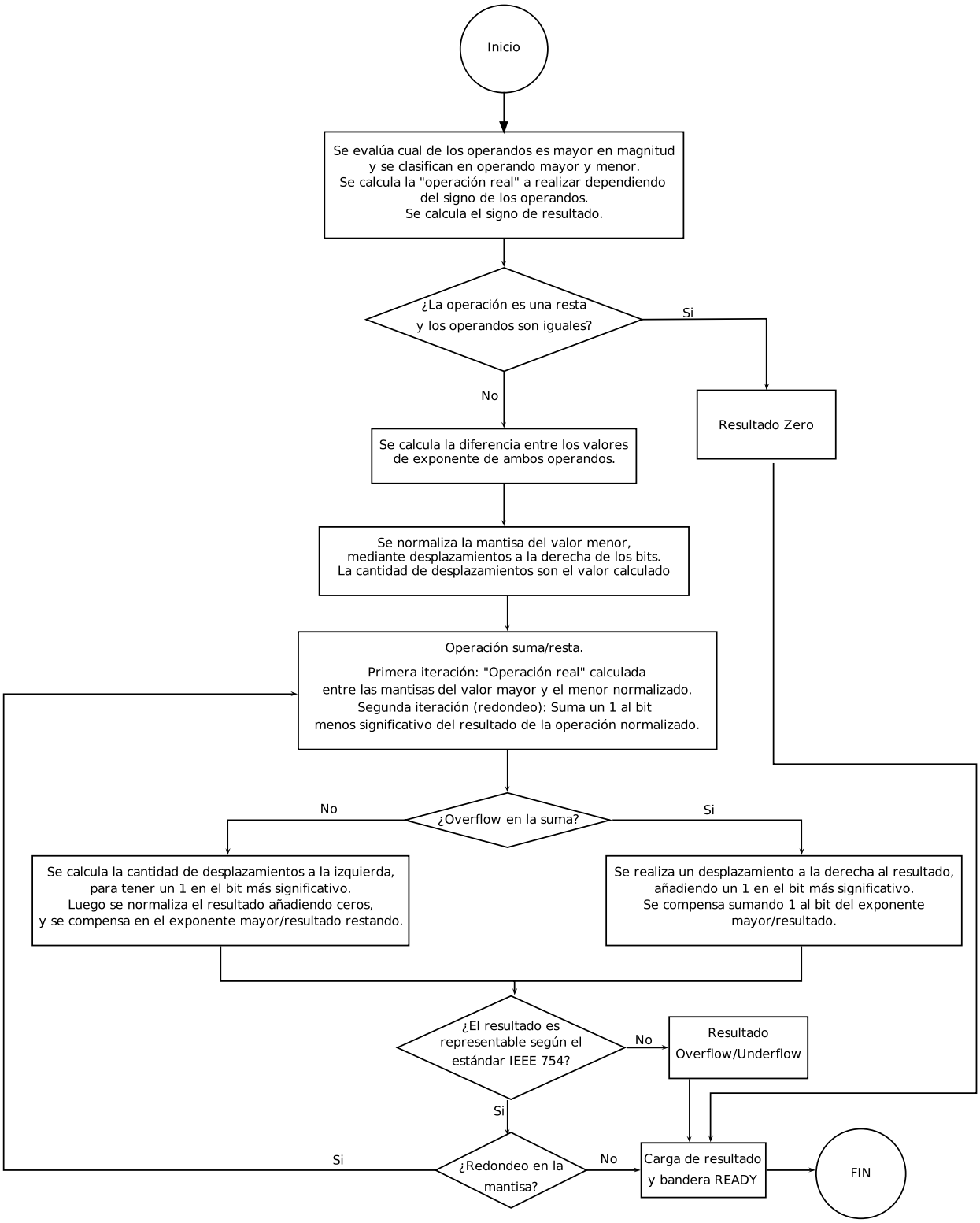

Figura 3.7: Diagrama de estados de la FSM para la suma/resta

Se modificó también el manejo de los signos de los operandos para calcular del signo de resultado y la operación real que se debe ejecutar dependiendo de los casos de inversión de signo en la resta. Esto para tener menos señales para evaluar en los decodificadores de verificación de resultado *Zero* y de operación en mantisa.

El módulo de salida del resultado también es modificado con la finalidad de que, mediante las banderas de *overflow/underflow*, reducir el tamaño de los multiplexores para el acomodo del resultado de salida. Esto ya que el único bit que diferencia el resultado *overflow* del *underflow* es el bit de signo, mientras que para el exponente y mantisa son iguales. A diferencia de la primera versión donde los se usan dos multiplexores de n bits de entrada (32 para precisión simple y 64 para precisión doble), en esta versión se utilizan multiplexores para cada sección de la representación del resultado.

Máquina de estados

Con la modificación de la arquitectura también se debió cambiar la FSM para el manejo del flujo de dato de acuerdo con el algoritmo planteado en la figura 3.7 la cual se explicarán los pasos a nivel de hardware dentro de la arquitectura. Con el cambio de arquitectura, la FSM se ha reducido de 47 a 15 estados (figura A.6.) los cuales se explican en el siguiente listado:

1. `Start`: se activa la señal reset para los módulos internos de la FPU en la suma y pasa al siguiente estado cuando recibe la señal beg_FSM.

2. `Load_Oper`: habilita la carga en los registros de entrada para los operandos y la señal op para la selección suma/resta. En este estado se ejecuta el módulo Oper_start_i.

3. `Zero_info_state`: habilita la carga en los registros de salida del módulo Oper_start_i y se evalúa la bandera Zero_Flag. En caso de cumplirse la condición se salta al estado `Ready_flag`.

4. `Load_diff_exp`: habilita la carga en los registros de salida del módulo Exp_Operation para el resultado del sumador/restador. En este estado se calcula la diferencia entre exponentes de los operandos para normalizar en operando de menor magnitud.

5. `Extra1_64`: estado extra de pipeline del módulo Barrel_shifter requerido en 64 bits.

6. `Norm_sgf_first`: segundo estado requerido para calcular el resultado del Barrel_shifter. Habilita los registros de salida del mismo, y evalúa si se esta realizando la normalización del operando de menor magnitud en donde realiza desplazamientos a la derecha para pasar al estado `Add_Subt`; ó si la normalización a realizar es para el resultado de la operación suma/resta en la mantisa, en este caso evalúa si hay un *overflow* en la suma (señal Add_Overflow). Si se cumple esta condición se realiza un desplazamiento a la derecha añadiendo un uno, caso contrario se desplaza a la izquierda añadiendo ceros según el valor de salida de la LZD. Finalizada la normalización del resultado se pasa la estado `round_sgf`.

7. `Add_Subt`: se habilitan los registros de salida del módulo Add_Subt_sgf y se selecciona la entrada D1 del multiplexor para la entrada Shift_Data_i del Barrel_shifter. Esta señal de control también es usada para la evaluación del dato a normalizar para el estado `Norm_sgf_first`.

8. `Overflow_add`: se habilitan los registros de salida del módulo LZD y se evalúa la señal Add_Overflow para seleccionar la entrada del multiplexor de Shift_Data_i del módulo Barrel_shifter así como para los multiplexores en las entradas de Exp_operation para la normalización en el exponente. Hace un salto hacia el estado `Extra1_64`.

9. `Round_sgf`: evalúa la condición de la señal Round_flag . En caso de cumplirse, se seleccionan las entradas D1 para los multiplexores de entrada del módulo Add_Subt_sgf para realizar la operación de redondeo. En caso contrario se hace un salto al estado `load_final_result`.

10. `Add_Subt_r`: estado homólogo a `Add_Subt` para el proceso de redondeo.

11. `Overflow_add_r`: estado homólogo a `Overflow_add` para el proceso de redondeo. Se diferencia en que en caso de no cumplirse la condición, no se ejecuta la normalización y solamente deje pasar los datos de exponente y mantisa resultado.

12. `Extra2_64`: estado homólogo a `Extra1_64` para el proceso de redondeo.

13. `Norm_sgf_r`: estado homólogo a `Norm_sgf_first` para el proceso de redondeo.

14. `load_final_result`: habilita la carga de los registros para el módulo Final_Result_IEEE. El dato de carga depende de las banderas de *Overflow* y *Underflow* que direccionan los datos de los multiplexores internos.

15. `Ready_flag`: Habilita la bandera Ready_FSM y no cambia de estado hasta que se active la señal ack_FSM el cual indica que el dato se ha obtenido con éxito. Se hace un salto al estado `Start` esperando otra ejecución de la operación.

Resultados

Se realizaron las simulaciones necesarias para la comprobación del funcionamiento de la Operación Suma/Resta en contraste con la referencia dorada calculada con el software Octave y con la primera versión de la operación para 1024 valores distintos tanto en precisión simple y doble (figuras 3.8 y 3.9 donde muestran los primeros 500 resultados) en donde la línea roja representa el porcentaje error contra la primera versión y los puntos azules representan el error respecto al valor teórico.

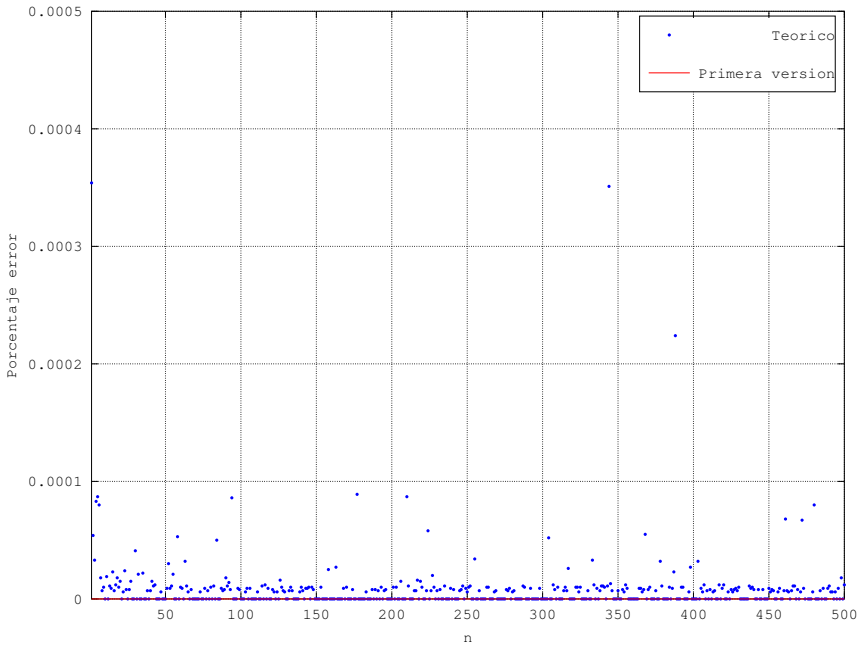

Figura 3.8: Porcentajes de error de 500 valores aleatorios para una simulación post-implementación de Suma/Resta en arquitectura de 32 bits

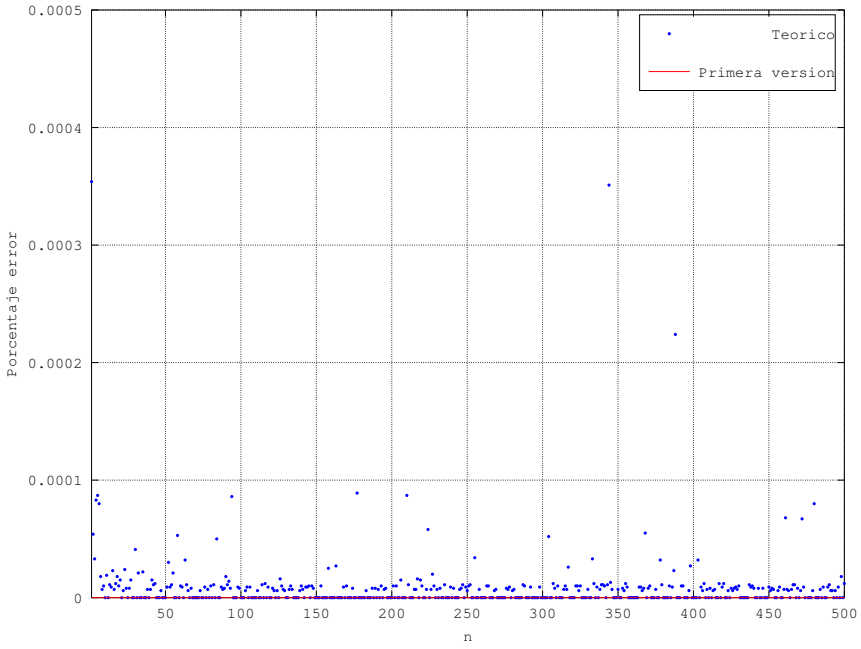

Figura 3.9: Porcentajes de error de 500 valores aleatorios para una simulación post-implementación de Suma/Resta en arquitectura de 32 bits

Las simulaciones post-implementación con el programa suite Vivado de Xilinx dieron los resultados de uso de recursos utilizados del módulo de suma/resta (Tabla 3.8) así como el consumo de potencia que requiere la unidad (Tabla 3.9) y el tiempo de ejecución que le toma a la unidad calcular 1024 valores aleatorios (Tabla 3.7).

Tabla 3.7: Tiempo de ejecución de la operación suma/resta para 1024 valores en arquitectura de 32 y 64 bits

	Tiempo de ejecución μs	
Arquitectura	32 bits	64 bits
Primera versión	463,975	455,695
Versión diseñada	174,025	174,025
Δ	289.95	281.67

Tabla 3.8: Comparación de uso de recursos lógicos en *post-implementation* de la FPGA, entre las versiones de la FPU en la operación suma/resta para arquitecturas de 32 y 64 bits

Diseño	Recuso lógico	Arquitectura 32 bits	Arquitectura 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

Tabla 3.9: Consumo de potencia del módulo Suma/Resta entre versiones implementadas en una FPGA (en mW) para arquitecturas de 32 y 64 bits

	Primera Versión		Solución	
On-chip	Arquitectura 32 bits	Arquitectura 64 bits	Arquitectura 32 bits	Arquitectura 64 bits
Clock	1	3	2	3
Logic	1	2	2	5
Signals	2	4	3	7
I/0	3	5	7	14
Static	91	91	91	91
Total	98	106	105	119

3.2.2 Multiplicación

Arquitectura

La figura 3.10 muestra la arquitectura creada para realizar la operación de multiplicación. Esta tuvo menores modificaciones con respecto a la suma por la complejidad del algoritmo, donde los pasos para realizar una multiplicación no requieren de una recursividad en el flujo de datos con la excepción del calculo del exponente resultado y la normalización de la mantisa.

En solución que se plantea en la multiplicación también se reducen los módulos para ser usados en diferentes procesos, los cuales se describen en la tabla 3.10. Se diseñó de tal forma que el módulo del exponente solo se tenga un par sumador/restador para que en dos ciclos de reloj se calcule el resultado del exponente donde: en el primer ciclo se calcula la suma entre los exponentes de los operandos y en el segundo ciclo se calcule la resta del *bias*. Esto aprovechando que la multiplicación entre mantisas está segmentado por registros y que va a ser más lenta que el cálculo del exponente.

La normalización de la mantisa en casos de *overflow*, ya sea en la operación de multiplicación o para el proceso de redondeo, se ha reducido en un solo arreglo de multiplexores sin los reversores de datos, ya que para cualquiera de los dos procesos solamente requiere de un desplazamiento a la derecha añadiendo un uno.

Se utilizó el mismo modelo para seleccionar el resultado de salida con respecto a los casos de *overflow* y *underflow* en la operación que fue usado en la suma/resta, para reducir la cantidad de datos que se deben seleccionar para obtener un resultado.

Se le ha agregado el módulo de multiplicación de Karatsuba, sustituyendo al modelo de multiplicación original que es el módulo con la ruta más crítica en *delay*, para resolver el problema de *timing* para 100MHz.

Se ha aprovechado la independencia que hay en los cálculos de la Zero_Flag y los resultados del algoritmo para el exponente, la mantisa y el bit de signo, para que se ejecuten en paralelo, sin la necesidad de esperar a que se ejecuten de manera secuencial como lo realizaba la primera versión.

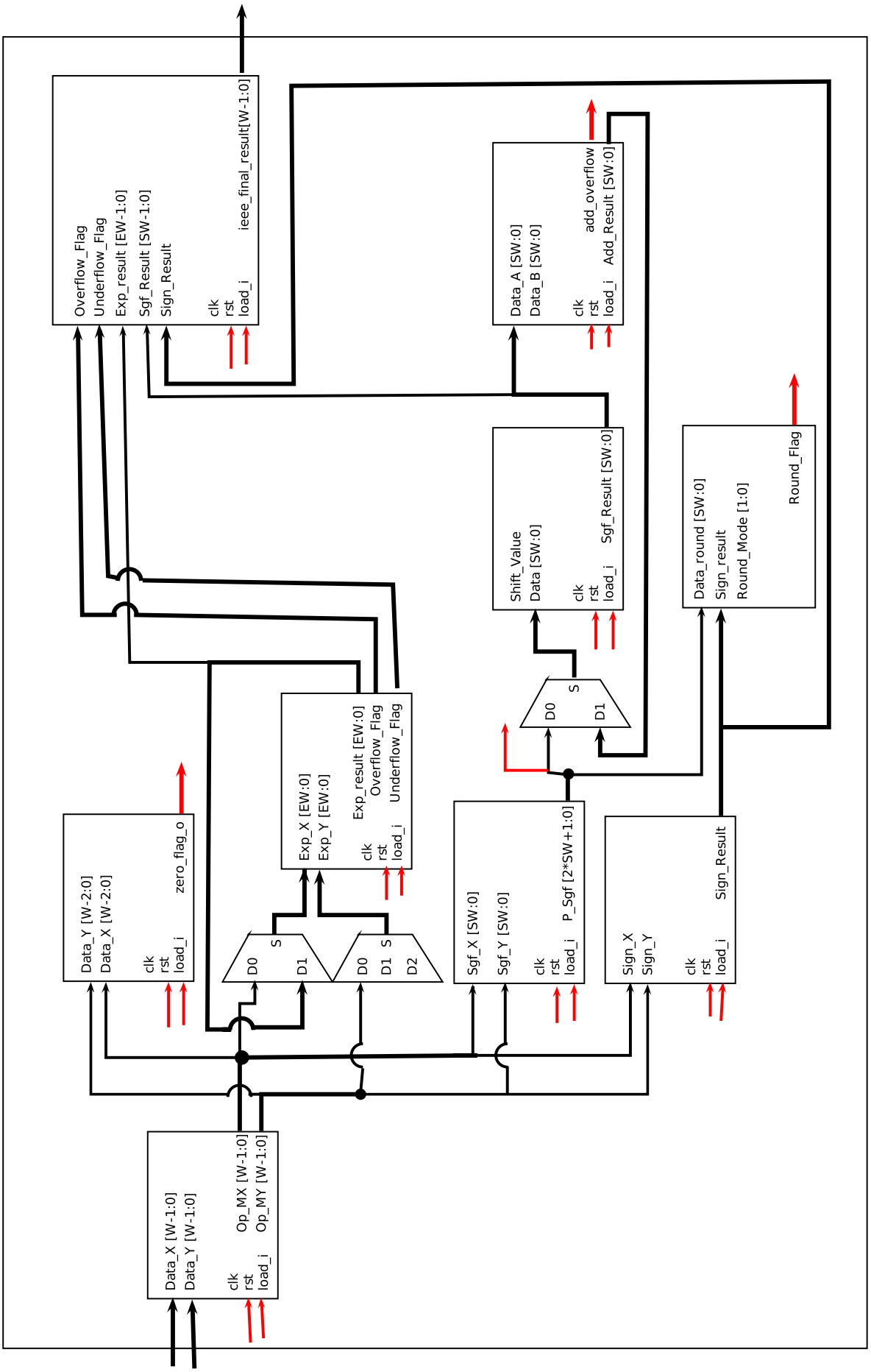

Figura 3.10: Diagrama de bloques para la operación multiplicación

Tabla 3.10: Descripción de los módulos que conforman la operación Multiplicación

Nombre	Descripción
Operands_load_reg (Figura B.2.)	Carga de datos a la entrada
Zero_result_detect (Figura B.3.)	Cálculo de bandera de resultado zero.
Exp_Operation_m (Figura B.4.)	Se calcula: • El exponente de resultado.(Requiere de 2 ciclos de reloj) • La compensación en el exponente de resultado para cada normalización de la mantisa. También se compara el exponente resultado con los límites máximo y mínimo para los casos de *overflow* y *underflow*.
Sign_Operation_m	Se calcula el signo de resultado.
Sgf_Operation_m (Figura 3.3)	Se calcula la multiplicación entre las mantisas de los operandos mediante el algoritmo de Karatsuba.
Barrel_shifter	Normalizador para los desplazamientos de los bits de datos en la mantisa del resultado tanto luego de la multiplicación como en el redondeo.
Adder_round	Sumador para añadir un uno en el bit menos significativo de la mantisa para los casos de redondeo.
Round_sgf_Dec	Decodificador que determina los casos de redondeo existentes en el estándar IEEE 754
Final_Result_IEEE (Figura A.5.)	Módulo que selecciona los resultados de la operación según se la excepción encontrada: • *Zero* • *Overflow/Underflow* • Resultado representable.

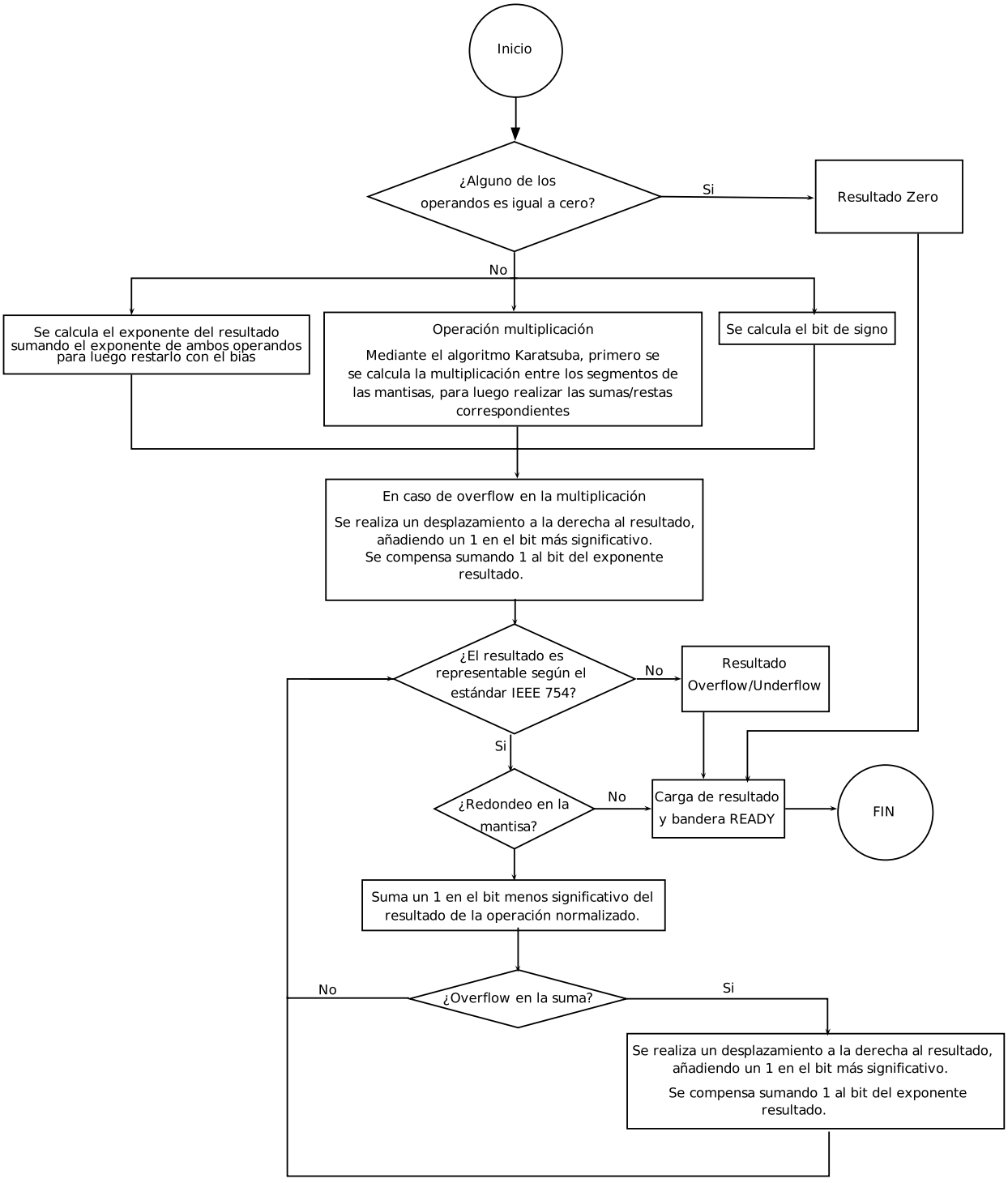

Figura 3.11: Diagrama de estados de la FSM para la multiplicación

Máquina de estados

La FSM para modulo de multiplicación de redujo de 46 a 13 estados ya que la arquitectura nueva ejecuta procesos independientes en paralelo con respecto al modelo original y la complejidad del algoritmo (figura 3.11) hace que no requiera de muchos saltos entre los estados, lo que la hace más lineal. El diagrama del mismo se puede se encuentra en el anexo 2 (Figura B.5.) y se enuncia en el siguiente listado el manejo de las señales de control de cada estado:

1. `Start`: es activada la señal reset para los módulos internos de la FPU y pasa al siguiente estado cuando recibe la señal beg_FSM.

2. `Load_Oper`: habilita la carga en los registros de entrada para los operandos.

3. `Extra64_1`: estado transitorio extra de pipeline del módulo Sgf_Operation_m requerido en 64 bits.

4. `Add_exp`: se evalúa la bandera Zero_Flag, en caso de cumplirse la condición se salta al estado `Ready_flag`. Además se realiza la suma entre los exponentes de los operandos y es el segundo ciclo de reloj para el módulo Sgf_Operation_m. se seleccionan la entrada D1 de los multiplexores para el módulo Exp_Operation_m para calcular la resta del *bias* en el siguiente estado. También se carga el registro con la bandera para la condición de *Underflow*.

5. `Subt_bias`: se realiza la resta entre el resultado anterior del exponente y el *bias* para obtener el exponente resultado, así como la carga en los registros de salida para el módulo Exp_Operation_m. Se carga el registro con la bandera para la condición de *Overflow*.

6. `Mult_Overf`: estado donde se evalúa si el resultado de la multiplicación requiere de normalización en la mantisa mediante el bit más significativo del resultado de la multiplicación. En caso de cumplirse, se selecciona la entrada D2 del multiplexor de la entrada Exp_Y del módulo Exp_Operation_m para la compensación en el exponente.

7. `Mult_norm`: se realiza el proceso de normalización correspondiente en la mantisa y la compensación del exponente.

8. `Mult_no_norm`: no se realiza normalización. Solamente carga en los registros del módulo *Barrel_shifter*, el resultado obtenido de la multiplicación en las mantisas.

9. `Round_case`: evalúa la condición de la señal Round_flag para realizar el proceso de redondeo. En caso de cumplirse, se selecciona la entrada D2 del multiplexor de la entrada Exp_Y del módulo Exp_Operation_m para la compensación en el exponente en caso de que se requiera normalización por un *overflow* en el proceso de redondeo. En caso contrario se hace un salto al estado `final_load`.

10. `Adder_round`: se realiza la suma correspondiente para el proceso de redondeo y se selecciona el resultado (D1) en el multiplexor del módulo Barrel_shifter para la normalización.

11. `Round_norm`: estado homólogo a `Mult_norm` para el proceso de redondeo. Se diferencia en que en caso de no cumplirse la condición, no se ejecute la normalización y solamente deje pasar los datos de exponente y mantisa.

12. `final_load`: habilita la carga de los registros para el módulo Final_Result_IEEE. El dato de carga depende de las banderas de *Overflow* y *Underflow* que direccionan los datos en los multiplexores internos.

13. `Ready_flag`: Habilita la bandera Ready_FSM y no cambia de estado hasta que se active la señal ack_FSM el cual indica que el dato se ha obtenido con éxito. Se hace un salto al estado `Start` esperando otra ejecución de la operación.

Resultados

Al igual que en la suma, se realiza la comprobación correspondiente de la arquitectura diseñada comparado con el valor teórico y la versión anterior para 32 y 64 bits (figuras 3.12 y 3.13 donde muestran los primeros 500 resultados) en donde la línea roja representa el porcentaje error contra la primera versión y los puntos azules representan el error respecto al valor teórico.

Se muestra en las tablas 3.12 y 3.13 los recursos utilizados por el módulo diseñado en contraste con el modelo original. Así como el tiempo de duración de la FPU para calcular 1024 valores aleatorios en la tabla 3.11.

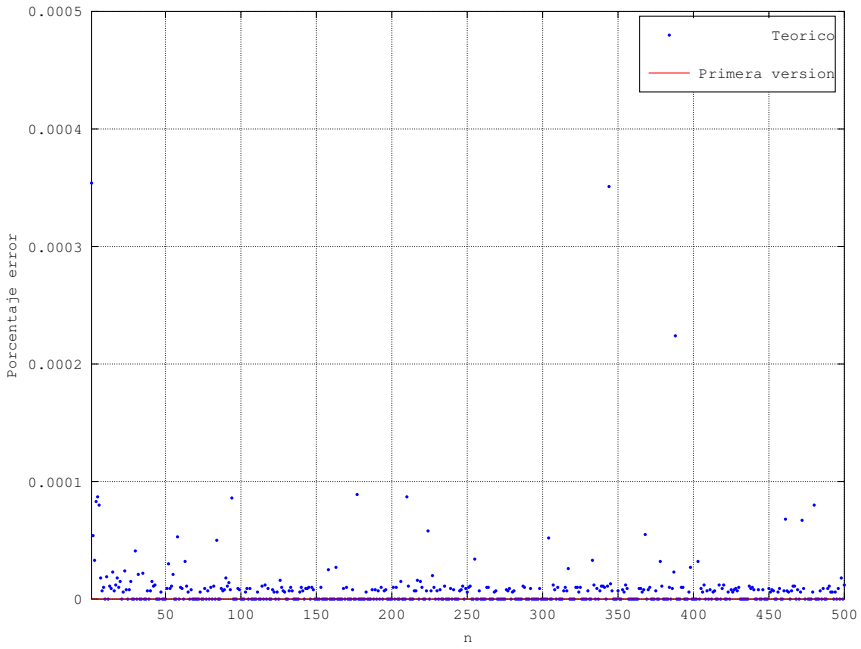

Figura 3.12: Porcentajes de error de 500 valores aleatorios para una simulación post-implementación de Multiplicación en arquitectura de 32 bits

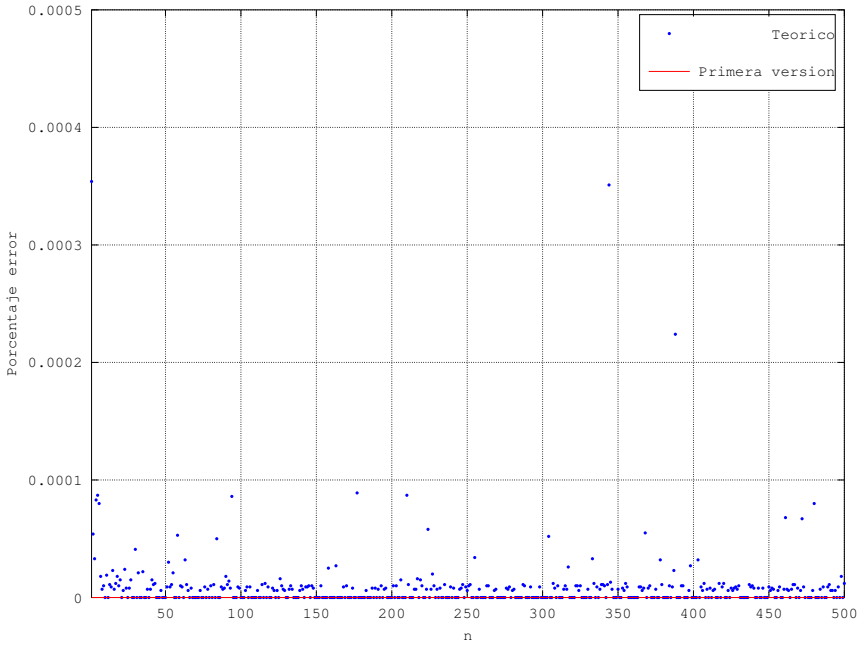

Figura 3.13: Porcentajes de error de 500 valores aleatorios para una simulación post-implementación de Multiplicación en arquitectura de 32 bits

Tabla 3.11: Tiempo de ejecución de la operación multiplicación para 1024 valores en arquitectura de 32 y 64 bits

	Tiempo de ejecución μs	
Arquitectura	32 bits	64 bits
Primera versión	470.695	470.695
Versión diseñada	143.345	143.345
Δ	327.350	327.350

Tabla 3.12: Comparación de uso de recursos lógicos en la FPGA, entre las versiones de la FPU en la operación multiplicación en arquitecturas de 32 y 64 bits

Diseño	Recuso lógico	Arquitectura 32 bits	Arquitectura 64 bits
Primera versión	LUT	173	358
	FF	297	594
	DSP	2	9
Versión diseñada	LUT	170	516
	FF	213	530
	DSP	4	12

Tabla 3.13: Consumo de potencia del módulo Multiplicación entre versiones implementadas en una FPGA (en mW) para arquitecturas de 32 y 64 bits

	Primera Versión		Solución	
On-chip	Arquitectura 32 bits	Arquitectura 64 bits	Arquitectura 32 bits	Arquitectura 64 bits
Clock	1	2	1	3
Logic	1	3	1	4
Signals	1	4	3	9
DSP	2	7	3	10
I/0	1	2	6	9
Static	91	91	91	91
Total	97	109	105	119

3.2.3 Análisis de resultados

La velocidad de ejecución de la FPU se aumentó considerablemente en todas la arquitecturas volviéndose tres veces más que el modelo anterior, lo cual permite que se pueda calcular más operaciones con el menor tiempo posible. Desafortunadamente, esto se ve reflejado en la potencia de consumo (aumentos menores al 10%) ya que los módulos deberán manejar más datos dependiendo de los ciclos de reloj así como en los tiempos de carga de los registros donde deberán habilitar más de una vez. Como la velocidad ha aumentado, lo que se recomienda es hacer un manejo en el reloj para que las unidades solamente trabajen cuando el microprocesador lo requiera, así se consumirá menos potencia y la velocidad de la unidad será beneficiosa para consumir menos por tiempo de ejecución.

Para el caso del área, para la suma se tiene una reducción de 28.12% en la arquitectura de 32 bits y 19.35% en la arquitectura de 64 bits cumpliendo con el objetivo planteado. En el caso de la multiplicación para la arquitectura de 32 bits se hace una reducción de 18.51% pero para el caso de 64 bits hay un aumento de 9.87% esto debido a que los módulos de multiplicación para el multiplicador de Karatsuba se hacen más grandes con respecto a la cantidad de bits a procesar, por lo que para una mantisa de 53 bits(requerirá de recursos de 54 bits por ser impar la trama del dato) y esto será mayor cantidad de recursos a utilizar. Con respecto a los problemas de temporizado de la versión anterior, no obstante, el circuito propuesto logra alcanzar una velocidad de reloj de 100MHz.

3.3 Verificación de la FPU

3.3.1 Descripción

Para comprobar que la nueva FPU cumple con la precisión requerida en el resultado, se diseñó un ambiente de verificación para comparar los resultados entre una " referencia dorada" y los resultados calculados por la FPU ya implementada en la FPGA (3.14) la cual, mediante las herramientas disponibles, calcular el porcentaje de error de la arquitectura diseñada con respecto al valor teórico para conocer la fiabilidad de los resultados.

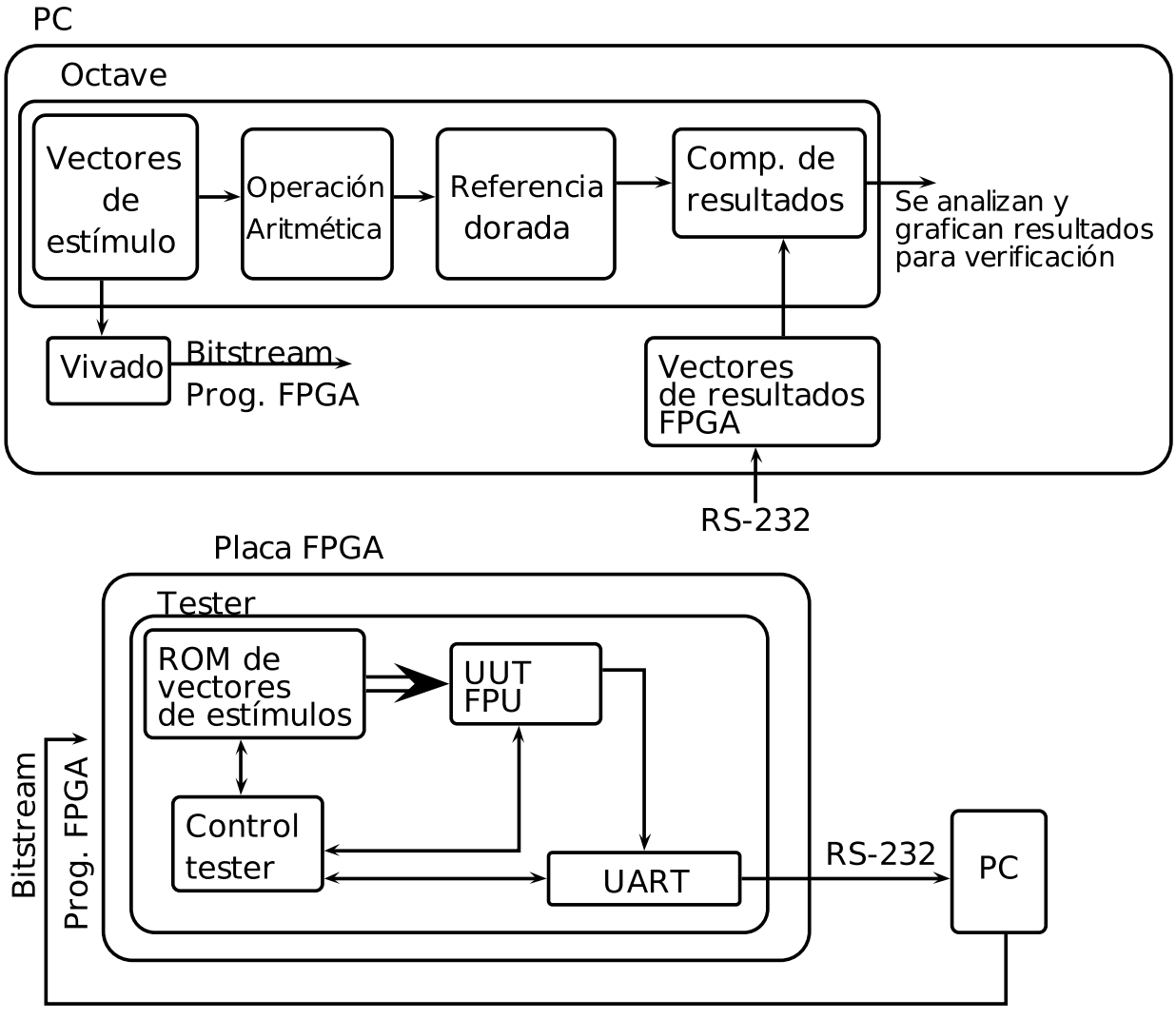

Figura 3.14: Diagrama de bloques del ambiente de verificación

Primero se debió crear la rutina de verificación (figura 3.15). Usando el lenguaje de programación **Python**, se pueden seleccionar diferentes subrutinas, tanto dentro de propio lenguaje como en las rutinas creadas para el programa **Octave** cual mediante el paquete **Oct2Py**, que permite ejecutarlas en segundo plano sin necesidad de abrir el programa.

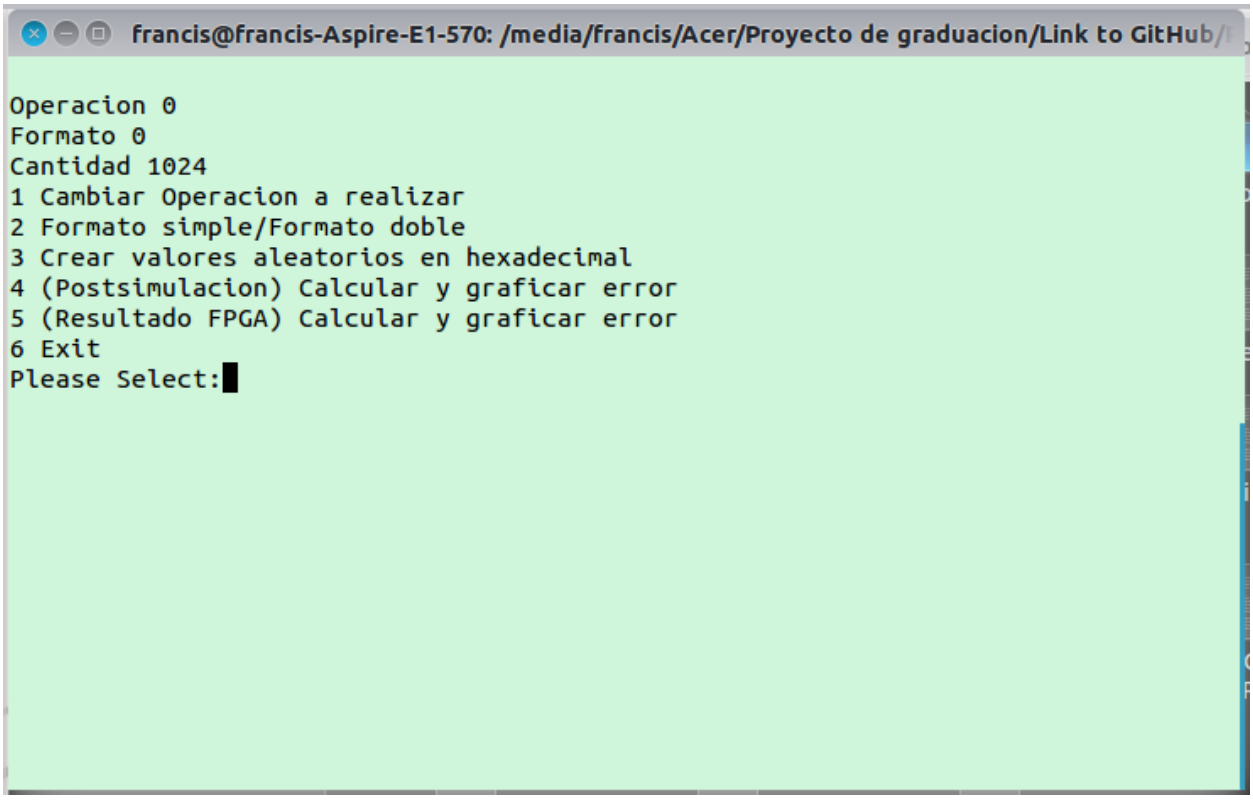

Figura 3.15: Menú principal de la rutina para la verificación de resultados y generación de valores para el ambiente de verificación

Se usó el programa **Octave** para los cálculos matemáticos, tanto para los valores teóricos en el estándar IEEE 754, donde se calcula la operación a simular para 1024 valores con decimales aleatorios dentro del rango [-500,+500]. Estos valores se van almacenando en archivos .txt, tantos los operandos generados como el resultado obtenido que será la "referencia dorada".

Ya con los valores generados, se procede a cargar los valores de los operandos en memorias ROM para un módulo de transmisión serial para la interfaz FPGA-PC. Este módulo es el que se muestra en la figura 3.16: consiste en una unidad de control que hace la lectura de las memorias ROM, ejecuta la FPU para el cálculo seleccionado, multiplexa el resultado en tramas de ocho bits y los va transmitiendo por la salida USB de la FPGA.

Se utiliza otra rutina en **Octave** para abrir el puerto serial en el computador para la captura de datos de la FPGA, en donde se obtienen las tramas de los bits y se carga en otro archivo .txt para realizar la comparación entre el valor teórico y el resultado experimental obtenido.

3.3.2 Resultados

En las figuras 3.17 , 3.18, 3.19 y 3.20 se muestra la graficación de resultados para diferentes operaciones y arquitecturas utilizando el ambiente de verificación expuesto: puede notarse la concordancia entre valores teóricos y experimentales

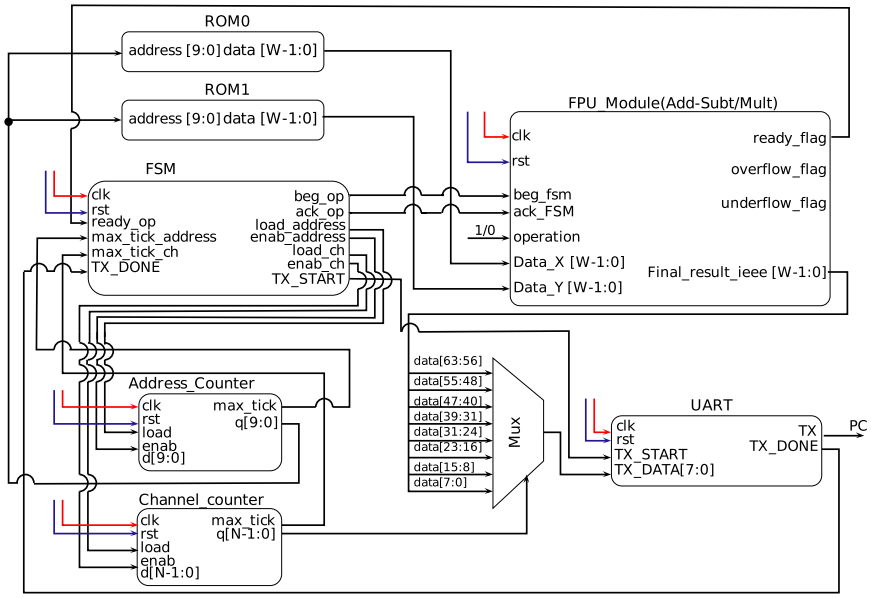

Figura 3.16: Modulo para interfaz serial de datos de la FPU para verificación de resultados en la FPGA

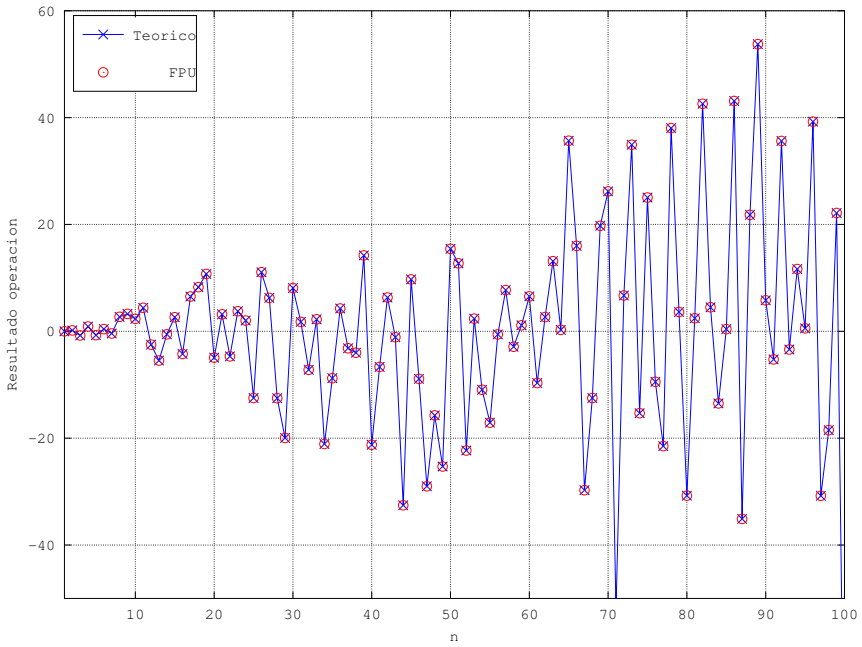

Figura 3.17: Gráfica de comparación de resultados entre el valor teórico y el valor obtenido de la FPU implementado en la FPGA para una suma/resta en arquitectura de 32 bits

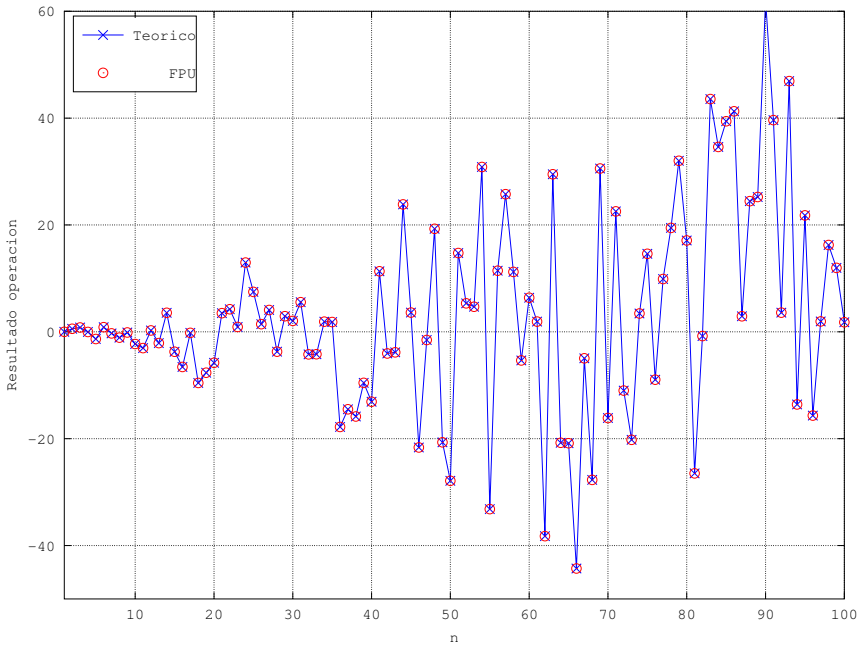

Figura 3.18: Gráfica de comparación de resultados entre el valor teórico y el valor obtenido de la FPU implementado en la FPGA para una suma/resta en arquitectura de 64 bits

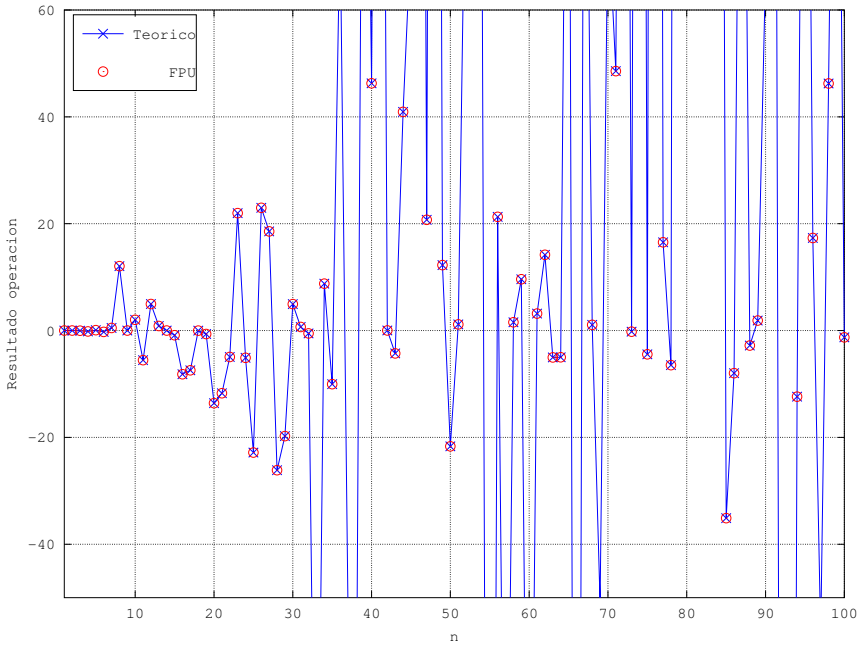

Figura 3.19: Gráfica de comparación de resultados entre el valor teórico y el valor obtenido de la FPU implementado en la FPGA para una multiplicación en arquitectura de 32 bits

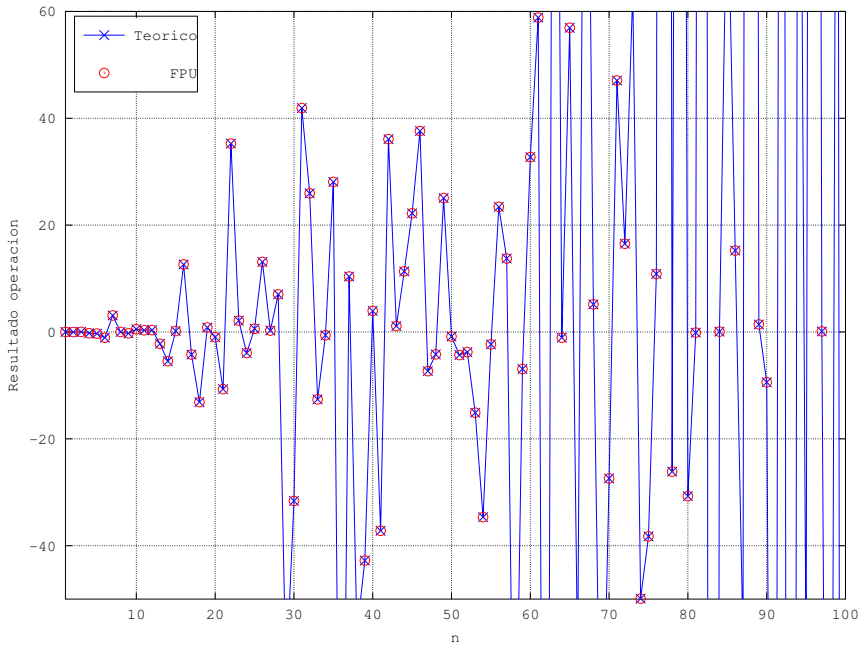

Figura 3.20: Gráfica de comparación de resultados entre el valor teórico y el valor obtenido de la FPU implementado en la FPGA para una multiplicación en arquitectura de 64 bits

En la tabla 3.14 se muestra el promedio de porcentaje de error para estas simulaciones tanto con respecto al valor teórico como para la comparación con la primera versión de la FPU. Así también en la tabla 3.15, se muestra la desviación estándar que hay entre el error de los cálculos con respecto al promedio.

Tabla 3.14: Promedio de los porcentajes de error del ambiente de verificación para las arquitecturas de 32 y 64 bits de la FPU

Operación	Error	Arquitectura 32 bits	Arquitectura 64 bits
Suma/Resta	Teórico	$9.7896e^{-6}$	$3.9247e^{-6}$
	Primera versión	0	0
Multiplicación	Teórico	$4.0614e^{-5}$	$1.4373e^{-5}$
	Primera Versión	0	0

Tabla 3.15: Desviación estándar del error teórico para las arquitecturas de 32 y 64 bits de la FPU

Operación	Arquitectura 32 bits	Arquitectura 64 bits
Suma/Resta	$6.4460e^{-5}$	$3.7438e^{-5}$
Multiplicación	$4.4743e^{-4}$	$8.6514e^{-5}$

3.3.3 Análisis de resultados

Los resultados muestran que, como es de esperar, el nuevo diseño debe ser exactamente igual en resultados a la versión anterior, ya que no debe haber ninguna alteración a nivel de hardware para que los resultados se diferencien entre versiones. Con respecto a la comparación con la "referencia dorada", tanto el error calculado como su desviación estándar nos muestran que, aunque la representación binaria del Estándar IEEE 754 tanto de los operandos como del resultado no dará con exactitud el valor decimal, los porcentajes de error y de desviación estándar son muy bajos por lo que la FPU diseñada obtiene resultados fiables.

Capítulo 4

Conclusiones y recomendaciones

Se concluye que la unidad diseñada, a pesar del aumento en la potencia, llega a ser una unidad más eficiente que la primera versión, ya que con una menor área y menos recursos lógicos, puede hacer más operaciones por cantidad de tiempo. Deber recordarse que el tiempo es un elemento crítico para el reconocimiento de patrones que debe ser bastante rápido en reconocer los sonidos.

Las unidades de hardware extra diseñadas para la unidad también fueron fundamentales ya que ellas permiten que se tenga menos estados para el cálculo de la operación correspondiente. No obstante, para el multiplicador de Karatsuba se deberá resolver el *delay* para la arquitectura en 64 bits y que se pueda ajustar para obtener el resultado de la mantisa en la misma cantidad de ciclos que el exponente. Esto podría lograrse aplicando el mismo algoritmo de Karatsuba en los módulos de multiplicación de las expresiones, tomando en cuenta si los operandos se pueden separar simétricamente a la mitad o no con respecto al ancho de los datos.

Aunque hay un aumento de potencia con el nuevo diseño de la FPU, el tiempo que toma para ejecutar una operación es mucho menor a su versión anterior. Por lo que, si se tiene un control para encender la FPU solo en los momentos en que se requiere calcular, la potencia no se vuelve un problema.

Bibliografía

[1] Ieee 754 number representation. URL http://cseweb.ucsd.edu/classes/wi07/cse140/IEEE754.pdf. 4

[2] Adding floating point numbers, jul 2003. URL https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BinMath/addFloat.html. 6

[3] Multiplying floating point numbers, jul 2003. URL https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BinMath/multFloat.html. 8

[4] Floating point numbers, oct 2008. URL https://www.doc.ic.ac.uk/~eedwards/compsys/float/. 3

[5] R.K.Sharma Arish S. An efficient floating point multiplier design for high speed applications using karatsuba algorithm and urdhva-tiryagbhyam algorithm, 2015. 13

[6] Bob Brown. Shifting and shifters, 1999. URL http://ksuweb.kennesaw.edu/faculty/rbrow211/papers/shifter.pdf. 9

[7] R. Cerdas-Robles, A. Rodriguez, A. Chacon-Rodriguez, and P. Julian. Design of an idm-based determinant computing unit for a 130nm low power cmos asic acoustic localization processor. *Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*, pages 1–4, 24–27, feb 2015. 1

[8] A. Chacon-Rodriguez, Shuo Li, M. Stanacevic, L. Rivas, E. Baradin, and P. Julian. Low power switched capacitor implementation of discrete haar wavelet transform. *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*, pages 1–4, mar 2012. 1

[9] R. K. Montoye E. Hokenek. Leading-zero anticipator (lza) in the ibm risc system/6000 floating-point execution unit, jan 1990. URL https://www.semanticscholar.org/paper/Leading-Zero-Anticipator-LZA-in-the-IBM-RISC-Hokenek-Montoye/d068414847f7ce307fe6d455244583953d7d1078/pdf. iii, 11, 12

[10] C. Gomez-Viquez, L.A. Li-Huang, O. Villalta-Gutierrez, A. Chacón-Rodriguez, and P. Alvarado-Moya. An embedded test system for acoustic pattern recognition intended for environmental monitoring and protection in tropical rain forest reserves. In *ETC2012 Third Embedded Technology Conference*, San José, Costa Rica, jan 2012. 1

[11] Vojin. G Oklobdzija. An algoritmic and novel design of a leading zero detector circuit: Comparison with logic synthesis, mar 1994. URL http://web.ece.ucdavis.edu/~vojin/CLASSES/EEC280/Web-page/papers/Arithmetic/AnAlgorithmiandNovelDesignofaLeading.pdf. 12

[12] D. Rodriguez-Valverde. Diseño e implementación de una unidad aritmético-lógica de coma flotante para un procesador de aplicación específica, 2015. 1, 23

[13] M. Rudolf Pillmeier. Barrel shifter design, optimization, and analysis, 2001. 10

[14] P. Sachan and A. Katiyar. Barrel shifter, sep 2014. iii, 10, 11

[15] C. Salazar. Implementación de un microprocesador de aplicación específica para la ejecución del algoritmo de modelos ocultos de Markov en el reconocimiento de patrones acústicos. Tesis de maestría, Escuela de Ingeniería en Electrónica, Instituto Tecnológico de Costa Rica, dec 2015. 1

[16] C. Salazar-Garcia, L. Alfaro-Hidalgo, M. Carvajal-Delgado, J. Montero-Aragon, R. Castro-Gonzalez, J.A. Rodriguez, A. Chacon-Rodriguez, and P. Alvarado-Moya. Digital integrated circuit implementation of an identification stage for the detection of illegal hunting and logging. *Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*, pages pp.1–4, 24–27, feb 2015. 1

[17] Eric W. Weisstein. Karatsuba multiplication. URL http://mathworld.wolfram.com/KaratsubaMultiplication.html. 13

Apéndice A

Arquitectura de la función Suma/Resta

En esta sección se hace referencia a los módulos que fueron modificados para el nuevo diseño de la arquitectura de la FPU pero que no requieren de una explicación estricta en el documento. Se ofrecen como referencia para el lector.

En la figura A.1 se muestra el diagrama de primer nivel de la unidad de Suma/resta, tal que no debia ser modificada su interfaz con respecto al diseño anterior. Para los demás módulos (figuras A.2 , A.3, A.4 y A.5) su función fue explicada en la tabla 3.6.

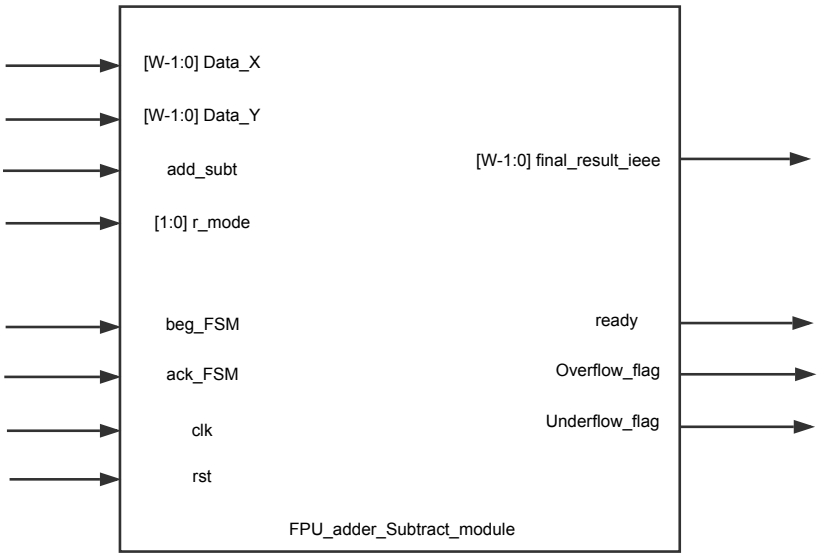

Figura A.1: Diagrama de primer nivel del modulo suma/resta

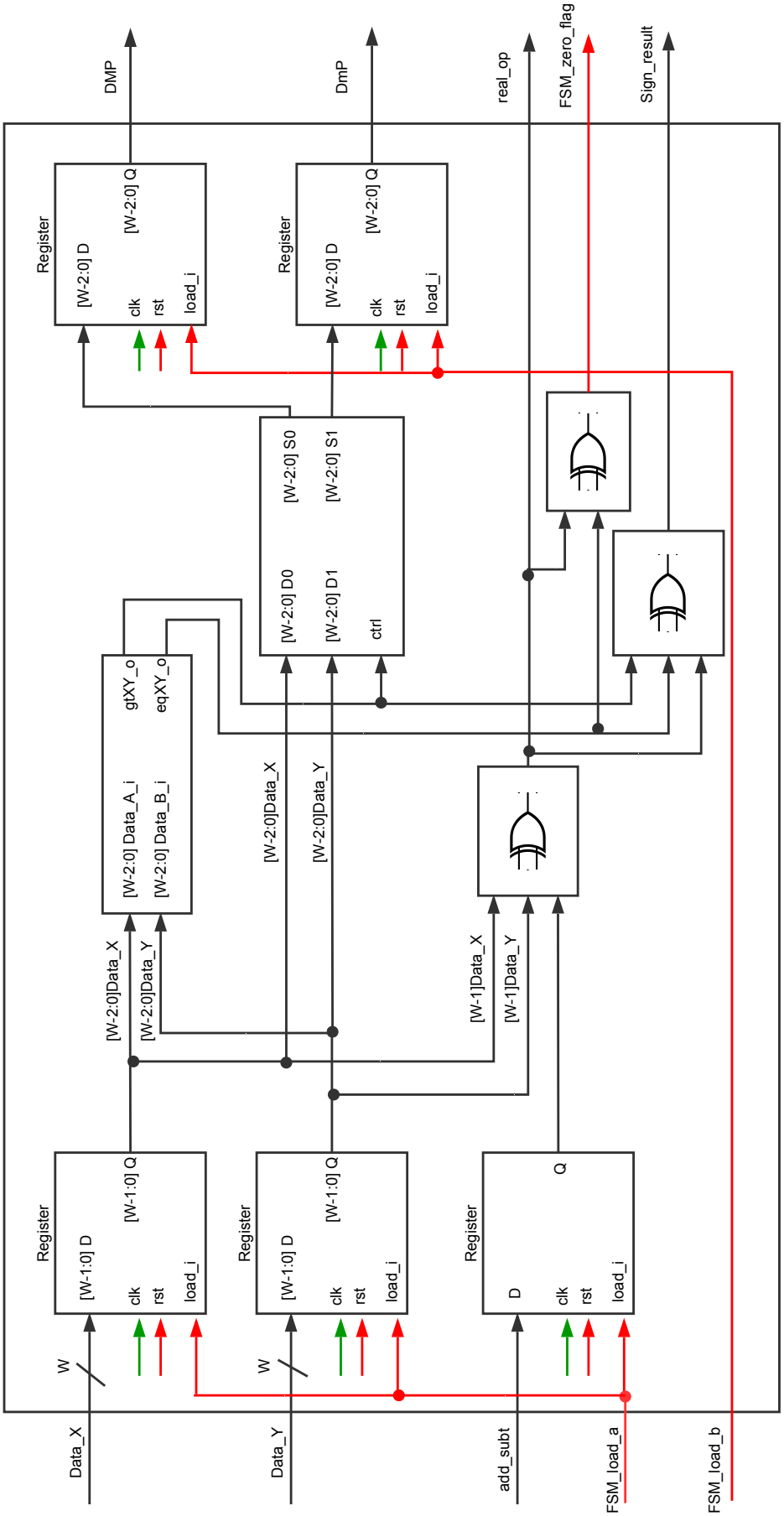

Figura A.2: Módulo Oper_Start_in de la función suma/resta

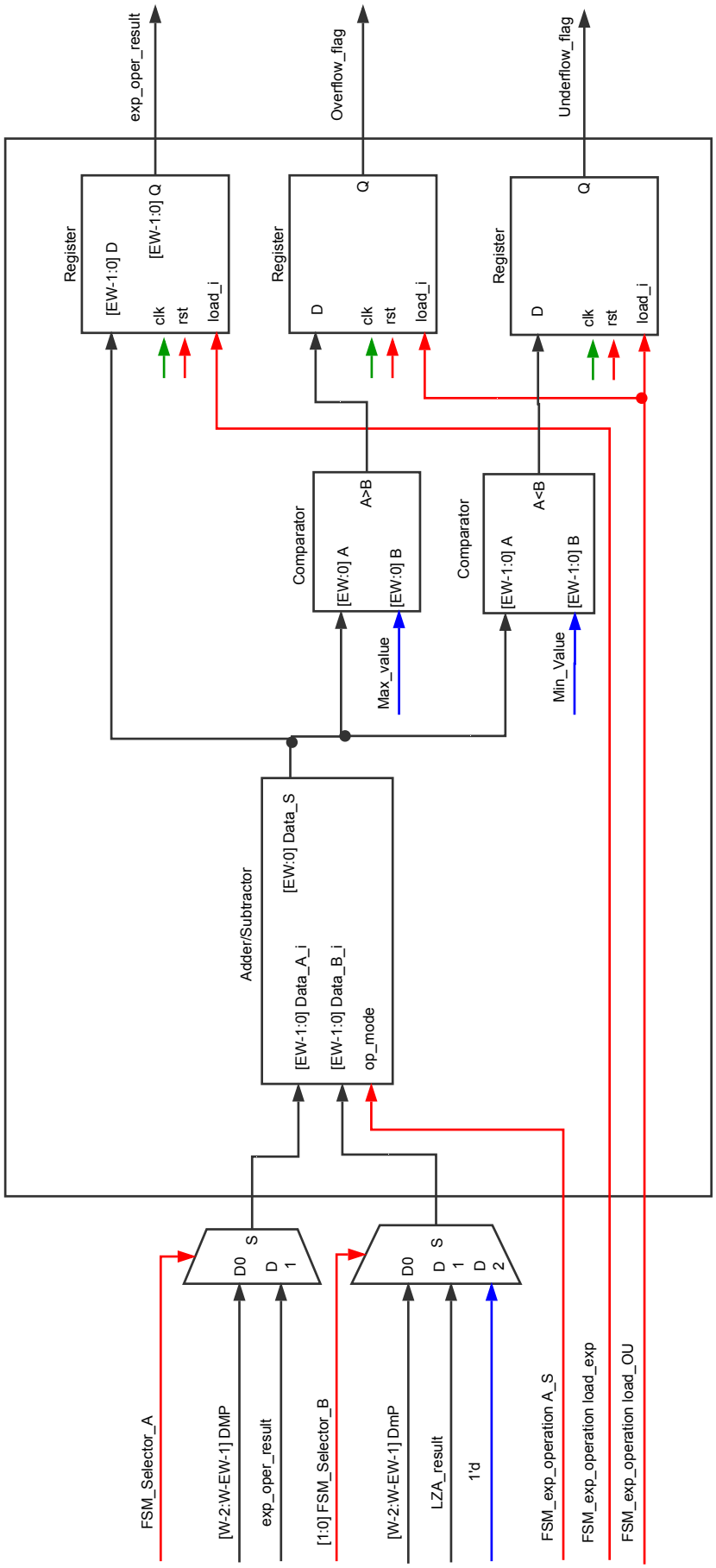

Figura A.3: Módulo Exp_Operation de la función suma/resta

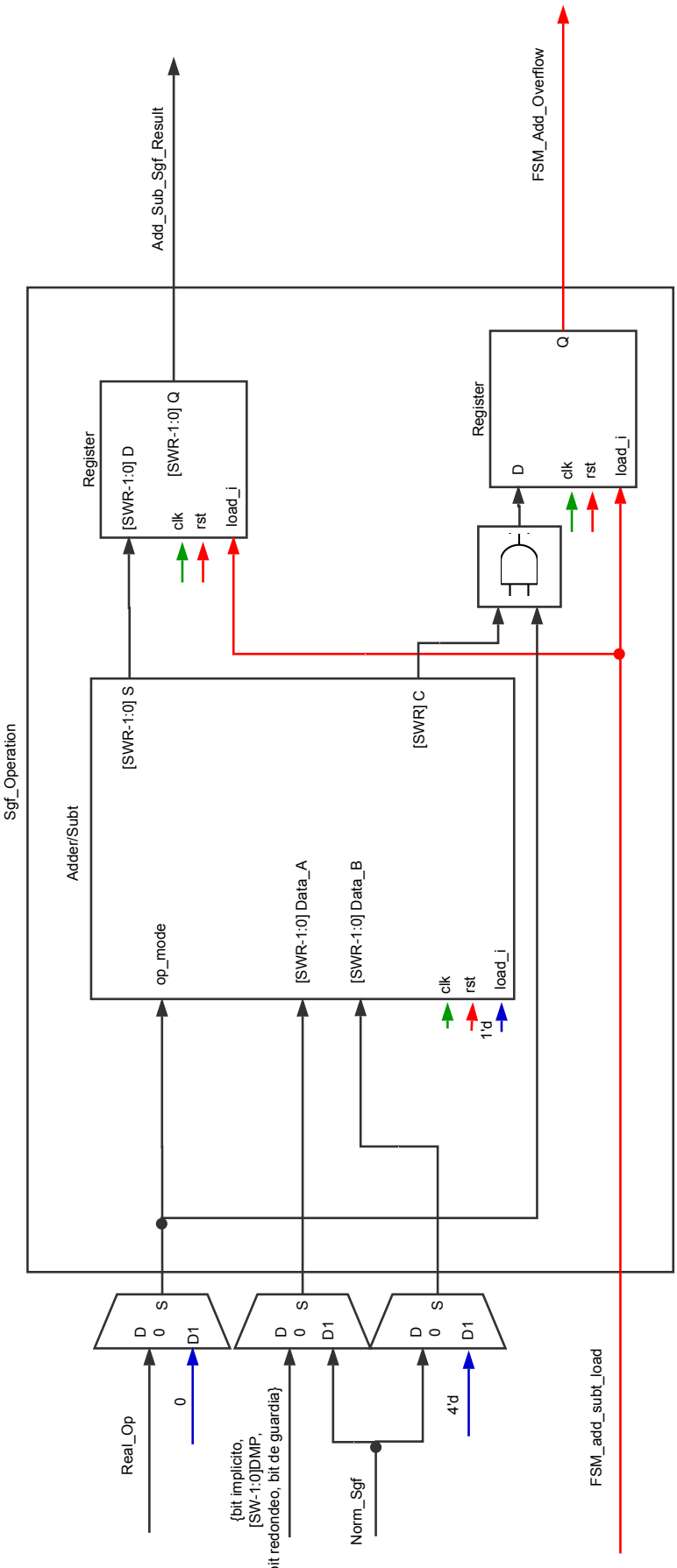

Figura A.4: Módulo Sgf_Operation de la función suma/resta

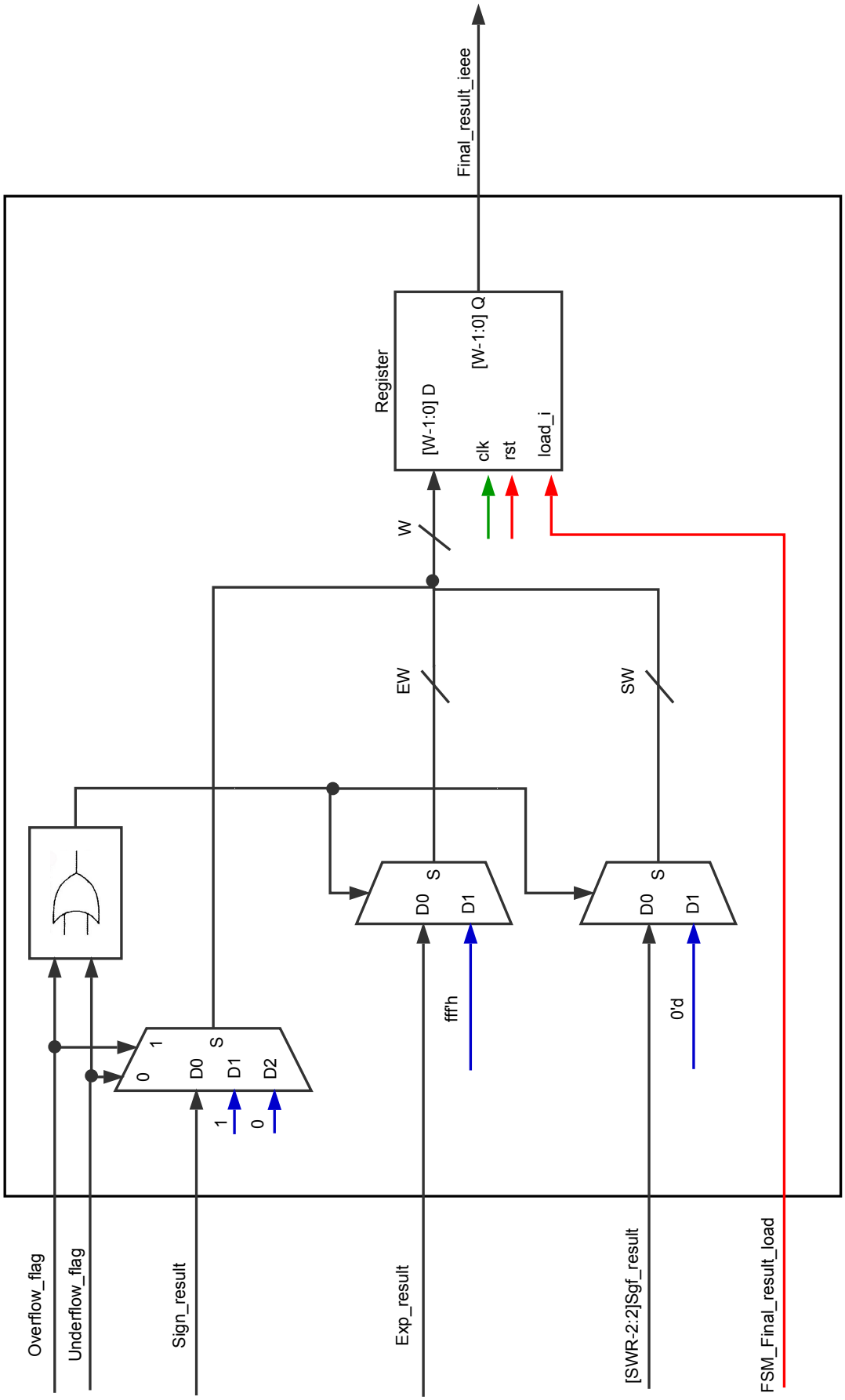

Figura A.5: Módulo Final_Result_IEEE de la función suma/resta

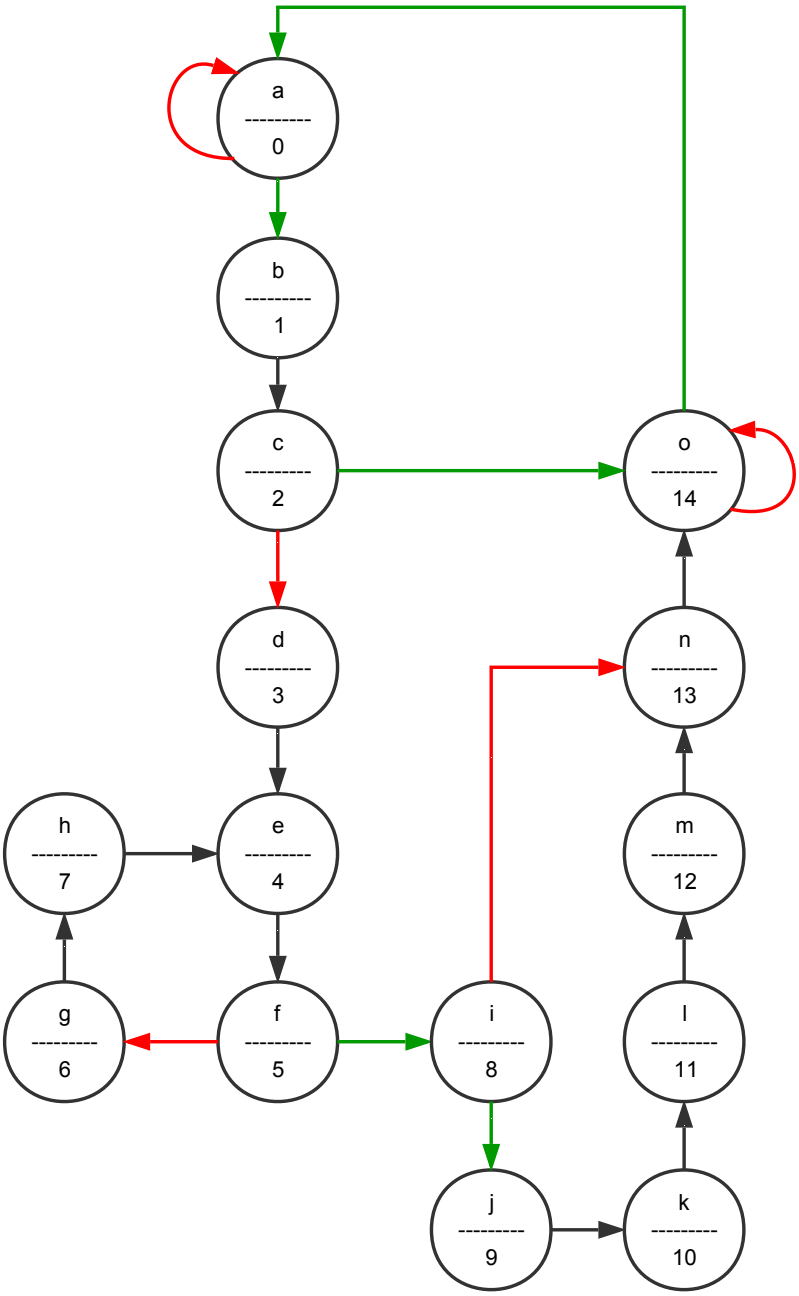

Figura A.6: Diagrama de estados de la FSM de la función suma/resta

Apéndice B

Arquitectura de la función Multiplicación

Se adjuntan en este anexo los diagramas de módulos de la multiplicación como referencia de diseño, entre ellos el diagrama de primer nivel de multiplicación (figura B.1) y los diagramas de los módulos explicados en la tabla 3.10 (figuras B.2 , B.3 , B.4.

También se añade el diagrama de estados de la FSM para el cálculo de la multiplicación (figura B.5).

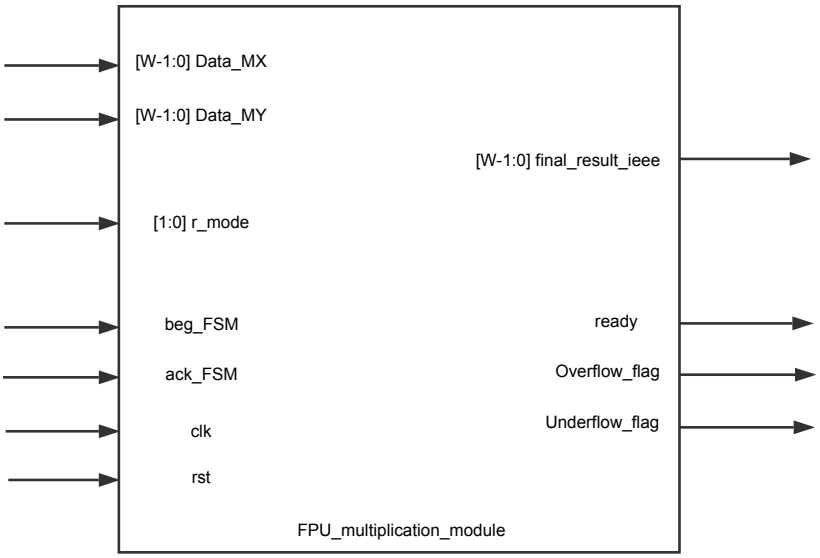

Figura B.1: Diagrama de primer nivel del modulo Multiplicación

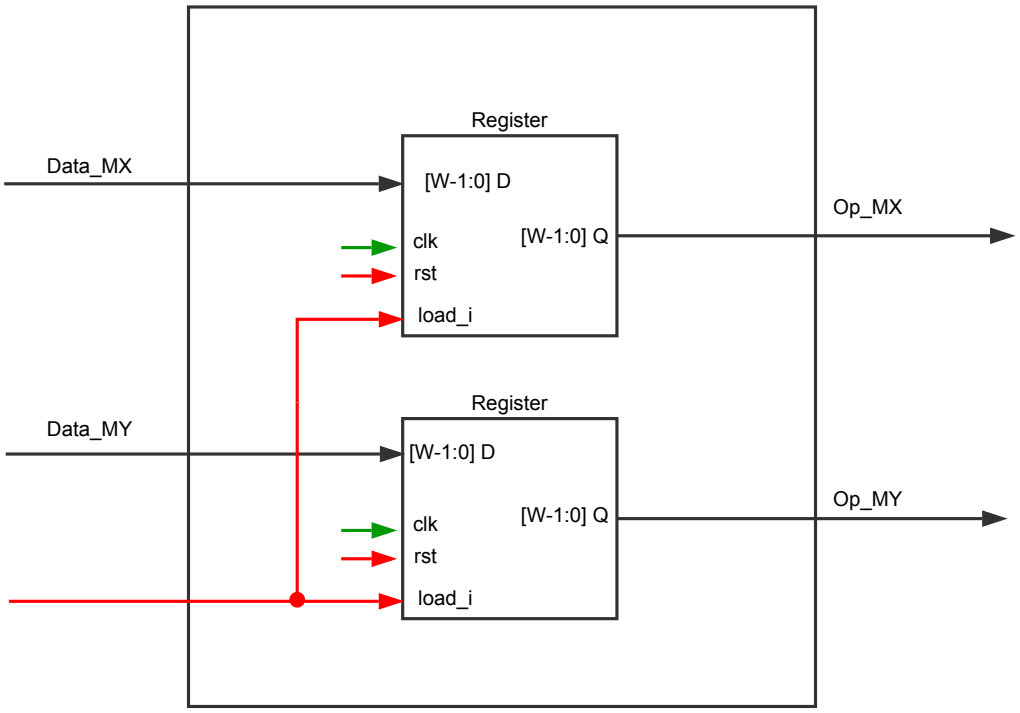

Figura B.2: Módulo Oper_Start_in de la función Multiplicación

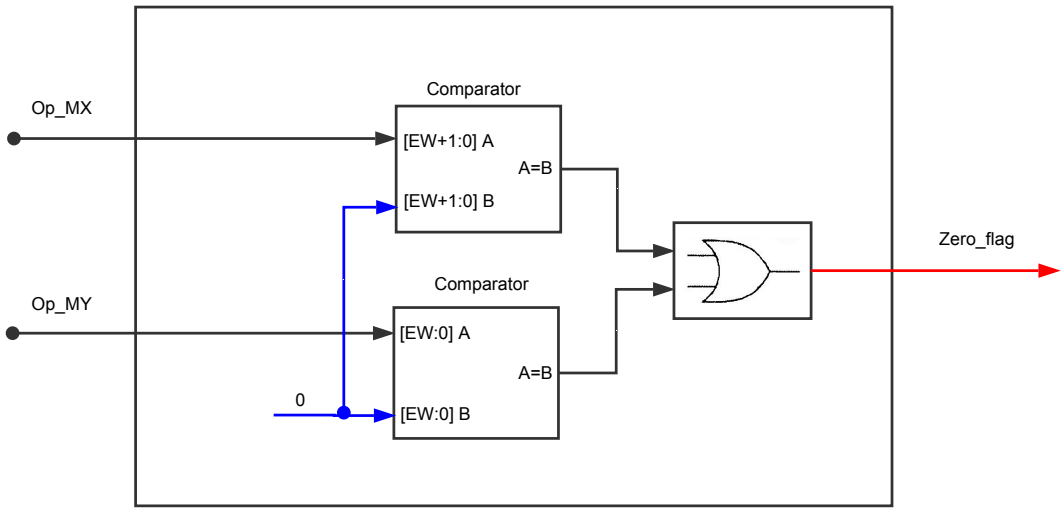

Figura B.3: Módulo Zero_result_detect de la función Multiplicación

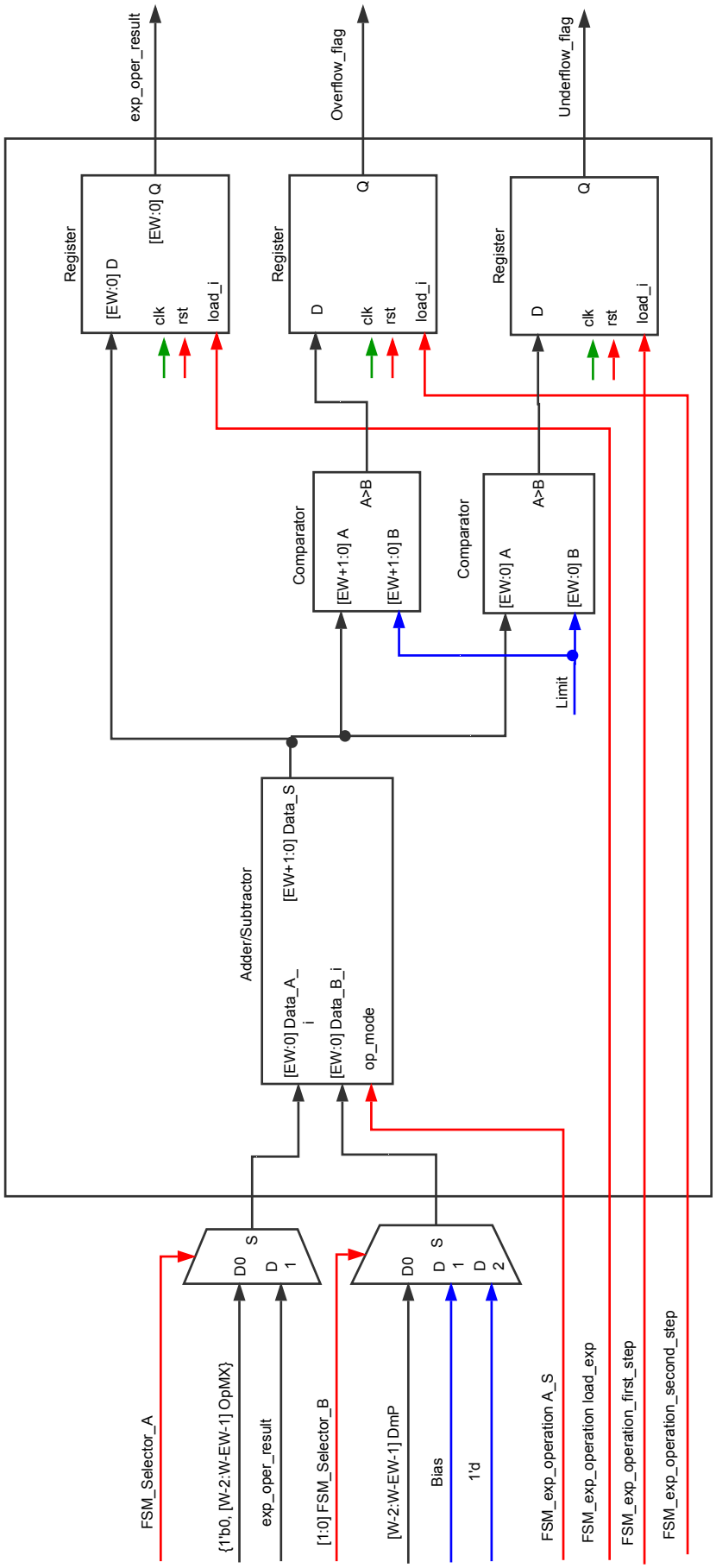

Figura B.4: Módulo Exp_Operation de la función Multiplicación

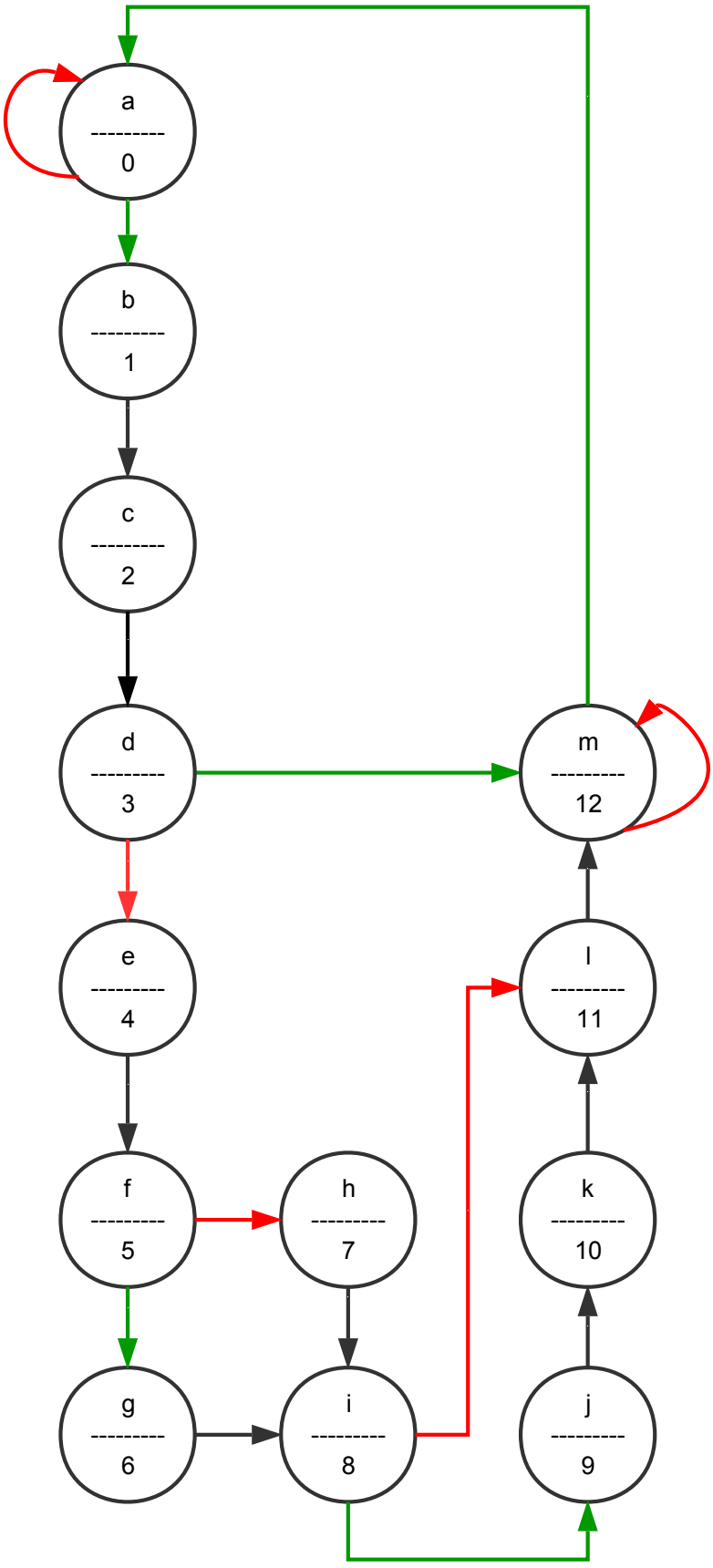

Figura B.5: Diagrama de estados de la FSM de la función Multiplicación