Diseño de una Unidad Funcional en Punto Flotante de Código abierto
Design of an Open Source Floating Point Unit

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Helio Otero Gutiérrez

Director
Alberto del Barrio García

Colaborador
David Mallasén Quintana

Calificación: 8 (NT)

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Diseño de una Unidad Funcional en Punto Flotante de Código abierto
Design of an Open Source Floating Point Unit

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Helio Otero Gutiérrez

Director
Alberto del Barrio García

Colaborador
David Mallasén Quintana

Calificación: 8 (NT)

Convocatoria: *Septiembre 2025*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

9 de septiembre de 2025

Dedicatoria

A mi padre, Alejo, y a mi prima, Elena

Agradecimientos

A Alberto y a David por su paciencia y amabilidad durante el desarrollo de este trabajo. A mi madre, a mi familia y amigos por su apoyo.

Resumen

Diseño de una Unidad Funcional en Punto Flotante de Código abierto

Este trabajo presenta el diseño e implementación de una unidad funcional para datos en punto flotante basada en módulos reutilizables de código abierto. El objetivo principal ha sido construir un bloque compatible con el estándar IEEE 754, capaz de realizar operaciones aritméticas en precisión simple con una arquitectura eficiente y modular. Se han empleado las bibliotecas HardFloat y VFloat como núcleo de cálculo y se han desarrollado módulos propios para el control, la conversión de formatos y la selección de resultados. El diseño ha sido implementado sobre una FPGA Xilinx Alveo U200. Los resultados obtenidos demuestran que el sistema es válido, escalable y apto para integrarse en arquitecturas más amplias dentro de un entorno hardware real.

Palabras clave

FPU, IEEE 754, HardFloat, FPGA, punto flotante, Verilog, Vivado, arquitectura, síntesis, coprocesador

Abstract

Design of an Open Source Floating Point Unit

This work presents the design and implementation of a floating-point functional unit based on reusable open-source modules. The main objective has been to build a block compliant with the IEEE 754 standard, capable of performing arithmetic operations in single precision with an efficient and modular architecture. The HardFloat and VFloat libraries have been employed as the computational core, while custom modules were developed for control, format conversion, and result selection. The design was implemented on a Xilinx Alveo U200 FPGA. The results obtained demonstrate that the system is valid, scalable, and suitable for integration into larger architectures within a real hardware environment.

Keywords

FPU, IEEE 754, HardFloat, FPGA, floating-point, Verilog, Vivado, architecture, synthesis, coprocessor

Índice

1. Introducción **1**

1.1. Motivación 2

1.2. Objetivos 2

1.3. Plan de trabajo 3

2. Antecedentes **5**

2.1. Representación Numérica en Sistemas Digitales 5

2.1.1. Representación de números enteros 6

2.1.2. Representación de números reales 8

2.2. Fundamentos de la Aritmética de Punto Flotante 11

2.2.1. Estructura básica 11

2.2.2. Rango y densidad de representación 11

2.2.3. Normalización y números subnormales 13

2.3. Estándar IEEE 754 15

2.3.1. Formatos del estándar 15

2.3.2. Codificación del exponente 17

2.3.3. Valores especiales 19

2.3.4. Modos de redondeo 20

2.3.5. Manejo de excepciones 21

2.4. Operaciones en Punto Flotante 23

2.4.1. Codificación y decodificación 23

2.4.2. Suma y resta 25

2.4.3. Multiplicación 27

2.4.4. División 28

2.4.5. Raíz cuadrada . . . 30
2.4.6. Fused Multiply-Add . . . 32
2.4.7. Conversiones entre formatos . . . 33
2.4.8. Operaciones de comparación . . . 34

3. Estado del Arte **37**
3.1. Motivación tecnológica . . . 37
3.2. Metodología de comparación . . . 38
3.2.1. Plataforma objetivo . . . 38
3.2.2. Herramienta y flujo de síntesis . . . 38
3.2.3. Métricas comparadas . . . 39
3.3. Panorama de FPUs open-source . . . 40
3.4. Descripción de las candidatas . . . 42
3.4.1. HardFloat . . . 42
3.4.2. FPnew . . . 45
3.4.3. VFloat . . . 48
3.4.4. Taner Öksüz FPU . . . 51
3.5. Evaluación y selección de la unidad funcional . . . 53

4. Diseño y Evaluación de la FPU **59**
4.1. Metodología y herramientas . . . 59
4.2. Diseño de la FPU . . . 60
4.2.1. Visión general . . . 61
4.2.2. Componentes reutilizados . . . 64
4.2.3. Módulos desarrollados . . . 64
4.2.4. Flujo de datos y control . . . 65
4.2.5. Tratamiento de redondeo y excepciones . . . 68
4.3. Simulación . . . 69
4.4. Resultados de síntesis . . . 73
4.5. Diseño alternativo con módulos VFloat . . . 74
4.5.1. Flujo de datos y control . . . 76
4.5.2. Simulación . . . 78
4.5.3. Resultados de síntesis . . . 80

5. Conclusiones y Trabajo Futuro **83**

Introduction **85**

5.1. Motivation 86

5.2. Objectives 86

5.3. Workplan 87

Bibliografía **91**

A. Apéndice A **93**

A.1. Resultados completos de FPnew 93

A.2. Licencias y aspectos legales 93

Índice de figuras

2.1. IEEE 754 (FP32) de 5.75: 0 | 10000001 | 01110000000000000000000. . . 15
2.2. Representación en IEEE 754 con 32 bits. 16
2.3. Representación en IEEE 754 con 64 bits. 16

4.1. Diagrama de bloques de la FPU de módulos Hardfloat. 63
4.2. FSM del módulo de control. 68
4.3. Simulación de la suma $2.0 + 2.0 = 4.0$ sin excepciones. 70
4.4. Simulación de la multiplicación $2.0 \times 2.0 = 4.0$ sin excepciones. 70
4.5. Simulación de la división $2.0/2.0 = 1.0$ sin excepciones. 71
4.6. Simulación de la raíz cuadrada $\sqrt{4.0} = 2.0$ sin excepciones. 71
4.7. Excepción *invalid*: división $0.0/0.0 \rightarrow$ NaN. 72
4.8. Excepción *overflow+inexact*: multiplicación de máximo número finito por 2. 73
4.9. Diagrama de bloques de la FPU de módulos VFloat. 75
4.10. Diagrama de estados de la FSM de VFloat. 77
4.11. Simulación de la operación de suma $1.0 + 2.0 = 3.0$ sin excepciones. . 78
4.12. Simulación de la operación de resta $5.5 - 2.5 = 3.0$. 79
4.13. Simulación de la multiplicación $2.0 \times 2.0 = 4.0$. 79
4.14. Simulación de la división $8.0 \div 2.0 = 4.0$. 80
4.15. Simulación de la operación de raíz cuadrada $\sqrt{25.0} = 5.0$. 80

Índice de tablas

2.1. Representación de +5 y −5 en diferentes codificaciones con 4 bits. . . 7

2.2. Comparativa de representaciones con signo (4 bits). 8

2.3. Comparativa entre punto fijo y punto flotante. 10

2.4. Extremos del rango representable en formatos binarios (IEEE 754). . 12

2.5. Comparación entre números normalizados y subnormales. 15

2.6. Parámetros de los formatos binarios definidos en IEEE 754-2019. . . . 17

2.7. Codificación de valores especiales en punto flotante. 19

2.8. Resultado de $|x| \pm |y|$ para diferentes combinaciones de operandos positivos. 26

2.9. Resultado de $|x| \times |y|$ para combinaciones de operandos positivos. . . 28

2.10. Resultado de $|x|/|y|$ para diferentes combinaciones de operandos positivos. 30

2.11. Resultado de $\sqrt{x}$ para operandos especiales. 32

2.12. Relaciones verdaderas y sus negaciones en IEEE 754. 35

3.1. Resultados comparativos por formato. 54

3.2. Resultados de síntesis de FPnew multiformato. 55

3.3. Resultados de VFloat con configuraciones LUT/DSP y PipeRegs recomendados por Vivado. 56

4.1. Interfaz externa e internas principales del módulo `FPU_Wrapper`. . . . 65

4.2. Codificación de operaciones y rutas de datos en la FPU. 67

4.3. Resultados de síntesis de la FPU para distintas precisiones. 73

4.4. Señales adicionales y específicas del módulo `FPU_VFloat_Wrapper`. . . 76

4.5. Resultados de síntesis de la FPU VFloat. 80

A.1. Resultados completos de síntesis de FPnew (MHz redondeados). . . . 93

Capítulo 1

Introducción

En el ámbito de la computación, la representación numérica juega un papel fundamental porque define cómo se almacenan y manipulan los números reales. Esta representación debe ser precisa y eficiente. A día de hoy, las representaciones utilizadas de manera más amplia son punto fijo y punto flotante, que se escogerán dependiendo del contexto y las necesidades de cada aplicación. Como norma general, la representación en punto fijo supone un aumento significativo del rendimiento, útil en sistemas empotrados; en cambio, si se trata de un proyecto que requiere alta precisión, como simulaciones científicas o análisis estadístico, es mejor usar el punto flotante.

La aritmética de punto flotante es fundamental en casi todas las áreas de la informática y la ingeniería. Las operaciones básicas como la suma, la resta, la multiplicación y la división son realizadas por unidades funcionales especializadas conocidas como Floating Point Unit (FPU). Estas FPUs son componentes críticos en arquitecturas de hardware modernas.

Aunque la aritmética de punto flotante se remonta a propuestas como la de Torres Quevedo en 1914, sus primeras implementaciones modernas reales se dieron en los ordenadores Z1 (1938) y Z3 (1941) de Konrad Zuse. Como se explica en (Muller et al., 2018a, cap. 1), el Z3 operaba con una arquitectura en coma flotante binaria y representaciones especiales como infinito e indeterminados. Décadas más tarde, estas ideas serían formalizadas en el estándar IEEE 754 (1985), que estableció una representación unificada y ampliamente adoptada en procesadores a partir de los años 2000.

Existen diversos proyectos de código abierto en los que se implementan diferentes operaciones de punto flotante. Las distintas unidades no tienen por qué interactuar entre ellas y se pueden tratar de manera independiente, es por eso que para facilitar la usabilidad y el encapsulamiento es frecuente el uso de una interfaz que englobe el resto de componentes, conocido como wrapper.

En este proyecto me centraré en comparar las diferentes unidades funcionales de punto flotante de código abierto para después poder desarrollar una FPU completa y ofrecer una manera unificada y eficiente de realizar operaciones en punto flotante.

1.1. Motivación

Dada la gran importancia de la representación y manipulación de los números reales en el campo de la computación, es esencial tener en cuenta diferentes factores para operar con una representación numérica de manera eficiente. El movimiento de código abierto adquiere un papel fundamental en este campo, ya que fomenta la colaboración entre investigadores y desarrolladores y facilita el acceso a tecnologías más avanzadas y adecuadas para cada proyecto. Sin embargo, esto también puede provocar confusión entre los investigadores que deseen utilizar uno de estos proyectos, debido tanto a la amplia variedad de opciones disponibles como a la falta de documentación en muchos casos.

La elección y diseño de una FPU adecuada puede tener un impacto significativo en el rendimiento y la eficiencia de una aplicación. Con el uso generalizado del estándar IEEE 754, las FPUs han evolucionado considerablemente, proporcionando una gran eficiencia en cálculos en punto flotante. No obstante, cada implementación puede variar en términos de rendimiento, precisión, consumo energético y complejidad, lo que hace que sea imprescindible una evaluación detallada de las mismas.

Este proyecto se propone abordar esta problemática mediante una comparación de las diferentes unidades funcionales de punto flotante de código abierto. Utilizando esta información, busco desarrollar una FPU completa que incluya estas unidades y sea capaz de operar de manera eficiente.

1.2. Objetivos

El objetivo principal de este trabajo es diseñar una FPU completa con módulos de operaciones de punto flotante y fácil de usar. Específicamente, las operaciones de suma/resta, multiplicación y división/raíz cuadrada. Para lograr esto, se tomaron como base los módulos desarrollados en código abierto del proyecto Hardfloat y de VFloat. Los objetivos específicos incluyen:

a. Investigar y evaluar las unidades de procesamiento de coma flotante (FPU) de código abierto disponibles.

b. Comparar la eficiencia, el coste y la frecuencia máxima de las diferentes FPUs disponibles.

c. Seleccionar los módulos de operaciones de punto flotante más adecuados para el desarrollo del wrapper.

d. Diseñar e implementar un wrapper en SystemVerilog para integrar los módulos seleccionados en un entorno de diseño de hardware.

1.3. Plan de trabajo

Para llevar a cabo este trabajo de manera eficaz, he dividido su estructura en múltiples etapas. Cada una de estas etapas me permitirá llevar una constancia de todo el proceso y los objetivos.

1. Investigación de FPUs de código abierto:
   - Realizar una busqueda exhaustiva de los proyectos existentes sobre FPUs de código abierto disponibles.
   - Identificar las características clave, ventajas y limitaciones de cada una de ellas.
   - Recopilar información relevante sobre la eficiencia, el coste, el área, la latencia y frecuencia máxima de operación de cada FPU.
2. **Selección y comparación de FPUs:**
   - Seleccionar un conjunto de FPUs candidatas basadas en la investigación realizada.
   - Comparar las FPUs seleccionadas en términos de eficiencia, coste y frecuencia máxima de operación.
   - Evaluar la idoneidad de cada FPU para su integración en el diseño del wrapper.
3. **Diseño inicial de la FPU:**
   - Definir las señales de entrada y salida del wrapper, incluyendo los formatos de los operandos y resultados.
   - Diseñar la lógica de control del wrapper para coordinar las operaciones de las FPUs seleccionadas.
4. **Implementación y simulación del diseño final en SystemVerilog:**
   - Desarrollar el código en SystemVerilog que implemente el diseño del wrapper definido previamente.
   - Realizar simulaciones para verificar el funcionamiento correcto del wrapper bajo diferentes condiciones de entrada.
   - Depurar y resolver cualquier problema de compatibilidad o incompatibilidad que pueda surgir durante la implementación.
5. **Verificación y validación del wrapper:**
   - Realizar pruebas de unidad para verificar el correcto funcionamiento del wrapper en diferentes escenarios de entrada.
   - Realizar pruebas de integración para asegurar la compatibilidad y coherencia del wrapper con las FPUs seleccionadas.

Capítulo 2

Antecedentes

En este capítulo se proporciona un contexto teórico y práctico para comprender el diseño y desarrollo de una unidad funcional en punto flotante.

En primer lugar, se exploran los conceptos básicos de la aritmética en punto flotante. Además, se explica el estándar IEEE 754 para la representación de números en punto flotante. Después se describen las características y aplicaciones del lenguaje de descripción de hardware SystemVerilog, y se analiza la importancia de las unidades funcionales en la computación moderna. Por último, se revisan las herramientas de diseño comúnmente utilizadas en el proceso de desarrollo de estas unidades funcionales.

Con estos fundamentos se establece una base sólida de conocimientos y conceptos necesarios para el desarrollo de este proyecto.

2.1. Representación Numérica en Sistemas Digitales

Los sistemas digitales operan internamente con secuencias binarias de longitud fija. Esta restricción requiere técnicas específicas para representar valores numéricos de forma eficiente, abarcando tanto enteros con signo como cantidades reales. A lo largo de la historia de la Arquitectura de Computadores, se han definido distintos esquemas de codificación que permiten realizar operaciones aritméticas de forma predecible y fiable, con implicaciones directas en el diseño del hardware y la detección de errores.

En esta sección se abordan los métodos más relevantes para la representación numérica: primero se estudiarán las técnicas para representar enteros con signo, esenciales para operaciones básicas como suma y resta; a continuación, se analizarán los formatos utilizados para codificar números reales, los cuales permiten ampliar el rango y la precisión mediante el uso de punto fijo y punto flotante.

2.1.1. Representación de números enteros

La representación de números enteros con signo es esencial para implementar operaciones aritméticas eficientes. La codificación binaria con una longitud de palabra fija, como puede ser de 8, 16, 32 o 64 bits, permite representar tanto números positivos como negativos. Sin embargo, esta codificación plantea retos técnicos, como la interpretación del bit más significativo (MSB), el tratamiento del signo y la detección de condiciones como el desbordamiento.

Desde sus orígenes, los sistemas digitales han ido utilizando tres esquemas principales para representar enteros con signo: **magnitud y signo**, **complemento a uno** y **complemento a dos**. Cada uno de estos métodos tiene implicaciones distintas en la lógica aritmética, la eficiencia del hardware y la semántica de operaciones como la suma y la comparación.

Magnitud y signo

Este formato reserva el bit más significativo para indicar el signo del número. Cuando dicho bit es 0, el valor es positivo, y cuando es 1, el valor es negativo. El resto de los bits codifican el valor absoluto. Este sistema presenta un rango de valores simétrico respecto al cero:

$$\text{Rango} = [-(2^{n-1} - 1),\ +(2^{n-1} - 1)]$$

Aunque su estructura es intuitiva, introduce una ambigüedad importante: existen dos representaciones distintas para el número cero. Por ejemplo, con 4 bits usaríamos `0000` y `1000` para codificar el cero. Esta duplicación complica las operaciones lógicas y el diseño del hardware.

Además, la operación de suma no puede realizarse directamente sin tener en cuenta el signo de los operandos, lo que obliga a introducir lógica adicional en la unidad aritmético-lógica (ALU) para gestionar los signos por separado.

Complemento a uno

Este sistema también utiliza el bit más significativo como indicador del signo. Los valores positivos se mantienen igual que en binario puro y los negativos se obtienen invirtiendo todos los bits del número positivo correspondiente. Aunque la lógica es más uniforme que en signo y magnitud, este método hereda el problema de la doble representación del cero. Con 4 bits, tanto `0000` como `1111` codifican el valor cero.

Otro inconveniente técnico destacado es la necesidad de gestionar el *acarreo circular* en las sumas, un fenómeno directamente relacionado con el concepto de acarreo de salida. Este último hace referencia al bit adicional que se genera cuando la suma de los bits más significativos produce un resultado que excede la capacidad del sistema. En complemento a uno, dicho bit no se descarta: debe sumarse nuevamente

al bit menos significativo para que la operación sea correcta. Este proceso introduce una etapa adicional en la lógica aritmética, encareciendo el hardware.

Estas desventajas motivaron la transición hacia un sistema más robusto: el complemento a dos.

Complemento a dos

Este método surge como respuesta a las carencias observadas en los sistemas anteriores, resolviendo varios problemas técnicos de forma simultánea.

En primer lugar, permite utilizar la misma lógica binaria tanto para la suma como para la resta, ya que restar un número equivale a sumar su complemento a dos. Esto elimina la necesidad de circuitos separados para cada operación y simplifica el diseño de la ALU. En segundo lugar, impide la doble representación del cero, lo que evita ambigüedades y facilita las comparaciones. En tercer lugar, la detección de desbordamiento se vuelve sencilla y consistente, gracias a un criterio basado en los acarreos del bit de signo, algo que en C1 no ocurre. Por último, ofrece una mejor utilización del rango disponible: al reservar una única combinación binaria para el cero, permite codificar un valor negativo adicional.

Para obtener la representación en complemento a dos de un número, primero se obtiene el complemento a uno, invirtiendo todos los bits y después se suma una unidad al resultado:

$$C_2(-5_{(10)}) \Rightarrow \text{NOT}(0101) = 1010 \Rightarrow 1010 + 1 = 1011$$

Decimal	Signo y magnitud	Complemento a uno	Complemento a dos
$+5$	0101	0101	0101
-5	1101	1010	1011

Tabla 2.1: Representación de $+5$ y -5 en diferentes codificaciones con 4 bits.

La detección del desbordamiento se facilita ya que puede determinarse comparando el acarreo de entrada y salida del bit más significativo mediante una puerta XOR:

$$\text{overflow} = \text{Carry}_{in_MSB} \oplus \text{Carry}_{out_MSB}$$

Por ejemplo, en una operación de 4 bits, la suma de $+7$ (0111) y $+3$ (0011) da como resultado 1010, que equivale a -6. Ambos operandos son positivos, por lo que el resultado también debería serlo. Sin embargo, el bit de signo es 1, indicando un valor negativo. Esto ocurre porque el resultado excede el rango positivo representable, activando la condición de desbordamiento. La fórmula anterior detecta esta situación comparando los acarreos de entrada y salida del bit de signo, que en este caso

son distintos. Esta verificación puede realizarse automáticamente por la ALU para prevenir errores silenciosos durante la ejecución.

El rango representable en complemento a dos es asimétrico, ya que incluye un número negativo más que positivos. Este comportamiento se debe a que el cero solo tiene una representación, lo cual deja disponible una combinación binaria adicional para representar un número negativo. Esta característica no solo mejora el aprovechamiento del espacio binario, sino que también resulta práctica en algoritmos que requieren cubrir un mayor espectro de valores negativos, como ciertos procesos de control y señal.

$$\text{Rango} = [-2^{N-1},\ 2^{N-1} - 1]$$

En conclusión, el estudio de las diferentes codificaciones para representar enteros con signo permite entender las decisiones que han guiado el diseño de los sistemas actuales. El complemento a dos resuelve eficazmente los problemas de ambigüedad, complejidad lógica y detección de errores presentes en los esquemas anteriores, consolidándose como la solución más utilizada en arquitecturas digitales contemporáneas.

Representación	**Rango**	**Doble cero**	**Suma directa**	**Hardware**
Magnitud y signo	$[-7, +7]$	Sí	No	Complejo
Complemento a uno	$[-7, +7]$	Sí	Parcial	Medio
Complemento a dos	$[-8, +7]$	No	Sí	Eficiente

Tabla 2.2: Comparativa de representaciones con signo (4 bits).

2.1.2. Representación de números reales

Los números reales deben representarse de tal forma que los sistemas informáticos sean capaces de entenderlos y manejarlos de manera eficiente y precisa. A diferencia de los números enteros, los números reales pueden tomar valores fraccionarios y cubrir un rango de valores mucho más amplio. La limitación de la memoria implica que no es posible almacenar números con precisión infinita; por lo tanto, es necesario establecer un límite en la precisión. A continuación, se analizan los dos métodos principales: punto fijo y punto flotante.

Punto fijo

En la representación en **punto fijo**, la posición del punto binario se establece de manera estática durante el diseño del sistema, dividiendo los bits disponibles en una parte entera y otra fraccionaria. Una manera de representar números en punto fijo es con el formato $Qm.n$, donde la letra "Q" se utiliza por convención para designar este tipo de codificación. Un número x se expresa como:

$$x = \frac{k}{2^n} \tag{2.1}$$

donde:

- k es un entero almacenado en complemento a dos,
- m bits representan la parte entera incluyendo el signo,
- n bits representan la parte fraccionaria.

Este sistema tiene la ventaja de ser fácil de implementar en hardware, porque las operaciones se hacen igual que con enteros. Lo que lo hace muy útil en aplicaciones que no requieren una gran variabilidad y los valores están acotados, por ejemplo, en aplicaciones de procesamiento de señal (DSP), donde los valores no suelen salirse de un rango predecible y el consumo energético debe ser bajo. Su principal desventaja es que el rango de números que puede representar está limitado por el número de bits asignado y no se puede cambiar sin rediseñar el sistema. Ese rango está dado por:

$$\text{Rango} = \left[-2^{m-1},\ 2^{m-1} - 2^{-n}\right] \tag{2.2}$$

Esto obliga a elegir cuidadosamente el formato adecuado. Si el valor a representar está fuera del rango, se debe reescalar manualmente, lo cual puede introducir errores.

Por ejemplo, con un formato $Q3.2$, el número binario `10110` se interpreta como un entero con signo en complemento a dos. Sabemos que hay 2 bits fraccionarios, por lo que el valor decimal correspondiente se obtiene como:

$$x = \frac{k}{2^2} = \frac{-10}{4} = -2.5$$

donde $k = -10$ es el valor en decimal del entero con signo `10110`, interpretado en complemento a dos de 5 bits. El rango de representaciones posibles en este formato, usando la Fórmula (2.2), es:

$$[-2^{3-1},\ 2^{3-1} - 2^{-2}] = [-4,\ 3.75]$$

Punto flotante

El punto flotante soluciona las limitaciones del punto fijo, permitiendo que la posición del punto binario se adapte dinámicamente en función del valor a representar. De esta forma, es posible codificar tanto números extremadamente pequeños como muy grandes sin necesidad de reescalarlos manualmente.

La idea se basa en una representación similar a la notación científica, donde los números se expresan como una cifra significativa multiplicada por una potencia de

base dos. Este enfoque permite ampliar notablemente el rango de valores representables y ajustar la precisión relativa según la magnitud del número.

Por ejemplo, el número 2.5 escrito en binario como 10.01_2, se puede adaptar al formato de punto flotante, se normaliza desplazando el punto binario hasta obtener $1.001_2 \times 2^1$.

Este tipo de representación es habitual en aplicaciones donde se requiere trabajar con órdenes de magnitud muy dispares, como en simulaciones científicas, gráficos tridimensionales o aprendizaje automático. Eso sí, su implementación en hardware es más compleja porque hay que alinear los exponentes antes de operar, redondear los resultados y controlar excepciones, conceptos que veremos más adelante.

Más adelante, en la sección 2.2.1, se explicará con mayor detalle la estructura interna de este formato y los elementos que lo componen.

Criterio	**Punto fijo**	**Punto flotante**
Rango	Limitado y fijo	Amplio y dinámico
Precisión	Constante	Relativa al valor
Complejidad hardware	Baja	Alta
Aplicación típica	DSP, sistemas embebidos	Cómputo científico, IA

Tabla 2.3: Comparativa entre punto fijo y punto flotante.

Ambas técnicas ofrecen ventajas significativas en distintos escenarios. Conocer sus principios conceptuales es fundamental para decidir cuál utilizar en función de las necesidades de precisión, rango, eficiencia o coste computacional del sistema.

2.2. Fundamentos de la Aritmética de Punto Flotante

2.2.1. Estructura básica

Un número se descompone en tres componentes: un signo S, una mantisa M, que almacena los dígitos significativos; y un exponente E, siguiendo la fórmula general:

$$x = (-1)^s \cdot 1.m \cdot 2^{e-bias} \tag{2.3}$$

donde:

- s es el bit de signo, que determina si el número es positivo o negativo.
- m representa la mantisa, que codifica la parte significativa del número.
- e es el exponente, que determina la escala del número mediante una potencia de la base β, que en los sistemas digitales es 2.

La parte significativa es el conjunto de dígitos que definen el valor del número, sin tener en cuenta su magnitud. Normalmente se representa como un número fraccionario normalizado. En ese caso, el dígito más significativo es un 1 implícito que no se almacena físicamente. Este aspecto se ampliará en los apartados dedicados a la normalización y codificación.

El exponente permite desplazar el punto binario a distintas posiciones. Esto hace que el punto "flote", lo que da nombre a este formato. Este desplazamiento es el que permite representar un gran rango de valores manteniendo una buena precisión relativa. Conviene señalar que el valor que se almacena en la representación binaria no es el exponente real, sino una versión ajustada mediante un sesgo.

En los siguientes apartados se explicarán aspectos como la normalización, los límites del rango representable, el sesgo del exponente y la codificación binaria de cada campo conforme al estándar IEEE 754.

2.2.2. Rango y densidad de representación

Tras haber presentado la estructura general del formato de punto flotante, en este apartado se profundiza en cómo el exponente determina el rango de valores representables y cómo estos se distribuyen a lo largo de la recta real. Estos aspectos afectan directamente a la precisión y al diseño de los circuitos encargados de operar con números en coma flotante.

Para comenzar, analizaremos cuál es el rango de representación permitido por el sistema de punto flotante. El rango está definido por los valores mínimo y máximo normalizados que puede representar, lo que depende directamente de los límites del exponente. Estos se expresan como $e_{\min}$ y $e_{\max}$, que indican el menor y mayor valor

del exponente real permitido, respectivamente. A partir de ellos, puede determinarse el intervalo de números normalizados mediante las siguientes expresiones:

$$x_{\min} = 1 \cdot 2^{e_{\min}}, \qquad x_{\max} = (2 - 2^{-p}) \cdot 2^{e_{\max}}$$

donde p es la precisión, es decir, el número de bits en la parte significativa.

Formato $\langle s, e, m \rangle$	**Min. subnormal positivo** $2^{e_{\min}+1-p}$	**Min. normal positivo** $2^{e_{\min}}$	**Max. finito** $2^{e_{\max}}(2 - 2^{1-p})$
`binary16` $\langle 1, 5, 10 \rangle$	2^{-14-10} $\approx 5.96 \times 10^{-8}$	2^{-14} $\approx 6.10 \times 10^{-5}$	$(2 - 2^{-10}) \cdot 2^{15}$ $= 65504$
`binary32` $\langle 1, 8, 23 \rangle$	$2^{-126-23}$ $\approx 1.401 \times 10^{-45}$	2^{-126} $\approx 1.175 \times 10^{-38}$	$(2 - 2^{-23}) \cdot 2^{127}$ $\approx 3.403 \times 10^{38}$
`binary64` $\langle 1, 11, 52 \rangle$	$2^{-1022-52}$ $\approx 4.941 \times 10^{-324}$	2^{-1022} $\approx 2.225 \times 10^{-308}$	$(2 - 2^{-52}) \cdot 2^{1023}$ $\approx 1.798 \times 10^{308}$
`binary128` $\langle 1, 15, 112 \rangle$	$2^{-16382-112}$ $\approx 6.68 \times 10^{-4966}$	2^{-16382} $\approx 3.36 \times 10^{-4932}$	$(2 - 2^{-112}) \cdot 2^{16383}$ $\approx 1.19 \times 10^{4932}$

Tabla 2.4: Extremos del rango representable en formatos binarios (IEEE 754).

Ahora bien, no solo importa hasta qué valores puede llegar el sistema, sino también cómo se distribuyen los números representables dentro de ese rango. Para entender esta distribución, es útil introducir el concepto de *ulp* (unidad en el último lugar), que mide la distancia entre dos números consecutivos representables y fue introducido por William Kahan en los años 60. A lo largo de este apartado seguiremos la definición extendida de Goldberg, (Muller et al., 2018b), que permite definir la ulp también para valores reales:

$$\text{ulp}(x) = \beta^{\text{máx}(e, e_{\min})-p+1}$$

siendo β la base y p la precisión del formato utilizado.

La representación en coma flotante no reparte sus valores de forma uniforme: cuanto mayor es el exponente, más separados están los números. Esto se debe a que, aunque el número de combinaciones posibles de la mantisa es fijo, cada valor se escala por potencias diferentes de dos. Para ilustrarlo, si utilizamos la mantisa `1.010`, obtenemos:

- Con exponente $e = 1$: $1.010 \times 2^1 = 10.10_2 = 2.5$
- Con exponente $e = 3$: $1.010 \times 2^3 = 1010.0_2 = 10$

Si incrementamos ligeramente la mantisa a `1.011`, los valores obtenidos cambian de manera diferente dependiendo del exponente:

- Con $e = 1$: $1.011 \times 2^1 = 10.11_2 = 2.75$
- Con $e = 3$: $1.011 \times 2^3 = 1011.0_2 = 11$

El mismo cambio en la mantisa tiene un impacto mucho mayor cuando el exponente es grande. Esto refleja que cerca del cero los números están mucho más juntos, mientras que para valores grandes están más espaciados.

Este patrón de separación entre números tiene una consecuencia directa sobre la precisión relativa que el formato es capaz de mantener. A pesar de que los valores posibles se separan más conforme el exponente aumenta, el formato mantiene una precisión relativa aproximadamente constante. Esto significa que, aunque el error absoluto crezca, el número de cifras significativas que se conservan sigue siendo similar.

La precisión relativa hace referencia a cuán grande es el error comparado con el valor original. Se puede calcular como:

$$\text{Error absoluto} = |x - \hat{x}|, \qquad \text{Error relativo} = \frac{|x - \hat{x}|}{|x|}$$

Este equilibrio se mantiene incluso cuando los errores absolutos crecen. Por ejemplo:

- $x = 1.06$ se redondea a $\hat{x} = 1.1$, con un error relativo de aproximadamente 0.0377
- $x = 10.6$ se redondea a $\hat{x} = 11$, con el mismo error relativo aproximado

Este comportamiento permite que el sistema mantenga una calidad de representación uniforme, lo que justifica el uso del formato en una amplia gama de aplicaciones numéricas.

Este comportamiento no solo garantiza una precisión relativa estable, sino que también permite interpretar los errores absolutos en función del valor de la *ulp* en cada punto. De este modo, la *ulp* se convierte en una herramienta fundamental para evaluar la calidad de las aproximaciones y acotar los errores en cálculos numéricos. Para más detalles sobre este análisis, se puede consultar el *Handbook of Floating-Point Arithmetic* (Muller et al., 2018a, §2.3.3).

2.2.3. Normalización y números subnormales

En la representación en coma flotante, un mismo valor puede expresarse de varias formas desplazando el punto binario y ajustando el exponente. Para evitar ambigüedades y garantizar una única representación para cada número distinto de cero, se aplica un proceso llamado **normalización**, que consiste en ajustar el valor del exponente para que la parte significativa del número sea distinta de cero. En binario,

esto implica desplazar el punto hasta que el primer bit significativo sea igual a 1. Así, se utiliza el menor exponente posible que permite conservar la precisión:

$$x = 1.\text{xxxx} \times 2^{e}$$

Una vez hecho ese ajuste, el primer bit de la mantisa siempre será un 1 y ya no será necesario almacenarlo en memoria. Este bit se omite y se da por supuesto durante las operaciones. Esta técnica se conoce como **bit implícito**, y permite ganar un bit adicional de precisión sin aumentar el número de bits reales en memoria ya que se almacena solo la parte fraccionaria de la mantisa.

Sin embargo, cuando el número que se quiere representar es demasiado pequeño y el exponente ya está en su valor mínimo, ya no se puede seguir desplazando el punto. En estos casos se utiliza una representación especial llamada número subnormal, también llamados *denormalizados*. En este caso, el bit implícito se considera 0:

$$x = 0.\text{xxxx} \times 2^{e_{\min}}$$

Al no poder asumirse el bit implícito 1, los bits almacenados dejan de representar únicamente la parte fraccionaria y pasan a codificar la mantisa completa, comenzando por un 0. Esta reorganización reduce ligeramente la precisión respecto a los números normalizados, pero a cambio tiene una ventaja importante: extiende el rango representable hacia valores más cercanos a cero y evita saltos bruscos entre el menor número normalizado y el cero. Esto se conoce como desbordamiento gradual (gradual underflow).

En hardware, implementar los números subnormales supone un desafío extra: requieren circuitos especiales para operar correctamente sin bit implícito y con reglas distintas para la alineación de mantisas y el ajuste de exponentes. Por ejemplo, algunas operaciones como la suma o la multiplicación necesitan lógica adicional para detectar si el resultado cae en el rango subnormal, ralentizando el procesamiento.

> *"They are the most difficult type of numbers to implement in floating-point units. As a consequence, they are often implemented in software rather than in hardware."*
> (Handbook of Floating-Point Arithmetic, 2.1.2)

En muchos procesadores, el soporte completo de subnormales supone una pérdida de rendimiento significativa, por lo que se opta directamente por ignorarlos y forzar su valor a cero directamente. Algo similar ocurre con las excepciones, que en la práctica no siempre se implementan de forma estricta.

Concepto	Normalizado	Subnormal
Forma de mantisa	1.xxxxx (con bit implícito)	0.xxxxx (bit explícito)
Precisión	Máxima posible	Reducida
Exponente	Dentro del rango normal	Exponente mínimo
Objetivo	Alta precisión y eficiencia	Extender rango sin discontinuidades

Tabla 2.5: Comparación entre números normalizados y subnormales.

2.3. Estándar IEEE 754

La representación de números reales en los sistemas digitales planteaba desafíos debido a la diversidad de arquitecturas y formatos existentes. Para resolver estos problemas y asegurar la fiabilidad y la interoperabilidad entre plataformas, en 1985 el Institute of Electrical and Electronics Engineers (IEEE) publicó el *IEEE Standard for Floating-Point Arithmetic*, conocido como IEEE 754.

Este estándar estableció un conjunto de reglas que definen cómo deben representarse y operar los números reales en la memoria de los computadores, marcando un hito en la historia de la computación al unificar la representación de los números en coma flotante.

Desde su publicación inicial, el estándar ha sido revisado y actualizado en varias ocasiones. La revisión más reciente, **IEEE 754-2019**, introdujo nuevas características y mejoras, como formatos decimales adicionales y soporte para operaciones extendidas. Esta es la versión del estándar que se sigue en este trabajo.

A continuación, se presentan los conceptos fundamentales definidos en IEEE 754-2019, incluyendo los formatos de representación, los valores especiales, los modos de redondeo y el manejo de excepciones.

2.3.1. Formatos del estándar

El estándar IEEE 754 define varios formatos de punto flotante, adaptados a diferentes necesidades de rango y precisión. Cada formato especifica cómo se distribuyen los bits entre el signo, el exponente y la mantisa, determinando así la gama de números representables y su exactitud.

Los principales formatos binarios son la precisión media, la precisión simple, la precisión doble y la precisión cuádruple.

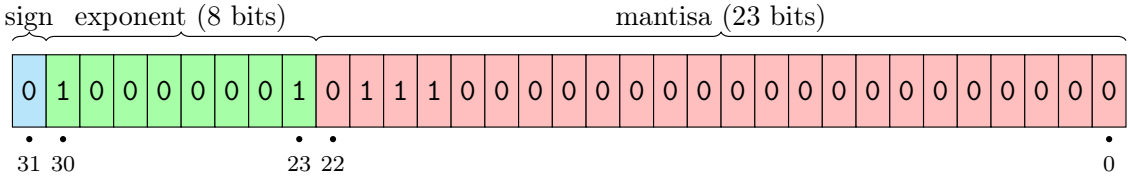

Figura 2.1: IEEE 754 (FP32) de 5.75: 0 | 10000001 | 01110000000000000000000.

- **Precisión Media**: utiliza 16 bits en total, distribuidos en 1 bit de signo, 5 bits de exponente y 10 bits de mantisa. Ofrece un equilibrio modesto entre rango y precisión, resultando útil en aplicaciones donde la eficiencia de memoria es prioritaria.
- **Precisión Simple**: emplea 32 bits, con 1 bit de signo, 8 bits de exponente y 23 bits de mantisa. Permite representar una amplia gama de valores con una precisión adecuada para la mayoría de aplicaciones de ingeniería, procesamiento de señales y simulación gráfica. Su distribución de bits es la siguiente:

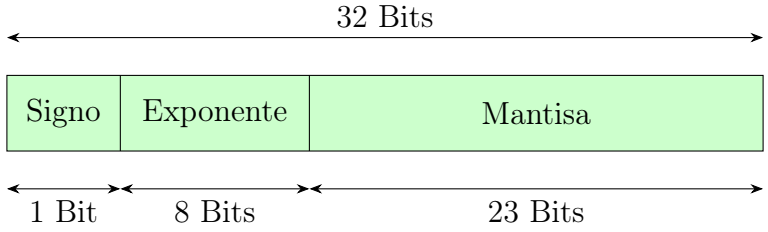

Figura 2.2: Representación en IEEE 754 con 32 bits.

- **Precisión Doble**: amplía el tamaño a 64 bits, distribuidos en 1 bit de signo, 11 bits de exponente y 52 bits de mantisa. Gracias a su mayor precisión y rango, es el preferido en cálculos científicos, simulaciones numéricas y aplicaciones de ingeniería que requieren minimizar errores acumulados. Su distribución de bits es la siguiente:

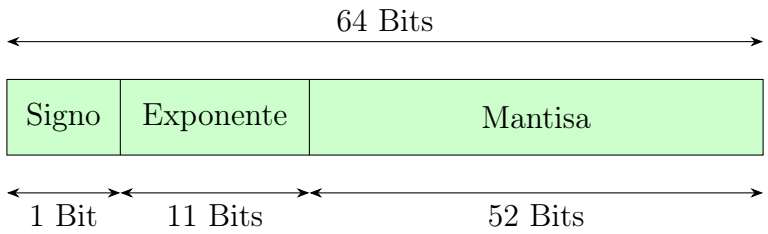

Figura 2.3: Representación en IEEE 754 con 64 bits.

- **Precisión Cuádruple**: Usa 128 bits para representar los números, con el bit de signo, 15 bits de exponente y 112 bits de mantisa. Se utiliza en tareas de alta precisión, como cálculos científicos muy sensibles o aplicaciones criptográficas avanzadas.

Con esta estructura de bits, los formatos permiten representar distintos tipos de valores:

- Números reales normalizados.
- Números subnormales.
- Ceros con signo ($+0$, -0).

- Infinitos $+\infty$ y $-\infty$
- Not a Number: Se definen dos tipos de NaNs. El primero llamado Quiet NaN (qNaN) y Signaling NaN (sNaN).

La distribución de bits entre el exponente y la mantisa responde a un compromiso fundamental: más bits en el exponente amplían el rango de números representables, mientras que más bits en la mantisa incrementan la precisión. Este equilibrio permite adaptar el formato a las necesidades específicas de rapidez, economía de espacio o exactitud.

Formato	**Exponente**	**Mantisa**	**Bias**	$e_{\max}$	$e_{\min}$
binary16	5	10	15	$+15$	-14
binary32	8	23	127	$+127$	-126
binary64	11	52	1023	$+1023$	-1022
binary128	15	112	16383	$+16383$	-16382

Tabla 2.6: Parámetros de los formatos binarios definidos en IEEE 754-2019.

Además de los formatos principales, el estándar contempla opciones más flexibles pensadas para necesidades especiales de cálculo. La **precisión extendida** (Extended Precision) se refiere a formatos de mayor tamaño, como el de 80 bits utilizado en arquitecturas x87, que ofrecen un rango y precisión superiores de forma fija y predefinida en el sistema.

Por otro lado, la **precisión extensible** (Extendable Precision) permite especificar dinámicamente tanto la precisión como el rango de los números flotantes, bajo control del usuario, adaptándose así a las exigencias concretas de cada operación. Esta idea ha sido aplicada en arquitecturas recientes como RISC-V, mediante extensiones como Xvpfloat, que permiten modificar la precisión en tiempo de ejecución sin recompilar el código y con soporte completo a nivel de hardware y software (Guthmuller et al., 2024).

2.3.2. Codificación del exponente

En el estándar IEEE 754, el exponente real de un número en punto flotante no se almacena directamente en binario con signo. En su lugar, se representa mediante un valor **sesgado** (*bias*), es decir, se le suma una constante positiva antes de codificarlo. Esta técnica desplaza todo el rango de exponentes posibles a valores no negativos.

Este mecanismo tiene una ventaja clave: el exponente puede tratarse como un número sin signo, lo que simplifica mucho el hardware. Al almacenar todos los exponentes como enteros positivos, se pueden utilizar comparadores y sumadores binarios estándar, sin preocuparse por el signo. Esto facilita la comparación entre números flotantes y las operaciones de alineación que dependen del exponente.

El sesgo depende del número de bits w del campo del exponente y se calcula mediante la fórmula:

$$\text{bias} = 2^{w-1} - 1$$

El valor almacenado es:

$$E = e + \text{bias}$$

donde e es el exponente real que aparece en la representación en punto flotante.

A partir de esta fórmula, los valores extremos que se pueden representar son:

$$e_{\min} = 1 - \text{bias}, \quad e_{\max} = \text{bias}$$

Los principales sesgos en los formatos más comunes son:

- **Precisión simple:** 8 bits de exponente, sesgo de 127.
- **Precisión doble:** 11 bits de exponente, sesgo de 1023.
- **Precisión cuádruple:** 15 bits de exponente, sesgo de 16383.

Cuando el valor del exponente almacenado alcanza ciertos patrones especiales, no se aplica el sesgo de forma normal:

- Si el exponente es todo ceros ($E = 0$), puede representar un número subnormal o un cero, dependiendo del valor de la mantisa.
- Si el exponente es todo unos ($E = 2^w - 1$), puede representar un infinito o un NaN, también según el valor de la mantisa.

El significado concreto de estos casos se explicará más adelante. Los valores intermedios, entre 1 y $2^w - 2$, se usan para codificar números normalizados aplicando el sesgo de manera habitual.

Si queremos codificar un exponente real $e = 0$ en formato de precisión simple:

$$E = 0 + 127 = 127$$

Para un exponente real $e = -2$, el exponente almacenado será:

$$E = -2 + 127 = 125$$

2.3.3. Valores especiales

En los formatos de punto flotante, algunos patrones del campo del exponente están reservados para representar valores especiales que no siguen la codificación normal de números reales. Estas codificaciones permiten representar situaciones como el cero con signo, el infinito o los resultados indefinidos (NaN), y son esenciales para garantizar un comportamiento consistente en cálculos excepcionales. Su significado exacto depende del contenido del campo del exponente y de la mantisa:

Exponente (E)	**Mantisa (M)**	**Significado**
$E = 0$	$M = 0$	$+0$ o -0
$E = 0$	$M \neq 0$	Número subnormal
$1 \leq E \leq 2^w - 2$	cualquier M	Número normalizado
$E = 2^w - 1$	$M = 0$	$+\infty$ o $-\infty$
$E = 2^w - 1$	$M \neq 0$	NaN

Tabla 2.7: Codificación de valores especiales en punto flotante.

Cero con signo

Cuando tanto el exponente como la mantisa son ceros, se representa el valor cero. El bit de signo permite distinguir entre $+0$ y -0. Aunque su valor numérico es el mismo, esta diferencia puede influir en ciertos contextos, como divisiones o límites laterales.

$$(+0) = 0\ 00000000\ 00000000000000000000000$$

Infinito

Si el exponente está formado solo por unos y la mantisa es cero, se representa un infinito. El signo diferencia entre $+\infty$ y $-\infty$. Este valor puede aparecer como resultado de una división por cero o por desbordamiento.

$$(+\infty) = 0\ 11111111\ 00000000000000000000000$$

Not a Number

Un exponente a unos y una mantisa distinta de cero indican un valor NaN. Este resultado aparece en operaciones indefinidas como $0/0$, $\infty - \infty$, o $\sqrt{-1}$. Dentro de los NaN, se distinguen dos tipos principales:

- **Quiet NaN (qNaN):** Indica que una operación ha producido un resultado indefinido pero no genera una excepción. Se utiliza para propagar errores en

cálculos posteriores. El bit más significativo de la mantisa es 1.

$$\text{qNaN} = S\ 11111111\ 1xxxxxxxxxxxxxxxxxxxxx$$

- **Signaling NaN (sNaN):** utilizado para señalar que una operación inválida ha ocurrido, generando una excepción que puede ser manejada por el sistema. El bit más significativo de la mantisa es 0, y debe haber al menos otro bit distinto de cero para que se distinga del infinito.

$$\text{sNaN} = S\ 11111111\ 0xxxxxxxxxxxxxxxxxxxxx$$

El contenido de la mantisa puede codificar un *payload*, es decir, información adicional específica del sistema o del compilador, que puede contener información diagnóstica sobre el origen del error. Su interpretación no está normalizada y depende de la implementación. El payload puede conservarse o modificarse durante conversiones o propagaciones, según las reglas del sistema.

La codificación detallada de estos valores especiales, así como las reglas que rigen su interpretación y comportamiento en las operaciones, puede encontrarse en la Sección 3.4 del estándar IEEE 754-2019 (IEEE Computer Society, 2019).

2.3.4. Modos de redondeo

Dado que el resultado exacto de una operación en punto flotante no siempre puede representarse con precisión en el formato disponible, se requiere un mecanismo para decidir cómo aproximarlo. Los modos de redondeo definen la estrategia seguida para seleccionar la representación final.

Los modos se agrupan en dos categorías generales según el estándar: redondeo al valor más cercano y redondeo dirigido.

Redondeo al más cercano

En este grupo, se elige el valor representable más próximo al resultado exacto. En caso de empate entre dos posibles representaciones, la decisión depende del modo seleccionado:

- **RoundTiesToEven (al par):** Es el modo por defecto en formatos binarios. Si el valor exacto está a igual distancia entre dos representaciones, se selecciona aquella cuyo dígito menos significativo sea par. Esta estrategia minimiza el sesgo acumulado en operaciones sucesivas.
- **RoundTiesToAway (hacia el más alejado):** En caso de empate, se elige la representación con mayor magnitud (más alejada de cero). Este modo es obligatorio en formatos decimales y opcional en binario. Puede ser útil cuando se desea evitar subestimaciones sistemáticas.

Redondeo dirigido

Estos modos fuerzan el redondeo hacia una dirección fija, independientemente del valor más cercano. Son especialmente útiles en cálculos con errores acotados, análisis numérico o aritmética de intervalos.

- **RoundTowardZero (truncamiento):** Redondea el resultado hacia cero. Es el comportamiento habitual en conversiones de punto flotante a entero en muchas arquitecturas.
- **RoundTowardPositive (hacia $+\infty$):** Siempre redondea hacia valores mayores o iguales al resultado exacto. Permite obtener una cota superior fiable.
- **RoundTowardNegative (hacia $-\infty$):** Redondea hacia valores menores o iguales. Proporciona una cota inferior garantizada.

Según la Sección 4.3.3 del estándar IEEE 754-2019, todas las implementaciones deben ofrecer al menos el modo `RoundTiesToEven` y los tres dirigidos. En cambio, `RoundTiesToAway` solo es obligatorio en formatos decimales. En binario, su implementación es opcional.

El modo por defecto es `RoundTiesToEven` para formatos binarios. En formatos decimales, el modo por defecto depende del lenguaje de programación o la plataforma, aunque también se recomienda el redondeo al par por su neutralidad estadística.

2.3.5. Manejo de excepciones

Durante el cálculo en punto flotante pueden surgir condiciones anómalas que impiden obtener un resultado exacto, válido o representable. El sistema de excepciones proporciona una forma estructurada de detectar, señalar y registrar estas situaciones sin interrumpir el flujo normal del programa. Esta capacidad resulta esencial en aplicaciones donde la estabilidad numérica es prioritaria.

El estándar define cinco tipos básicos de excepciones, que pueden activarse de forma independiente durante una operación. Cada una responde a una condición específica:

- **Operación inválida**: Se activa cuando una operación no está matemáticamente definida o involucra operandos no válidos. También ocurre si se utiliza un NaN señalizado (sNaN), lo que transforma el resultado en un quiet NaN (qNaN). Casos típicos:

  $$0/0, \quad \infty - \infty, \quad \sqrt{-1}$$

  También se activa al intentar convertir un NaN a entero o al comparar dos NaNs.

- **División por cero**: Se activa cuando un número finito se divide por cero. En lugar de provocar un error, se produce un infinito con el signo adecuado y se activa su correspondiente flag:

$$\frac{5.0}{+0} = +\infty, \quad \frac{-3.0}{-0} = +\infty$$

- **Overflow**: Se activa cuando el valor calculado excede el mayor número representable. El comportamiento depende del modo de redondeo y del valor intermedio antes de redondear:

  - *RoundTiesToEven* y *RoundTiesToAway*: se redondea a $+\infty$ o $-\infty$ según el signo.
  - *RoundTowardZero*: se devuelve el mayor número finito representable en dirección a cero.
  - *RoundTowardPositive*: si el resultado es positivo, se devuelve $+\infty$; si es negativo, el mayor número finito negativo.
  - *RoundTowardNegative*: si es negativo, se devuelve $-\infty$; si es positivo, el mayor número finito positivo.

  En todos los casos, se activan las flags de overflow e inexactitud.

$$1.0 \times 10^{308} \times 10.0 = +\infty^{1}$$

- **Underflow**: Se activa cuando el resultado cae por debajo del rango normalizado y, además, la operación es inexacta. El valor resultante puede ser un número subnormal o cero, dependiendo del modo de redondeo y del signo:

  - *RoundTowardZero*:devuelve un subnormal cercano a cero con el mismo signo del resultado exacto.
  - *RoundTowardPositive*: devuelve cero si el resultado exacto es negativo, o el menor subnormal positivo si es positivo.
  - *RoundTowardNegative*: devuelve cero si el resultado exacto es positivo, o el menor subnormal negativo si es negativo.
  - *RoundTiesToEven* y *RoundTiesToAway*: pueden devolver cero o un subnormal, dependiendo del valor exacto del resultado intermedio.

  El estándar introduce el concepto de *tininess*, que define cuándo se considera que un resultado cae fuera del rango normalizado. Esta condición puede evaluarse antes o después del redondeo, a elección del implementador y siempre se debe mantener el mismo criterio en todas las operaciones. La excepción de underflow solo se activa si, además de detectarse tininess, el resultado final es inexacto, en cuyo caso también se debe señalizar la excepción *inexact*.

  Este caso es especialmente importante en operaciones que generan resultados muy pequeños, donde mantener la mayor precisión posible puede ser crítico:

$$\text{Resultado} \approx \text{subnormal (con menor precisión) o } 0$$

[1] El valor resultante, 1.0×10^{309}, supera el máximo finito representable en el formato `binary64`, que es aproximadamente 1.798×10^{308}, lo que provoca una excepción por *overflow*.

- **Inexactitud**: Se activa cuando el resultado exacto no puede representarse con precisión y requiere redondeo:

$$\frac{1}{3} \approx 0.333\ldots$$

  Es una de las excepciones más frecuentes y suele aparecer incluso en operaciones rutinarias.

El estándar adopta un modelo *non-stop*: las excepciones no interrumpen la ejecución del programa. En su lugar:

- Se devuelve un valor definido (NaN, infinito, cero o subnormal).
- Se registra la excepción mediante un flag.
- La ejecución continúa normalmente.

Cada tipo de excepción está asociado a una *flag* (indicador de estado), que puede ser leída, escrita o borrada por el programador. Estas flags permiten comprobar si se han producido errores durante una operación sin necesidad de gestionarlos en el momento.

- Las flags son **persistentes**: permanecen activadas hasta que se limpian manualmente.
- Son **acumulativas**: si una excepción ocurre varias veces, la flag se mantiene activada.
- No provocan interrupciones, salvo que el entorno esté configurado para ello.

El manejo de excepciones puede variar entre plataformas y lenguajes. Algunos entornos permiten establecer traps o handlers para capturarlas y tratarlas durante la ejecución, mientras que otros ofrecen funciones para consultar las flags tras una operación. Esta flexibilidad permite adaptar el comportamiento del sistema a las necesidades específicas de precisión, rendimiento o depuración.

2.4. Operaciones en Punto Flotante

2.4.1. Codificación y decodificación

Antes de realizar operaciones aritméticas, los operandos deben convertirse desde su representación real al formato binario IEEE 754 mediante codificación. Una vez completada la operación aritmética correspondiente, el resultado obtenido se reconvierte en su representación real mediante la decodificación. Estos procesos garantizan interoperabilidad y coherencia en el tratamiento de datos.

La representación codificada se define formalmente como una tripleta:

$$r = (S, E, T)$$

donde se indican los bits que ocuparán el signo, el campo exponente y la mantisa sin el bit implícito. Estos componentes se distribuyen a lo largo de k bits totales, y sus dimensiones vienen determinadas por los siguientes parámetros:

- k: anchura total del número representado.
- p: número de bits de precisión, incluyendo el bit implícito.
- w: número de bits asignado al exponente.
- $t = p - 1$: número de bits del campo fracción (mantisa almacenada).
- bias $= 2^{w-1} - 1$: valor de sesgo aplicado al exponente real.
- $e_{\max} = 2^{w-1} - 1$, $e_{\min} = 1 - e_{\max}$: límites del exponente real permitidos.

Como ya se ha explicado en la Sección 2.3.2 , la interpretación de los campos E y T permite diferenciar entre varios tipos de números representables. A continuación se presentan las condiciones establecidas por el estándar para representar cada tipo y como se obtiene el tamaño de cada componente de la tripleta codificada:

- **Si** $E = 2^w - 1$ **y** $T \neq 0$**:** Se representa un NaN. El bit más significativo de T indica si es un qNaN (1) o un sNaN (0).
- **Si** $E = 2^w - 1$ **y** $T = 0$**:** se representa un valor infinito con el signo indicado por S.

$$r = (-1)^S \cdot (+\infty)$$

- **Si** $1 \leq E \leq 2^w - 2$**:** Se representa un número normalizado cuyos campos tienen los siguientes tamaños:

$$r = (S,\ E - \text{bias},\ (1 + 2^{1-p} \cdot T))$$

- **Si** $E = 0$ **y** $T \neq 0$**:** Se representa un número subnormal:

$$r = (S,\ e_{min},\ (0 + 2^{1-p} \cdot T))$$

- **Si** $E = 0$ **y** $T = 0$**:** se representa el cero con signo. El exponente y la mantisa se interpretan como nulos.

$$r = (S, e_{min}, 0)$$

Una vez determinado el tipo de número representado, se aplica un flujo de pasos sistemático para obtener los valores concretos de cada campo de la tripleta:

1. **Identificación del signo**: Se analiza el número original y se asigna el bit S como 0 si es positivo, o 1 si es negativo.
2. **Conversión a valor absoluto y binario**: El número se convierte a valor absoluto y se expresa en base 2.
3. **Normalización**: Si el número lo permite, se reescribe en forma $1.xxxxx \cdot 2^e$, desplazando la coma binaria.
4. **Cálculo del exponente sesgado**: Se determina $E = e + \text{bias}$.
5. **Codificación de la mantisa**: Se almacena T, con los $p - 1$ bits de la parte fraccionaria.
6. **Asignación de categoría especial**: Si aplica, se ajustan E y T para representar NaN, infinito, cero o subnormal.

Una vez hecho el cálculo, la decodificación invierte este proceso para reconstruir el valor real a partir de los campos almacenados.

- Se resta el sesgo al exponente sesgado E si el valor es normal.
- Se restaura el bit implícito correspondiente.
- Se reconstruye la mantisa y se aplica la potencia de 2 correspondiente al exponente.

En los siguientes apartados se detallarán las fases específicas de cada operación aritmética, partiendo siempre de la representación codificada descrita aquí y finalizando con su correspondiente decodificación.

2.4.2. Suma y resta

La suma y la resta en punto flotante son operaciones básicas pero fundamentales. Su correcta implementación es esencial para garantizar la precisión de los cálculos y evitar errores sutiles en programas numéricos. Aunque el resultado final pueda parecer sencillo, el proceso interno es más complejo, ya que sigue varias fases descritas a continuación.

Fases del proceso:

- **Alineación de exponentes:** Si los operandos tienen exponentes distintos, se desplaza la mantisa del que tenga menor exponente hacia la derecha, hasta igualarlos. Esto puede provocar pérdida de bits significativos, sobre todo si la diferencia entre exponentes es grande. Sin embargo, si los operandos son

similares en magnitud, la operación puede ser exacta. Esto lo recoge el *lema de Sterbenz*, que afirma que si $\frac{1}{2} \leq \frac{x}{y} \leq 2$, entonces $x - y$ se representa sin error de redondeo.

- **Operación sobre mantisas:** Una vez alineadas, se suman o restan las mantisas, dependiendo del signo de los operandos. Si el signo es igual, se suman. Si es distinto, se restan. Cuando los números implicados son muy parecidos, las cifras significativas pueden anularse entre sí, dejando un resultado pequeño con menos bits útiles. Esta situación, conocida como cancelación, es una fuente habitual de pérdida de precisión.
- **Normalización y redondeo:** Si el resultado no está en forma normalizada tras la operación, se ajusta la mantisa y el exponente. Si el valor resultante es demasiado pequeño para representarse como normal, se convierte en subnormal. Después se aplica el redondeo, que puede introducir error y activar el flag `inexact`.

Flags que pueden activarse:

- `inexact`, si el resultado no puede representarse exactamente.
- `underflow`, si el resultado final es subnormal e inexacto.
- `invalid`, si se suman infinitos de signo opuesto.

Además de los casos habituales, existen combinaciones especiales de operandos que siguen un comportamiento normativo. La siguiente tabla resume el resultado esperado al operar dos valores positivos con diferentes categorías: cero, normal o subnormal, infinito o NaN.

$\lvert x\rvert$ \ $\lvert y\rvert$	$+0$	(sub)normal	$+\infty$	NaN
$+0$	$+0$	$\lvert y\rvert$	$+\infty$	qNaN
(sub)normal	$\lvert x\rvert$	$\circ(\lvert x\rvert \pm \lvert y\rvert)$	$+\infty$	qNaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	qNaN
NaN	qNaN	qNaN	qNaN	qNaN

Tabla 2.8: Resultado de $|x| \pm |y|$ para diferentes combinaciones de operandos positivos.

Nota: El símbolo $\circ(\cdot)$ indica que el resultado ha sido redondeado según el modo de redondeo vigente.

Fórmulas generales:

- **Suma:**

$$z = x + y = (-1)^{s_x} \cdot |x| + (-1)^{s_y} \cdot |y|$$

- **Resta:**

$$z = x - y = (-1)^{s_x} \cdot |x| - (-1)^{s_y} \cdot |y|$$

Estas fórmulas unifican la descripción de la operación, sean cuales sean los signos de los operandos. A diferencia de la aritmética real exacta, la suma y la resta en punto flotante no cumplen en general las propiedades asociativa ni distributiva. Esto se debe a que, al aplicarse redondeo tras cada operación, el orden en que se evalúan las expresiones puede cambiar el resultado. Por ejemplo, el valor de $(x+y)+z$ puede no coincidir con $x+(y+z)$ si las magnitudes de los operandos difieren, siendo estas desviaciones en parte una consecuencia inevitable de trabajar con una precisión limitada.

2.4.3. Multiplicación

La multiplicación de números en punto flotante consiste en operar por separado los tres componentes de cada operando: los signos, las mantisas y los exponentes. La operación sigue las siguientes fases:

- **Signo del resultado:** se obtiene con el operador XOR sobre los signos de los operandos. El resultado será negativo si exactamente uno de los operandos es negativo.
- **Producto de mantisas:** Las mantisas de los dos operandos se multiplican directamente. El producto resultante puede tener el doble de bits significativos que la mantisa de los operandos originales, lo que requiere una normalización posterior para ajustarlo al formato estándar.
- **Suma de exponentes:** Se suman los exponentes reales de ambos operandos. El exponente sesgado del resultado se ajusta posteriormente.
- **Normalización y redondeo:** Si el producto de las mantisas no está normalizado, se normaliza y se ajusta el exponente. Finalmente, se aplica el modo de redondeo activo.

Fórmula general:

$$x \times y = (-1)^{s_x \oplus s_y} \cdot (M_x \cdot M_y) \cdot 2^{E_x + E_y}$$

La multiplicación puede dar lugar a varias excepciones:

- **invalid**: si uno de los operandos es 0 y el otro es ∞, la operación se considera indefinida. El resultado es qNaN.
- **overflow**: si el resultado, tras el redondeo, supera el valor finito representable más grande.

- **underflow**: si el resultado cae por debajo del rango normalizado y además es inexacto.
- **inexact**: si el resultado no puede representarse exactamente y requiere redondeo.

El comportamiento con operandos especiales está estandarizado y se resume en la siguiente tabla:

$\lvert x\rvert$ \ $\lvert y\rvert$	+0	(sub)normal	$+\infty$	NaN
+0	+0	+0	qNaN	qNaN
(sub)normal	+0	$\circ(\lvert x\rvert \times \lvert y\rvert)$	$+\infty$	qNaN
$+\infty$	qNaN	$+\infty$	$+\infty$	qNaN
NaN	qNaN	qNaN	qNaN	qNaN

Tabla 2.9: Resultado de $|x| \times |y|$ para combinaciones de operandos positivos.

Nota: El símbolo $\circ(\cdot)$ indica que el resultado ha sido redondeado según el modo activo.

2.4.4. División

La división en punto flotante implica dividir las mantisas de los operandos y restar sus exponentes. Aunque el procedimiento general es directo, esta operación tiene ciertas particularidades que conviene tener en cuenta, especialmente en el manejo de subnormales, el redondeo y los casos excepcionales.

- **División de mantisas:** se calcula el cociente entre las mantisas M_x y M_y. Este valor puede tener más precisión de la que permite el formato de destino, por lo que puede ser necesario un redondeo posterior. Si alguna de las entradas es subnormal, lo habitual es normalizarla previamente para garantizar una precisión adecuada antes de dividir.
- **Resta de exponentes:** se calcula el exponente del resultado como $E_x - E_y$. Si la división genera un resultado que requiere ser normalizado, este valor también se ajusta en consecuencia.
- **Normalización y redondeo:** como en otras operaciones, el cociente puede no estar en forma normalizada. El redondeo se realiza tras el cálculo del cociente, pudiendo activar la excepción `inexact`.

Fórmula general:

$$z = \frac{x}{y} = (-1)^{s_x \oplus s_y} \cdot \left(\frac{M_x}{M_y}\right) \cdot 2^{E_x - E_y}$$

Esta fórmula expresa el resultado como el producto de tres componentes: el signo (calculado con el XOR de los signos de los operandos), el cociente de sus mantisas y la potencia de dos correspondiente a la diferencia entre sus exponentes.

Existen múltiples criterios para clasificar los algoritmos de división en punto flotante. Algunos autores los agrupan en algoritmos *multiplicativos* y *sustractivos*, según el tipo de operación aritmética predominante en su implementación (Soderquist y Leeser, 1997). Otros proponen una clasificación más detallada basada en las estructuras hardware necesarias, dividiéndolos en *digit recurrence*, *functional iteration*, *very high radix*, *lookup-table* y *variable latency* (Obermann y Flynn, 1997). En este trabajo se adopta una clasificación basada en el principio matemático del algoritmo (Muller et al., 2018a, §7.6.2), ya que permite agrupar bajo un mismo enfoque categorías que en otras clasificaciones se consideran independientes:

- **Digit-recurrence:** Este método sigue el mismo principio que la división manual: en cada iteración se calcula una cifra del cociente estimando cuántas veces cabe el divisor en el dividendo parcial. El proceso se repite hasta alcanzar la precisión deseada. Este tipo de algoritmo es muy común en hardware por su eficiencia y porque el cálculo intermedio se mantiene exacto en cada paso, lo que facilita el redondeo final. Dentro de esta categoría se encuentran variantes históricas como los métodos *restoring* y *non-restoring* en base 2, así como el algoritmo SRT (Sweeney-Robertson-Tocher), que introduce redundancia y lookup tables en la lógica de selección del cociente y permite operar en distintas bases. También existe la variante *very-high radix*, que son aquellos algoritmos que obtienen diez o más bits del cociente por iteración. Reducen significativamente el número de ciclos necesarios, a costa de una mayor complejidad hardware. Aunque menos comunes, estas implementaciones se utilizan en procesadores de altas prestaciones, donde la latencia mínima es prioritaria (Bruguera, 2018).

- **Algoritmo iterativo:** Este método transforma la división en una multiplicación, calculando una aproximación al cociente mediante la identidad:

  $$x/y \approx x \cdot (1/y)$$

  La estimación inicial de $1/y$ se toma de una lookup table que utiliza los bits más significativos del divisor para seleccionar la entrada con el valor aproximado. A partir de ahí, se mejora la precisión mediante multiplicaciones sucesivas. Este enfoque requiere menos pasos que los de la categoría anterior, tiene un coste logarítmico en función de la precisión deseada, pero cada iteración es más costosa. Además, como se redondea en cada etapa, es importante controlar el error acumulado para garantizar un resultado final correcto. A esta clase pertenecen algoritmos como *Newton–Raphson*, *Goldschmidt*, o *series de Taylor* (Patankar y Koel, 2021), todos ellos basados en sucesivas aproximaciones del recíproco. Este enfoque fue adoptado en el AMD-K7, que emplea una variante de Goldschmidt con precisión extendida de 68 bits (Oberman, 1999).

- **Aproximación polinómica:** Este enfoque utiliza un polinomio previamente calculado para aproximar $1/y$ o directamente x/y, evaluado mediante una se-

cuencia fija de operaciones sin necesidad de iteraciones, generalmente a través del esquema de Horner. Este tipo de aproximaciones también puede emplearse como punto de partida en algoritmos iterativos, proporcionando la estimación inicial del recíproco. Aunque la precisión es menor que en los métodos anteriores, permite realizar divisiones rápidas con latencia constante, lo que resulta útil en aplicaciones donde el rendimiento es más importante que la exactitud total, como DSPs embebidos o shaders de GPU. Los polinomios se construyen generalmente mediante técnicas como Taylor, Chebyshev o el algoritmo de Remez.

En todos los casos, el objetivo es obtener un resultado lo más preciso posible antes del redondeo final. Una descripción detallada puede encontrarse en el *Handbook of Floating-Point Arithmetic* (Muller et al., 2018a, §7.6.2).

La división puede dar lugar a las siguientes excepciones:

- `divideByZero`: si el divisor es cero y el numerador es finito, se devuelve $\pm\infty$.
- `invalid`: si se realiza $0/0$ o ∞/∞, se devuelve qNaN.
- `inexact`: si el cociente no puede representarse exactamente tras el redondeo.

A continuación se muestra el resultado esperado al dividir operandos positivos de distintas categorías:

$\lvert x\rvert$ \ $\lvert y\rvert$	$+0$	(sub)normal	$+\infty$	NaN
$+0$	qNaN	$+0$	$+0$	qNaN
(sub)normal	$+\infty$	$\circ(\lvert x\rvert/\lvert y\rvert)$	$+0$	qNaN
$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
NaN	qNaN	qNaN	qNaN	qNaN

Tabla 2.10: Resultado de $|x|/|y|$ para diferentes combinaciones de operandos positivos.

Nota: el símbolo $\circ(x/y)$ indica que el resultado se redondea según el modo vigente.

2.4.5. Raíz cuadrada

La raíz cuadrada en punto flotante se aplica a un único operando y está definida únicamente para valores no negativos. Aunque su complejidad es menor que la de la división, requiere un manejo cuidadoso del redondeo y de los casos especiales. A diferencia de otras operaciones, no puede producir ni overflow ni underflow.

Fases del proceso:

- **Signo y validez:** la operación solo está definida para operandos positivos o cero. Si el valor de entrada es negativo y finito, se considera inválido y se devuelve un NaN silencioso.
- **Cálculo de la mantisa:** se evalúa la raíz cuadrada de la mantisa mediante métodos iterativos o mediante aproximaciones polinómicas. El resultado intermedio suele tener más precisión que el formato de destino, por lo que se redondea posteriormente.
- **Ajuste del exponente:** se calcula dividiendo entre dos el exponente del operando. Si el valor es impar, se ajusta la mantisa para conservar la normalización del resultado.
- **Normalización y redondeo:** si el valor resultante no está en forma normalizada, se desplaza la mantisa y se ajusta el exponente hasta alcanzar la normalización. Luego se aplica el redondeo según el modo activo, lo que puede activar la excepción correspondiente.

Fórmula general:

$$\sqrt{x} = (-1)^{s_x} \cdot \sqrt{M_x} \cdot 2^{E_x/2}$$

Esta fórmula solo es válida si el operando es positivo o cero.

Los algoritmos utilizados para implementar la raíz cuadrada en punto flotante son los mismos que se emplean en la división, adaptados a la forma específica de esta operación. Cada uno presenta un equilibrio diferente entre coste computacional, precisión y facilidad de implementación:

Los algoritmos utilizados para implementar la raíz cuadrada en punto flotante son, en esencia, los mismos que se emplean en la división, adaptados a la forma específica de esta operación. De hecho, ambas operaciones suelen compartir la misma unidad funcional ya que requieren recursos similares. Cada enfoque presenta un equilibrio distinto entre coste computacional, precisión y facilidad de implementación:

- **Digit-recurrence:** este método calcula la raíz cuadrada bit a bit, generando un dígito del resultado en cada iteración, comenzando por los más significativos. En cada paso se estima si añadir un 1 o un 0 en la posición actual mejora la aproximación, en función del resto parcial acumulado. Se trata de una técnica similar a la división binaria clásica, pero adaptada a la estructura de la raíz cuadrada. Como solo requiere operaciones sencillas (sumas, restas y desplazamientos), es habitual en implementaciones hardware con recursos limitados, aunque su lógica de control es algo más compleja que en la división.
- **Algoritmo iterativo:** al igual que en la división, se parte de una estimación inicial que se refina en cada iteración. La diferencia principal es que aquí no se calcula un inverso, sino directamente un valor cuya raíz cuadrada se aproxime al operando, resolviendo la ecuación $y^2 = x$. El método más habitual es Newton-Raphson, que ajusta el valor en cada iteración para reducir el error.

Aunque la estructura de las iteraciones es similar a la de la división, los errores se amplifican al elevar el valor al cuadrado, por lo que suele requerirse mayor precisión intermedia en las últimas etapas.

- **Aproximación polinómica:** como en la división, se utiliza un polinomio previamente calculado que permite obtener una aproximación de la raíz cuadrada sin necesidad de iteraciones. El polinomio se construye de forma que su salida se acerque lo máximo posible al valor de $\sqrt{x}$ dentro de un intervalo concreto. Como la función raíz cuadrada cambia más rápidamente en ciertos tramos —especialmente cerca de cero—, es necesario limitar el rango de entrada o transformarlo antes de aplicar el polinomio, para mantener el error bajo control.

Durante el cálculo, pueden activarse las siguientes excepciones:

- **invalid:** si el operando es negativo y finito, el resultado es un NaN silencioso.
- **inexact:** si el resultado no puede representarse exactamente y debe redondearse.

Operando x	$+0$	$+\infty$	-0	< 0	NaN
Resultado r	$+0$	$+\infty$	-0	qNaN	qNaN

Tabla 2.11: Resultado de $\sqrt{x}$ para operandos especiales.

Nota: la operación solo está definida para valores no negativos.

2.4.6. Fused Multiply-Add

La operación Fused Multiply-Add (FMA) combina una multiplicación y una suma en un único paso, sin redondeo intermedio entre ambas. Esto permite conservar mayor precisión que si las dos operaciones se realizan por separado, lo que resulta especialmente útil en cálculos repetidos o acumulativos. La operación evalúa la expresión $x \times y + z$ de forma exacta hasta el redondeo final, siguiendo el comportamiento definido por el estándar.

Fases del proceso:

- **Multiplicación y alineación:** se calcula el producto de las mantisas de los dos primeros operandos mientras, en paralelo, se alinea el tercer operando si los exponentes difieren, desplazando su mantisa y condensando los bits descartados en un sticky bit. El producto se genera en formato carry-save para su posterior suma.
- **Suma del producto con el tercer operando:** el producto se suma directamente con el operando alineado, sin aplicar redondeo intermedio. Dependiendo del signo de los valores, esta operación puede ser una suma o una resta efectiva.

- **Normalización y redondeo:** el resultado se normaliza si es necesario y se redondea según el modo activo, garantizando una única fuente de error numérico.

Fórmula general:

$$\text{FMA}(x, y, z) = (x \times y) + z = (-1)^{s_x \oplus s_y} \cdot (M_x \cdot M_y + M_z) \cdot 2^{E_x + E_y}$$

Durante el cálculo, pueden activarse las siguientes excepciones:

- **invalid:** si se produce una operación indefinida, como $\infty \times 0 + \text{NaN}$, el resultado es qNaN.
- **overflow:** si el resultado excede el valor máximo representable tras el redondeo.
- **underflow:** si el resultado cae por debajo del mínimo normalizado y además es inexacto.
- **inexact:** si el resultado no puede representarse exactamente y debe redondearse.

FMA se utiliza habitualmente en expresiones del tipo $a = a + b \times c$, donde realizar ambas operaciones de forma combinada permite reducir los errores acumulados en cada iteración. Por esta razón, es especialmente valiosa en algoritmos de álgebra lineal, gráficos por ordenador o cálculos numéricos de alta precisión.

2.4.7. Conversiones entre formatos

Las operaciones de conversión en punto flotante permiten transformar valores numéricos entre distintos tipos de representación, como entre formatos de diferente precisión o entre enteros y punto flotante. Estas conversiones son esenciales para adaptar los datos al formato requerido en cada etapa del cálculo, especialmente cuando se trabaja con arquitecturas mixtas o tipos numéricos heterogéneos.

Las tres formas más habituales de conversión son:

- **ConvertFormat**: convierte un número entre dos formatos binarios de punto flotante, como de precisión simple a doble. Si el formato de destino tiene igual o mayor precisión, la conversión es exacta. Si tiene menor precisión, el valor se redondea según el modo activo.
- **ConvertFromInt**: transforma un valor entero con o sin signo a punto flotante. La conversión es exacta si el entero cabe dentro del rango representable del formato destino. En caso contrario, se aplica redondeo o puede generarse una excepción.

- **ConvertToInt**: convierte un número en punto flotante a un valor entero. Si el número tiene parte fraccionaria, se elimina o se redondea según el modo de redondeo configurado. Si el valor está fuera del rango del tipo entero, se genera una excepción. Existen varias versiones con modos de redondeo explícitos, como `TowardZero` o `TiesToEven`, así como variantes que detectan si la conversión ha sido exacta.

Durante estas conversiones pueden activarse las siguientes excepciones:

- `invalid`, si el valor no puede convertirse (por ejemplo, NaN, infinito o número fuera de rango).
- `inexact`, si la conversión requiere redondeo porque el resultado no puede representarse exactamente.
- `overflow`, si un entero se convierte a un número flotante que excede el rango del formato destino.

Además, algunos comportamientos están definidos de forma precisa: los ceros enteros sin signo se convierten a +0; los NaN señalizantes generan una excepción; y los infinitos solo pueden convertirse a enteros si están dentro del rango representable, lo cual normalmente no ocurre.

Cuando la conversión es exacta, el valor numérico permanece inalterado. Sin embargo, al convertir a un formato de menor precisión o a un tipo entero, puede perderse información por redondeo o saturación. El redondeo aplicado depende del modo de redondeo activo en el sistema.

2.4.8. Operaciones de comparación

Las operaciones de comparación permiten evaluar relaciones entre dos valores en punto flotante. Estas relaciones incluyen igualdad, desigualdad y comparaciones relacionales como mayor o menor. Las comparaciones pueden realizarse entre operandos del mismo formato o entre diferentes formatos, siempre que compartan la misma base.

Las comparaciones se dividen en dos grandes categorías:

- **Igualdad y desigualdad**
  - **Igualdad (==)**: devuelve verdadero si los dos operandos tienen exactamente el mismo valor en punto flotante, incluyendo el signo.
  - **Desigualdad (!=)**: se cumple si los operandos son distintos o si alguno de ellos es NaN.
- **Comparaciones relacionales**

- **Menor que** ($<$): verdadero si el primer operando es estrictamente menor que el segundo.
- **Menor o igual** ($<=$): verdadero si el primer operando es menor o igual que el segundo.
- **Mayor que** ($>$): verdadero si el primer operando es estrictamente mayor que el segundo.
- **Mayor o igual** ($>=$): verdadero si el primer operando es mayor o igual que el segundo.

Estas operaciones pueden clasificarse además según su comportamiento ante operandos NaN:

- **Quiet comparisons**: no generan excepción cuando uno de los operandos es NaN. Simplemente devuelven falso para cualquier comparación relacional y verdadero para "no igual".
- **Signaling comparisons**: generan una excepción si algún operando es NaN. Se utilizan cuando se desea detectar explícitamente la presencia de datos inválidos.

El estándar define un conjunto de condiciones que se consideran "verdaderas" al evaluar una comparación. Estas condiciones, llamadas *true relations*, permiten expresar de forma clara tanto las comparaciones directas como sus negaciones. La siguiente tabla resume sus combinaciones posibles y los nombres que reciben:

True Relation	**Nombre**	**Negación**	**Nombre**
EQ	compareEqual	LT, GT	compareNotEqual
GT	compareGreater	EQ, LT	compareNotGreater
GT, EQ	compareGreaterEqual	LT	compareNotGreaterEqual
LT	compareLess	EQ, GT	compareNotLess
LT, EQ	compareLessEqual	GT	compareNotLessEqual

Tabla 2.12: Relaciones verdaderas y sus negaciones en IEEE 754.

Casos especiales:

- Cualquier comparación con un NaN como operando devuelve falso, salvo "no igual", que devuelve verdadero en comparaciones silenciosas.
- Los ceros con signo contrario (+0 y -0) se consideran iguales.
- El estándar también define una relación total, `totalOrder`, que permite ordenar todos los valores, incluidos los NaNs, de forma coherente. Esta operación se usa en contextos donde se requiere una comparación determinista entre cualquier par de valores.

Capítulo 3

Estado del Arte

Este capítulo examina las principales unidades funcionales de punto flotante (FPUs) de código abierto disponibles en la actualidad y establece las bases para justificar la selección del núcleo que será empleado en este trabajo.

La evaluación se organiza en dos bloques:

1. Una **revisión cualitativa** de las arquitecturas más representativas en entornos académicos, fundamentalmente.
2. Una **comparativa cuantitativa preliminar**, centrada en las prestaciones obtenidas mediante síntesis sobre una *FPGA Xilinx Alveo U200*.

Los resultados de este análisis orientan la elección de los módulos de partida y permiten contextualizar su posterior integración y optimización.

3.1. Motivación tecnológica

Las operaciones en coma flotante son esenciales en aplicaciones de computación intensiva, gráficos, aprendizaje automático y, cada vez más, en sistemas embebidos de precisión mixta. Contar con una FPU dedicada en hardware permite:

- Reducir significativamente la latencia frente a soluciones software.
- Disminuir el consumo energético por operación.
- Cumplir estrictamente con el estándar IEEE, lo que garantiza resultados portables y reproducibles.

Existen múltiples núcleos hardware de código abierto que implementan este tipo de operaciones. Sin embargo, presentan diferencias importantes en arquitectura, compatibilidad con IEEE 754-2019, profundidad de *pipeline*, configurabilidad, consumo de recursos y licencia. Este capítulo compara cuatro candidatas: **HardFloat**, **FPnew/CVFPU**, **VFloat** y **OpenCores-FPU**.

3.2. Metodología de comparación

Para comparar distintas unidades de punto flotante de forma justa y reproducible, se ha definido un procedimiento común de síntesis e implementación. Este proceso permite extraer métricas representativas del área, la frecuencia máxima alcanzable y el rendimiento operativo de cada núcleo.

El análisis se ha estructurado en cinco aspectos: la plataforma hardware utilizada, la herramienta de síntesis y su configuración, las restricciones de temporización aplicadas, el conjunto de métricas comparadas y la procedencia de los datos obtenidos.

3.2.1. Plataforma objetivo

Todas las síntesis se han realizado sobre la tarjeta *Xilinx Alveo U200*, equipada con un FPGA UltraScale+ de la familia `xcu200-fsgd2104-2-e`. Este dispositivo cuenta con:

- 1.182.000 LUTs,
- 2.364.000 registros,
- 6.840 bloques DSP48E2.

Estas características la convierten en una plataforma FPGA de gran capacidad, adecuada para comparar núcleos de punto flotante en un entorno sin restricciones de recursos. No obstante, si se persiguiera una evaluación más realista en términos de integración final, sería necesario recurrir a un flujo de síntesis orientado a ASIC, fuera del alcance de este trabajo.

No se aplicaron limitaciones físicas de colocación ni de distribución lógica (como *pblocks* o áreas reservadas). De este modo, cada núcleo pudo ser ubicado libremente por la herramienta de síntesis.

En el caso de **VFloat**, fue necesario instanciar el IP *mul_man_multiplier* recomendado por sus autores para la multiplicación de mantisas. Los núcleos **HardFloat**, **FPNew** y **OpenCores-FPU** se sintetizaron directamente a partir de su RTL, sin recurrir a IPs del catálogo de Vivado.

3.2.2. Herramienta y flujo de síntesis

Se utilizó **Vivado 2020.2** como herramienta de síntesis e implementación para todos los núcleos evaluados. Esta versión permite trabajar directamente con código RTL o integrar IPs cuando es necesario, y ofrece soporte completo para el dispositivo `xcu200-fsgd2104-2-e` de la tarjeta Alveo U200.

Los diseños se sintetizaron en modo `out_of_context`, lo que permite evaluar cada módulo de forma aislada, sin necesidad de integrarlo en un sistema completo. Este enfoque facilita la comparación entre núcleos bajo condiciones homogéneas.

La síntesis se realizó a través de la interfaz gráfica de Vivado, aunque en el caso de algunos núcleos (como FPnew) se emplearon scripts TCL automatizados proporcionados por los propios repositorios. En todos los casos se utilizó la siguiente línea base de síntesis:

```
synth_design -top <nombre_top> -part xcu200-fsgd2104-2-e
```

Tras la síntesis, se ejecutaron las fases completas de implementación:

- `opt_design`: optimización lógica inicial.
- `place_design`: ubicación física de los recursos.
- `phys_opt_design`: optimización física posterior a la colocación.
- `route_design`: enrutado final de señales.

Este flujo fue común a todos los núcleos, garantizando coherencia metodológica en las métricas obtenidas posteriormente.

3.2.3. Métricas comparadas

Para comparar los núcleos evaluados, se han utilizado cinco métricas. Todas se han obtenido bajo condiciones controladas y homogéneas, utilizando implementaciones en formato FP32 salvo que se indique lo contrario.

- **Área**: se refiere a los recursos consumidos en la FPGA tras la implementación. Se mide en número de LUTs, flip-flops y, en su caso, bloques DSP48 utilizados. Los valores se extrajeron del reporte de implementación de Vivado para cada módulo.
- **Frecuencia máxima**: frecuencia de reloj más alta alcanzable por el núcleo tras la implementación completa. Se define como el inverso del mínimo periodo de reloj que no presenta violaciones de temporización en las comprobaciones de *setup* y *hold*. El valor se obtuvo reduciendo iterativamente el periodo de reloj hasta alcanzar el límite aceptado por la herramienta sin errores de temporización.
- **Latencia**: número de ciclos de reloj necesarios para completar una operación desde que los datos entran hasta que se obtiene el resultado final. En módulos segmentados, la latencia corresponde exactamente al número de etapas del pipeline. En módulos secuenciales representa el número total de ciclos consumidos por la máquina de estados para completar una operación, sin poder

aceptar nuevos datos hasta finalizar la actual. En aquellos casos en los que la latencia no pudo determinarse mediante simulación directa, se recurrió a la documentación del proyecto.

- **Operaciones y formato**: las métricas se han recogido para varios formatos de coma flotante, incluyendo FP16, FP32 y FP64, según las capacidades de cada núcleo. FP32 se ha usado como formato base de comparación, ya que está implementado en todos los diseños. Las operaciones evaluadas incluyen suma y resta, multiplicación, división y raíz cuadrada. En cada tabla del capítulo se indica explícitamente el formato y la operación a la que corresponde cada resultado.

- **Procedencia de los datos**: en cada tabla se indica si los resultados provienen de síntesis propias hechas sobre Vivado 2020.2 y Alveo U200 o de documentación pública. En los casos en que no fue posible completar la implementación, se incluyeron datos publicados para completar la comparativa.

3.3. Panorama de FPUs open-source

El ecosistema open-source ofrece hoy una variedad creciente de unidades de punto flotante orientadas a distintos entornos: desde bibliotecas académicas para FPGA hasta bloques altamente configurables pensados para integración ASIC o uso en procesadores RISC-V. Estas FPUs difieren en complejidad, arquitectura, cobertura del estándar IEEE 754, soporte de formatos numéricos y grado de adopción. Este trabajo se ha centrado en aquellas implementaciones con licencia abierta, documentación disponible y posibilidad de síntesis en FPGA mediante Vivado.

A continuación se presentan algunos de los proyectos disponibles:

- **HardFloat** (Hauser, 2019), una de las implementaciones más adoptadas en el ecosistema RISC-V, con arquitectura recodificada, soporte para FP16, FP32 y FP64, y módulos secuenciales para operaciones complejas como división o raíz cuadrada.

- **FPnew** (Mach et al., 2021), arquitectura configurable desarrollada en el marco del proyecto PULP, orientada a operaciones de transprecisión. Permite definir formatos personalizados de punto flotante, abarcando desde representaciones compactas como FP8 hasta formatos estándar cómo FP16, FP32 y FP64. Ha sido integrada en procesadores del OpenHW Group, como CV32E40P (antes RI5CY) y CVA6 (antes Ariane) y está diseñada para facilitar si integración en otros núcleos RISC-V.

- **VFloat** (Wang y Leeser, 2010), biblioteca desarrollada por la Northeastern University en VHDL, orientada a diseños de precisión variable. Permite parametrizar el número de bits de exponente y mantisa por operación, cubriendo tanto los formatos estándar como configuraciones personalizadas. Segmenta la

lógica en bloques funcionales independientes para suma, multiplicación, división, raíz cuadrada, acumulación y conversión entre fijo y flotante, con una arquitectura completamente segmentada en etapas de pipeline. No cumple completamente el estándar ya que no soporta subnormales ni todos los modos de redondeo.

- **Open Floating Point Unit** (Ussleman, 2000), núcleo de precisión simple publicado en *OpenCores.org* y diseñado en Verilog. Implementa ADD, SUB, MUL y DIV, con soporte para cuatro modos de redondeo del estándar IEEE 2008, todos menos *Round Ties to Away*. Acepta una operación por ciclo y entrega el resultado a los 4 ciclos. Genera todas las señales de excepción, pero no propaga `SNaN` ni soporta subnormales, por lo que su conformidad con IEEE 754 es parcial.

- **FPU100** (Al-Eryani, 2000), FPU de precisión simple implementada en VHDL, con cumplimiento completo del estándar IEEE 754-1985. Soporta operaciones de suma, resta, multiplicación, división y raíz cuadrada, incluyendo los cuatro modos de redondeo del estándar y gestión de excepciones. Cada operación se implementa como un bloque independiente desactivable, estructurado en tres etapas: pre-normalización, núcleo aritmético y post-normalización. Emplea segmentación en pipeline para alcanzar frecuencias de hasta 100 MHz (en FPGA Altera Cyclone I), con multiplicación paralela y unidades de división y raíz cuadrada implementadas de forma iterativa. El diseño fue desarrollado como proyecto individual y publicado en *OpenCores.org*

- **FPU-Dawson** (Dawson, 2013), conjunto de módulos en Verilog publicados en GitHub como proyecto independiente, con soporte para precisión simple y doble conforme a IEEE 754-1985. Implementan ADD, MUL y DIV, así como conversiones entre enteros y entre formatos flotantes (float/double). Gestiona subnormales, NaN, $\pm\infty$ y ± 0, con el modo *round-to-nearest*, aunque sin flags de excepción IEEE. Cada operación se implementa como un módulo autónomo mediante una máquina de estados secuencial, sin segmentación, procesando los operandos de forma iterativa. El diseño está orientado a minimizar el área a costa de la velocidad.

- **FPU Single and Double Precision** (Öksüz, 2019), proyecto independiente en GitHub que implementa una FPU en Verilog y VHDL con soporte para precisión simple y doble según IEEE 754-2008. Ofrece operaciones de suma, resta, multiplicación, división, raíz cuadrada, FMA, comparación, máximo/mínimo, sign injection y conversiones entre enteros y formatos flotantes, todas con cinco modos de redondeo y propagación de excepciones mediante flags IEEE. Emplea una arquitectura unificada que escala ambos formatos a una pseudo-precisión extendida para simplificar el diseño y reducir recursos, evitando la extensión del pipeline para subnormales, que son tratados como normalizados. Esta aproximación permite un menor coste en área, pero la hace incompatible con arquitecturas SIMD. Todas las operaciones salvo división y raíz cuadrada están segmentadas en tres ciclos, estas últimas se procesan de forma iterativa con dos modos de latencia (rápido/lento).

- **FloPoCo** Dinechin y Pasca (2011), generador de hardware que permite crear operadores aritméticos optimizados para FPGA a partir de descripciones parametrizadas, pero que no constituye una FPU integrada como tal. Cada operador se implementa como una clase en C++ que genera directamente código VHDL, permitiendo ajustar tanto la precisión de los datos como la frecuencia objetivo mediante segmentación automática adaptada al modelo concreto de FPGA. Los módulos generados no garantizan cumplimiento total del estándar IEEE 754, ya que no incluyen por defecto soporte para subnormales, NaN o flags de excepción.

Este recorrido evidencia la gran diversidad y madurez alcanzadas por el ecosistema open-source de FPUs, que abarca desde núcleos ampliamente adoptados y documentados, hasta herramientas generadoras y proyectos individuales de menor alcance.

Sin embargo, no todas las implementaciones presentadas cumplen los requisitos necesarios para una evaluación técnica detallada en este trabajo. Se han priorizado aquellas FPUs que:

- ofrecen soporte, al menos, para operaciones básicas en precisión simple,
- cuentan con licencia abierta y documentación suficiente,
- presentan un grado de madurez comprobado por su integración en proyectos relevantes o su uso académico,
- y son compatibles con la síntesis en FPGA a través de Vivado.

Por ello, proyectos como FloPoCo, aunque relevantes como generadores de operadores, o núcleos extremadamente básicos, quedan fuera del análisis comparativo principal.

El estudio se centrará, por tanto, en un subconjunto de FPUs representativas que cumplen estos criterios: HardFloat, FPnew, VFloat.

En los apartados siguientes se analiza en detalle su arquitectura, grado de cumplimiento del estándar IEEE 754, y resultados de síntesis sobre FPGA, a fin de identificar la alternativa más adecuada para los objetivos del proyecto.

3.4. Descripción de las candidatas

3.4.1. HardFloat

HardFloat es una biblioteca de módulos en Verilog desarrollada por John R. Hauser como parte de su trabajo en la Universidad de California, Berkeley. Su primera versión pública se publicó en 2019 y está distribuida bajo una licencia BSD modificada, que permite uso académico sin restricciones, pero requiere autorización

expresa del autor para la integración en proyectos comerciales. Se puede usar con arquitecturas como x86-SSE, ARM-VFPv2 y RISC-V, lo que permite adaptar el sistema a las convenciones de cada entorno de ejecución.

Este diseño se caracteriza por su modularidad y alta parametrización. Cada operación se encapsula en un módulo independiente, configurable mediante los parámetros **expWidth** y **sigWidth**, que definen respectivamente el ancho del campo de exponente y la precisión de la mantisa. Esta flexibilidad permite soportar tanto los formatos binarios estándar de 2008, como otros formatos personalizados, siempre que cumplan la relación correspondiente que se indica a continuación:

$$p \leq 2^{(w-2)} + 3$$

donde w representa el valor de expWidth y p el de sigWidth. Esta condición impone un mínimo de 3 bits para cada uno, y asegura que la codificación interna pueda manejar adecuadamente los casos especiales definidos por la norma. Sin embargo, debe tenerse en cuenta que el soporte garantizado por la biblioteca se limita a los formatos estándar. Para cualquier configuración distinta, es necesario validar el comportamiento mediante pruebas adicionales.

Hardfloat transforma los valores en formato estándar a una representación interna recodificada (`recFN`), que añade un bit extra al exponente y normaliza los subnormales. En este contexto, "normalizar" significa transformar un subnormal en un número que se comporta internamente como si fuese normal, ajustando su exponente y desplazando la mantisa hacia la izquierda, al igual que en los normales, el bit implícito sea 1. Esto no altera el valor representado, pero permite tratar todos los operandos con una lógica uniforme dentro de los módulos, lo que reduce la complejidad de las unidades funcionales. De forma complementaria, puede utilizar una representación descompuesta (`rawFN`) que separa el signo, el exponente tratado como un entero con signo, y la mantisa completa. También incluye indicadores como **isZero**, **isInf** o **isNaN**, útiles para clasificar valores especiales.

La biblioteca incluye un conjunto de módulos independientes que implementan operaciones estándar del punto flotante:

- `addRecFN`: módulo combinacional que realiza suma y resta sobre operandos recodificados.
- `mulRecFN`: módulo combinacional para calcular el producto de dos operandos.
- `mulAddRecFN`: módulo combinacional que implementa la operación FMA.
- `divSqrtRecFN_small`: lleva a cabo división y raíz cuadrada mediante un algoritmo secuencial de una iteración por bit con entrada/salida por handshake. La latencia puede estimarse como latencia $\approx$ `sigWidth` $+\delta$, donde `sigWidth` es el número de bits significativos y δ es una constante pequeña, que representa los ciclos adicionales necesarios para el arranque del algoritmo y el manejo de condiciones especiales.

- `recFNToRecFN`: conversión entre formatos recodificados con diferente precisión.
- `fNToRecFN`: convierte valores en formato IEEE estándar a formato recodificado interno.
- `recFNToFN`: convierte valores recodificados al formato IEEE estándar.
- `compareRecFN`: compara dos operandos recodificados conforme al estándar.
- `isNaNRecFN`, `isSignalingNaNRecFN`: detectan y clasifican valores NaN.

Todos los módulos operativos admiten los cinco modos de redondeo definidos por la norma: round to nearest even, toward zero, toward $+\infty$, toward $-\infty$, round to nearest, ties to max magnitude. Además de un sexto modo adicional, round to odd, útil en entornos donde se requiere conservar información de error en los bits menos significativos, técnica conocida como jamming. Se seleccionan según la macro asignada a la señal de control `roundingMode`.

Estos módulos operan sobre las representaciones internas (`recFN` o `rawFN`). Por ello, es necesario usar los bloques para convertir los formatos, como `fNToRecFN` y `recFNToFN`, antes y después de cada módulo funcional, con el fin de transformar los operandos desde y a formato IEEE estándar. Esta separación entre la lógica funcional y la lógica de interfaz aporta flexibilidad y reutilización, aunque requiere encapsular correctamente los componentes al integrarlos en arquitecturas con interfaces externas.

El comportamiento de los módulos puede ajustarse mediante la señal de entrada `control`. El ancho y valor por defecto de esta señal se definen en el archivo `HardFloat_consts.v`. El objetivo por defecto es permitir seleccionar si la condición de *tininess*, necesaria para detectar underflow, se comprueba antes o después del redondeo, de acuerdo con lo estipulado por el estándar. Esta señal puede ampliarse para incluir ajustes adicionales, como la forma de gestionar conversiones inválidas o desbordamientos, la activación de la propagación de los *payloads* de `NaN` cuando están presentes en los operandos o la definición del `NaN` por defecto a utilizar.

Las excepciones definidas por el estándar se detectan y transmiten mediante cinco señales de salida: `invalid`, `divideByZero`, `overflow`, `underflow`, `inexact`.

Además, incluye una infraestructura de pruebas basada en las bibliotecas TestFloat y SoftFloat, con soporte para simuladores como Verilator e Icarus Verilog. Esto permite verificar con alta precisión la conformidad con la norma, comparando los resultados con referencias conocidas.

En definitiva, esta biblioteca pone a disposición del diseñador un repertorio preciso y bien estructurado de módulos funcionales, con los que construir una unidad de punto flotante conforme al estándar, adaptable en formato y comportamiento, y sin restricciones de uso.

3.4.2. FPnew

El proyecto FPnew introduce la TP-FPU (Transprecision Floating-Point Unit), una unidad de punto flotante altamente configurable diseñada para operaciones de transprecision (Malossi et al., 2018). Esta técnica permite ajustar dinámicamente la precisión de los cálculos, optimizando tanto el rendimiento como el consumo energético. Ha sido desarrollada por el laboratorio Integrated Systems Laboratory (IIS) de ETH Zürich y mantenida por el OpenHW Group. Está escrita en SystemVerilog y diseñada como una solución paramétrica y escalable para procesadores RISC-V.

La unidad soporta una amplia gama de formatos de punto flotante, desde los estándares FP64, FP32, FP16 y FP8, hasta formatos personalizados de longitud variable. Además, implementa técnicas avanzadas de optimización, como pipelines configurables y clock-gating, que ayudan a reducir el consumo energético durante las operaciones. Estas características permiten que la FPU mantenga un alto nivel de eficiencia operativa, ajustándose dinámicamente a las necesidades de la aplicación.

La arquitectura interna está organizada en bloques modulares que permiten ejecutar eficientemente una variedad de operaciones avanzadas. Estos bloques pueden configurarse para operar de manera paralela o fusionada, según los requisitos de rendimiento y eficiencia del sistema. También es importante destacar el uso de técnicas avanzadas de control de flujo y gestión de pipeline, lo que permite un procesamiento eficiente de datos y una reducción de la latencia.

El núcleo, llamado `fpnew_top`, puede recibir hasta tres operandos de punto flotante por ciclo de reloj, acompañados de señales de control que determinan el tipo de operación y los formatos involucrados. Los operandos se direccionan hacia uno de los cuatro bloques de operaciones, que son los encargados de ejecutar las instrucciones específicas. Además, incorpora un sistema de **pipelining** totalmente configurable para cada bloque, sin depender de una codificación rígida a ninguna profundidad del pipe específica.

Estos bloques incluyen:

- **ADDMUL**: Maneja las operaciones de suma, resta, multiplicación y FMA. Este bloque utiliza unidades FMA completamente segmentadas, implementadas con un único flujo de datos, que ofrecen una precisión de hasta $1/2$ ulp. En configuraciones con merged slices, se emplean versiones multi-formato de estas unidades FMA, capaces de realizar la multiplicación en un formato y la suma en otro, permitiendo operar con operandos de distintas precisiones.

- **DIVSQRT**: Este bloque se encarga de las operaciones de división y raíz cuadrada, implementadas mediante un divisor iterativo non-restoring que calcula tres bits de la mantisa por ciclo de reloj. El número de iteraciones puede ajustarse para intercambiar precisión por rendimiento en entornos de transprecision, permitiendo finalizar el cálculo antes de alcanzar la máxima precisión, lo que aumenta el error y *no garantiza el redondeo correcto exigido por IEEE 754*. En configuración estándar, la latencia se aproxima por $3 + \lceil p/3 \rceil$, siendo p los bits de mantisa del formato.

- **DIVSQRT**: Este bloque se encarga de las operaciones de división y raíz cuadrada, implementadas mediante un divisor iterativo non-restoring. El núcleo iterativo calcula tres bits de la mantisa por ciclo de reloj. El número de iteraciones puede ajustarse para intercambiar precisión por rendimiento en entornos de transprecision, permitiendo finalizar el cálculo antes de alcanzar la máxima precisión, lo que aumenta el error y no garantiza el redondeo correcto exigido por IEEE 754.

- **COMP**: Este bloque incluye operaciones de comparación, mínimo y máximo, inyección de signo y clasificación. Las comparaciones (`<`, `<=`, `==`) y operaciones como `minimumNumber` o `copySign` se seleccionan en función del modo de redondeo activo (`rnd_mode_i`), y se implementan mediante lógica ligera con operaciones a nivel de bit. También permite determinar si un número es normal, subnormal, cero, infinito o NaN, conforme a IEEE 754.

- **CONV**: Este bloque gestiona las conversiones entre distintos formatos de punto flotante, así como entre enteros y flotantes, tanto con signo como sin signo. Incluye también operaciones de empaquetado como `CPKAB` y `CPKCD`, que permiten combinar múltiples resultados en un único registro de salida. En configuraciones MERGED, el tercer operando (op[2]) puede usarse como registro destino para empaquetado SIMD

Se admiten los cinco modos de redondeo del estándar y además, incluye el modo `roundToOdd`, útil en conversiones con truncado y propagación controlada del error (jamming). En conformidad con el estándar, la unidad genera cinco flags de excepción definidos en `fpnew_pkg`: NV (invalid operation), DZ (division by zero), OF (overflow), UF (underflow) y NX (inexact), que se codifican en la salida `status_o`.

Además, se incorpora un sistema de etiquetas (operation tags). Estas etiquetas son metadatos que acompañan a cada operación a través de los bloques funcionales y permiten rastrear los resultados, vinculándolos con las operaciones específicas que los produjeron. Estas permanecen sincronizadas con las operaciones y pueden configurarse según las necesidades de la aplicación, o desactivarse si no son necesarias.

Dentro de estos bloques funcionales se emplean secciones del procesador (slices) para gestionar el procesamiento de diferentes formatos de punto flotante. Los slices permiten procesar datos en paralelo o compartir hardware entre múltiples formatos, optimizando el rendimiento y el uso del área del chip. También soporta operaciones SIMD, lo que permite ejecutar varias operaciones en paralelo dentro de un único ciclo de reloj.

- **Parallel Slices**: Estos slices están dedicados a un solo formato de punto flotante. Cada slice contiene su propio pipeline interno, y las slices inactivas pueden apagarse mediante técnicas de clock-gating, reduciendo así el consumo energético.

- **Merged Slices**: Estos slices permiten que múltiples formatos compartan el mismo hardware, lo que es más eficiente en términos de uso del área del chip.

Pueden introducir una latencia adicional debido al manejo compartido, especialmente en operaciones SIMD.

La implementación incluye el mecanismo de NaN-boxing requerido por la arquitectura RISC-V. Este consiste en exigir que los operandos de formatos de precisión reducida (como FP16 o FP8), codificados dentro de registros más anchos, tengan los bits más significativos establecidos a uno. La unidad permite activar o desactivar la verificación de esta condición mediante el parámetro `EnableNanBox`, y siempre garantiza que los resultados producidos en estos formatos estén correctamente codificados.

El control de flujo se gestiona mediante un protocolo de **handshaking** síncrono basado en señales `valid` y `ready` para coordinar la transferencia de datos entre los bloques funcionales. Este protocolo asegura que los datos se transmitan solo cuando tanto el emisor como el receptor están listos, evitando cuellos de botella. Cada etapa del pipeline envía una señal `valid` cuando los datos están listos para ser transferidos, y la etapa siguiente responde con una señal `ready` cuando está preparada para recibir esos datos. La transferencia solo ocurre cuando ambas señales están activas, lo que garantiza que los datos no se pierdan y que se mantenga una sincronización efectiva entre las diferentes partes de la FPU.

Además, este protocolo de handshake permite la implementación del gating de reloj tanto a nivel de grano fino como grueso. A nivel de grano fino, las etapas individuales del pipeline que no están en uso pueden desactivarse dinámicamente, lo que minimiza el consumo de energía. A nivel de grano grueso, bloques completos de la FPU pueden apagarse si no están siendo utilizados, reduciendo así la actividad innecesaria y el consumo de energía global.

Para gestionar el manejo de las operaciones con diferentes tiempos de procesamiento y permitir que puedan ejecutarse simultáneamente, FPnew utiliza un mecanismo de arbitraje justo de tipo round-robin. Este sistema distribuye el acceso a la salida de la FPU entre los bloques que han completado sus operaciones, evitando que una sola operación monopolice los recursos o que se generen cuellos de botella. Este arbitraje es especialmente crucial para manejar operaciones con latencias variables, permitiendo que las operaciones más rápidas no se retrasen por las más lentas, sin necesidad de predeterminar estas latencias.

Por último, cabe destacar que esta FPU es altamente configurable, y su comportamiento se define mediante dos estructuras principales: **fpu_features_t** y **fpu_implementation_t**. La primera define las características operativas externas de la unidad: qué formatos puede manejar, cómo se representan los datos, y si se habilita el procesamiento vectorial o la compatibilidad con codificaciones IEEE extendidas. La segunda estructura controla la arquitectura interna y el comportamiento temporal de cada bloque funcional. A continuación, se describen sus campos más relevantes:

- `fpu_features_t`:
  - `Width`: ancho total del datapath, que define el número de bits con los

que se transmiten los operandos y resultados. Debe ser al menos igual al formato más amplio activado.

- `FpFmtMask`: máscara que activa los formatos de punto flotante disponibles.
- `IntFmtMask`: permite activar conversiones entre enteros con y sin signo, según el grupo de instrucciones `CONV`.
- `EnableVectors`: habilita la ejecución SIMD sobre formatos estrechos cuando el ancho lo permite.
- `EnableNanBox`: activa la verificación del NaN-boxing, usada en formatos reducidos como FP8 y FP16 dentro de registros más anchos.

- `fpu_implementation_t`:

  - `PipeRegs`: número de registros de pipeline por operación y formato. Esta segmentación reduce la lógica combinacional en cada etapa y permite mejorar la frecuencia de operación.
  - `PipeConfig`: define la ubicación física de esos registros (antes, después o distribuidos en el operador).
  - `UnitTypes`: permite elegir entre una implementación con una unidad funcional dedicada por formato (`PARALLEL`), que mejora el rendimiento a costa de mayor área, o una unidad compartida (`MERGED`), que reduce el consumo de recursos pero introduce multiplexado interno y posibles cuellos de latencia.
  - `DivSqrtSel`: selecciona entre diferentes versiones del bloque `DIVSQRT`, que difieren en el conjunto de formatos soportados, el uso de operaciones vectoriales, la latencia y el consumo de recursos. Este parámetro permite adaptar la unidad a distintos perfiles de complejidad o rendimiento según las necesidades del sistema.

Gracias a esta parametrización explícita, FPnew puede configurarse como una unidad puramente combinacional o como una arquitectura segmentada de alto rendimiento, adaptándose a distintas aplicaciones y entornos de ejecución.

3.4.3. VFloat

VFloat es una biblioteca escrita en VHDL que proporciona un conjunto de operadores de punto flotante con precisión variable, diseñada para su uso en arquitecturas FPGA. Fue desarrollada por Xiaojun Wang y Miriam Leeser (2010) en la Northeastern University con el objetivo de ofrecer una alternativa abierta, portable y controlable frente a los bloques IP propietarios de los principales fabricantes. VFloat adopta un enfoque modular y explícito, en el que cada operador se instancia y configura manualmente, proporcionando al diseñador control sobre la precisión, la estructura interna y el flujo de datos.

La biblioteca está dividida en dos bloques principales: por un lado, el conjunto de `General Modules`, que incluye unidades de normalización, redondeo y alineación, además de bloques de corrección y paquetes de constantes; y por otro, los `Variable Precision Modules`, que implementan las operaciones aritméticas principales: suma/resta, multiplicación, división, raíz cuadrada, recíproco, acumulación, multiplicador-acumulador y conversión entre representaciones numéricas. Cada módulo está diseñado como una unidad autónoma, que puede integrarse de forma aislada o formar parte de una arquitectura más amplia.

Una de las características clave es la precisión parametrizable, que permite ajustar el número de bits del exponente y de la fracción mediante parámetros de tipo `generic` en cada módulo. Esta configuración no se limita a los formatos predefinidos por IEEE 754, sino que permite definir formatos personalizados adaptados a los requisitos del sistema. El archivo `float_pkg.vhd` centraliza los tipos, constantes y funciones reutilizables, y actúa como punto de entrada para la configuración global de cada instancia.

Los módulos funcionales más representativos de la biblioteca son los siguientes:

- `variable_precision_adder`: implementa la suma y resta de operandos flotantes mediante cuatro etapas: reordenamiento (`swap`), alineación de mantisas (`shift_adjust`), operación aritmética (`add_sub`) y corrección (`correction`). La alineación introduce un bit de guarda, y la corrección ajusta el exponente si hay acarreo, insertando el bit sobrante y truncando el menos significativo. Esta etapa también ensambla el resultado y emite una señal de excepción. Si los signos coinciden se realiza una suma; en caso contrario, una resta, reutilizando este mismo módulo tras invertir el signo del segundo operando. La operación requiere una latencia fija de cuatro ciclos de reloj.

- `variable_precision_multiplier`: implementa la multiplicación en tres ciclos de reloj. En el primero, en paralelo, se realiza el producto de las mantisas, la suma de exponentes y el cálculo del signo; en el segundo se corrige el sesgo acumulado debido al formato sesgado de entrada; y en el tercero se comprueba si se ha producido una excepción. Si se detecta, la salida se fuerza a cero y se activa la señal `EXCEPTION_OUT`. La latencia total depende del retardo del multiplicador IP empleado, y se calcula como `MUL_MANTISSA_DELAY + 2`.

- `variable_precision_reciprocal`: este módulo calcula la inversa de un operando flotante mediante una método optimizado basado en tablas de búsqueda y expansión polinómica de Taylor (Ercegovac et al., 2000). El cálculo se divide en tres etapas: una reducción inicial que obtiene una aproximación del recíproco mediante una LUT, una evaluación polinómica que refina la estimación usando multiplicadores dedicados, y una etapa final de postprocesado que ajusta el resultado. Además de su uso independiente, este módulo actúa como bloque fundamental para implementar la división y la raíz cuadrada.

- `variable_precision_divider`: la operación de división se apoya en el módulo de recíproco descrito anteriormente. A las tres fases ya explicadas, se añade

una multiplicación final del recíproco estimado por el dividendo, por lo que la latencia total corresponde a cinco multiplicadores más dos ciclos adicionales.

- `variable_precision_square_root`: la raíz cuadrada sigue la misma estructura que el recíproco, adaptando la función objetivo y las tablas de búsqueda para aproximar $\sqrt{Y}$, e incorporando una corrección adicional si el exponente es impar. Comparte los mismos multiplicadores y esquema de latencias que el recíproco.

- `variable_precision_accumulator`: implementa un sumador en coma flotante con arquitectura pipeline, capaz de realizar acumulaciones continuas gracias a un sistema de buffers y multiplexores que permite aceptar nuevos operandos cada ciclo. Emplea un normalizador optimizado que divide la mantisa en dos partes para acelerar la detección de ceros significativos. La latencia típica incluye 1 ciclo para el multiplexor de entrada, 4 para la suma, 2 para la normalización y 1 de buffering. Esta biblioteca no implementa un multiply-accumulate (MAC) de forma nativa, pero puede lograrse añadiendo un multiplicador previo.

- La biblioteca también incluye módulos de conversión entre punto flotante y punto fijo, tanto con signo como sin signo. Estas unidades permiten ajustar el punto binario y el tamaño de entrada y salida, gestionando el redondeo, la saturación y el truncado.

La latencia de cada módulo depende del número y tipo de multiplicadores empleados en su estructura, así como del retardo de cada uno de ellos. Estos valores deben indicarse manualmente mediante constantes en el archivo `float_pkg.vhd`, en función del comportamiento real del recurso IP utilizado. El diseñador debe conocer, por la documentación del fabricante, cuántos ciclos requiere cada operador, y reflejar esta información en los parámetros de configuración antes de la integración final del sistema.

Estas IP se instancian para crear las tablas de búsqueda y los productos parciales en las operaciones de recíproco, multiplicación, división y raíz cuadrada. En consecuencia, el desempeño real depende de los recursos proporcionados y su configuración concreta. Vivado permite su construcción mediante LUT o DSP y optimización basada en área o en velocidad. Es importante destacar respecto al uso de las tablas de búsqueda implementadas mediante **ROMs** que VFloat solo aporta ficheros de inicialización hasta FP64, por lo que cualquier precisión superior no es soportada.

VFloat emplea la codificación binaria del formato estándar. No obstante, no implementa el estándar en su totalidad. No ofrece soporte para subnormales y solo admite dos modos de redondeo: `roundTiesToEven` y `roundTowardZero`. La selección se realiza mediante una señal binaria de control (`ROUND`). Aunque no se garantiza redondeo exacto en todos los casos, los resultados mantienen una precisión cercana a 1 ulp en las configuraciones habituales.

Cada componente dispone de señales `EXCEPTION_IN` y `EXCEPTION_OUT` que permiten que las condiciones de error, tanto recibidas como generadas internamente

(por ejemplo, división por cero, raíz cuadrada de número negativo o desbordamiento), se propaguen junto con los resultados a través del pipeline aritmético. Si una excepción es detectada o recibida, el módulo marca su salida como errónea y sustituye el valor de resultado por ceros. Sin embargo, estas señales no codifican el tipo de excepción según los cinco casos definidos por IEEE 754, un `EXCEPTION_OUT` = 1 indica que el resultado no es válido según la lógica del módulo y, por defecto, la salida numérica se fuerza a cero, lo que obliga al diseñador a gestionar manualmente estas situaciones si se requiere una conformidad normativa completa.

El control del flujo de datos se realiza mediante señales de control. Ciertos módulos como la división, la raíz cuadrada y el recíproco utilizan `READY` y `DONE` para indicar disponibilidad de datos y finalización de la operación. La interfaz incluye la señal `STALL`, utilizada para pausar el avance del pipeline hasta que vuelva a estar en bajo. Estas señales están disponibles únicamente en aquellos operadores que requieren control secuencial explícito, mientras que los módulos combinacionales como la suma o la multiplicación funcionan de forma continua sin necesidad de señalización de estado. No existe un protocolo handshake unificado del tipo `valid/ready`.

En conjunto, VFloat proporciona una colección de operadores modulares de punto flotante con precisión variable, ajustables en latencia y configuración interna, y orientados a arquitecturas FPGA donde se prioriza el control explícito sobre la ejecución y el uso de recursos. Su integración requiere que el diseñador configure adecuadamente las constantes de latencia y gestione la propagación de errores cuando sea necesario, lo que permite un diseño optimizado, pero exige un mayor grado de intervención manual.

3.4.4. Taner Öksüz FPU

Esta unidad de punto flotante (Öksüz, 2019) implementa un conjunto de operaciones aritméticas y de conversión compatibles con el estándar IEEE 754-2008 para formatos de precisión simple y doble. El repertorio abarca suma, resta, multiplicación, implementadas mediante FMA, división, raíz cuadrada, comparaciones, funciones `min`/`max`, operaciones de signo y conversiones entre formatos enteros y de punto flotante.

El diseño emplea una única ruta de datos compartida para ambos formatos, escalada a una representación extendida denominada *pseudo extended precision* (12 bits de exponente y 52 bits de mantisa), lo que permite normalizar todos los datos de entrada y evita el tratamiento explícito de subnormales, simplificando el pipeline pero limitando el soporte SIMD o vectorial.

El control del flujo se realiza a través de señales de tipo `ready`/`done` para la división y la raíz cuadrada. Estas comparten un submódulo común, pero cada una sigue un camino algorítmico distinto, seleccionable mediante un parámetro genérico denominado `performance`. Este parámetro permite conmutar entre un algoritmo iterativo (rápido) o uno en punto fijo (más lento).

La implementación emplea la codificación canónica para los valores `NaN`, com-

patible con TestFloat, empleando los patrones `0x7FC00000` para simple precisión y `0x7FF8000000000000` para doble. Las operaciones elementales tienen latencia fija de tres ciclos, mientras que las iterativas varían según el modo de precisión y rendimiento, como se detalla:

- **Precisión simple**: división 14/29 ciclos, raíz cuadrada 17/28 ciclos (rápido/-lento).
- **Precisión doble**: división 14/58 ciclos, raíz cuadrada 17/57 ciclos.

Todas las operaciones soportan modos de redondeo estándar (`RNE`, `RTZ`, `RDN`, `RUP`, `RMM`) y se acompañan de flags de excepción que siguen la codificación del estándar IEEE: `NX` (inexacto), `UF` (underflow), `OF` (overflow), `DZ` (divide-by-zero) y `NV` (operación inválida).

- `fp_exe`: módulo de control que recibe los operandos, el código de operación (`opcode`), el formato (`fmt`) y el modo de redondeo (`rm`), y los distribuye a los módulos funcionales correspondientes. Recoge el resultado y los flags de excepción del bloque activo, y aplica un redondeo final cuando la operación lo requiere. Controla el flujo y el pipeline mediante señales de disponibilidad y borrado (`ready`, `clear`), actuando como intermediario central entre la interfaz externa y los bloques internos.
- `fp_fma`: módulo pipelineado de tres etapas que implementa de forma unificada `fmadd`, `fmsub`, `fnmadd`, `fnmsub`, `fadd`, `fsub` y `fmul` mediante adaptación dinámica de operandos y signos. Realiza la multiplicación, alineación y suma/resta de mantisas en precisión extendida, normaliza el resultado mediante conteo de ceros significativos con el bloque `lzc_256` y extrae los bits GRS (guard, round, sticky) para el redondeo IEEE 754. Detecta las excepciones sNaN, qNaN, Inf y cero y propaga los flags junto con la señal `ready`.
- `fp_fdiv`: unidad iterativa controlada por FSM que gestiona división y raíz cuadrada. El parámetro `performance` selecciona el algoritmo: `1` para iteraciones funcionales rápidas con Newton–Raphson, `0` para iteraciones en punto fijo más lentas basado en lógica secuencial. Sincroniza el flujo con señales `start/ready/done` y propaga excepciones IEEE.
- `fp_mac`: módulo sencillo de multiplicador-acumulador que realiza la operación $(b \times c) \pm a$ en precisión extendida, sumando o restando el producto a un sumando alineado según el bit de control. Carece de lógica de normalización, detección de excepciones o redondeo IEEE.
- `fp_rnd`: bloque combinacional que aplica el modo de redondeo IEEE seleccionado (RNE, RTZ, RDN, RUP, RMM) sobre el resultado normalizado, a partir de los bits GRS.
- `fp_cmp`, `fp_max`, `fp_sgnj`, `fp_cvt`: módulos dedicados a comparación, cálculo de mínimo/máximo, operaciones de signo y conversiones entre formatos enteros y flotantes. Todos presentan latencia fija y arquitectura pipeline.

- `lzc_*`: módulos de conteo de ceros líderes en la mantisa extendida, utilizados para normalización tras operaciones aritméticas. Implementan versiones parametrizadas para distintos anchos de entrada.

El diseño es portable y optimizado para FPGA, con soporte de scripts automatizados para simulación y validación. Ofrece una solución conforme al estándar para operaciones en punto flotante de precisión simple y doble, con equilibrio entre complejidad, portabilidad y rendimiento configurable.

3.5. Evaluación y selección de la unidad funcional

A continuación se presentan los resultados obtenidos para los formatos FP16, FP32 y FP64 tras la síntesis de los principales módulos de las tres FPUs analizadas. Las métricas utilizadas en la comparación son las descritas en la Sección 3.2.

Las configuraciones empleadas fueron:

- **HardFloat**: Esta FPU no ofrece opciones de configuración; se implementaron directamente los módulos individuales (`addRecFN`, `mulRecFN`, `divSqrtRecFN`) y `fNToRecFN`/`recFNToFN` para diferentes precisiones.
- **FPnew**: configuración por defecto con `Implementation` = PARALLEL para ADDMUL y MERGED para DIVSQRT, que implementa hardware compartido entre varios formatos, `PipeConfig` = AFTER, `PipeRegs` = 0 y paralelismo desactivado. Esta elección aproxima el comportamiento de bloques separados y permite compararlo con el resto de candidatas.
- **VFloat**: La única posible configuración viene a la hora de implementar los multiplicadores mediante los IP Core, eligiendo el pipeline de cada uno. Vivado permite implementarlos con LUTs o con DSP, optimizando por área o frecuencia. Para la comparación general se ha optado por un pipeline de 1 y la implementación mediante DSPs para facilitar su similitud con el resto de FPUs.

Los resultados incluyen número de LUTs, FF, bloques DSP, frecuencia máxima alcanzable y latencia en ciclos de reloj. Más adelante se presentan resultados obtenidos con otras configuraciones. A continuación se muestran las tablas correspondientes:

Formato	**Operación**	**HardFloat**					**FPnew**					**VFloat**				
		LUTs	FFs	DSP	MHz	Ciclos	LUTs	FFs	DSP	MHz	Ciclos	LUTs	FFs	DSP	MHz	Ciclos
FP16	ADD/SUB	285	0	0	73.8	—	565	0	1	75.5	—	172	158	0	607	4
	MUL	162	0	0	89.2	—	565	0	1	75.5	—	287	160	0	602	3
	DIV/SQRT	250	57	0	280.7	—	1907	221	0	226.2	—	639 / 623	699 / 256	0	360 / 326	18 / 16
FP32	ADD/SUB	510	0	0	62.4	—	1049	0	2	35.9	—	380	283	0	581	4
	MUL	270	0	0	77.1	—	1049	0	2	35.9	—	902	275	0	455	3
	DIV/SQRT	448	100	0	283.9	—	2133	254	0	207.7	—	1877 / 1842	516 / 735	0	263 / 234	24 / 17
FP64	ADD/SUB	1106	0	0	43.8	—	2377	0	9	40.5	—	380	283	0	546	4
	MUL	655	0	0	48.9	—	2377	0	9	40.5	—	3630	530	0	372	3
	DIV/SQRT	829	191	0	281.6	—	2350	301	0	193.2	—	7245 / 6529	1147 / 1196	0	208 / 176	28 / 23
FP128	ADD/SUB	2282	0	0	43.0	—	5563	0	49	25.1	—	1962	1044	0	352	4
	MUL	1620	0	0	39.8	—	5563	0	49	25.1	—	—	—	—	—	—
	DIV/SQRT	1748	376	0	384.9	—	2320	302	0	209.3	—	— / —	— / —	—	— / —	— / —

Tabla 3.1: Resultados comparativos por formato.

HarFloat

En HardFloat todas las operaciones se implementan de forma combinacional, sin registros de pipeline internos, lo que explica la baja ocupación de FFs y la ausencia de DSPs en la tabla. La latencia en ciclos es fija e igual a uno, mientras que la frecuencia máxima está limitada por el camino crítico de cada operación.

FPnew

Como ya se ha explicado en la Sección 3.4.2, uno de los aspectos distintivos de FPnew es su alto grado de parametrización. A diferencia de HardFloat o VFloat, permite explorar un espacio de diseño con múltiples combinaciones que afectan al área, la latencia y la frecuencia máxima alcanzable.

Tal y como se aprecia en la tabla del Apéndice A.1, la configuración `MERGED`, en la que se comparte hardware entre formatos, presenta un mayor consumo de recursos que `PARALLEL`, que implementa cada formato con su propia lógica. Aunque resulte contraintuitivo, este sobrecoste se debe a que la lógica para manejar el multiformato sigue presente, lo que penaliza a la opción `MERGED` en este escenario.

A continuación se muestran los resultados de síntesis en el caso multiformato, que ilustran esta diferencia de manera clara.

Implementation	PipeRegs	Fmax [MHz]	WNS [ns]	WPWS [ns]	LUTs	FFs	DSP
MERGED	0	203.4	0.000	2.183	12568	305	49
	1	207.5	0.000	2.135	12892	964	49
PARALLEL	0	203.4	0.000	2.183	15905	305	49
	1	205.5	0.000	2.158	15979	1097	61

Tabla 3.2: Resultados de síntesis de FPnew multiformato.

Como se observa en los resultados, la configuración `MERGED` resulta significativamente más eficiente en términos de área cuando se activan múltiples formatos simultáneamente, ya que reutiliza gran parte de la lógica común entre ellos. Sin embargo, `PARALLEL` ofrece una mayor granularidad de control al permitir ajustar individualmente la latencia por formato mediante la inserción de registros de pipeline. La elección entre una u otra configuración dependerá del contexto de uso.

VFloat

En la tabla comparativa principal se observa un uso elevado de LUTs porque se ha optado por implementaciones sin DSPs con 1 registro de pipeline. Sin embargo, VFloat permite variar el recurso base y la profundidad de pipeline de los IP Cores, lo que cambia significativamente el equilibrio entre área, frecuencia y latencia.

Para FP128 únicamente se han obtenido resultados del bloque ADD/SUB. En el resto de operaciones VFloat requiere un multiplicador de $(man_bits + 1) \times$

($man_bits + 1$) y tablas ROM de gran tamaño. Con Vivado no es posible generar ese tamaño de multiplicador y la biblioteca tampoco proporciona los archivos de inicialización para las ROMs a partir de FP64.

A continuación se muestran ejemplos representativos que ilustran configuraciones más eficientes en términos de recursos o frecuencia, según la operación y el formato.

Formato	Operación	MHz	Lat.	LUTs	FFs	DSP
FP16	MUL	507 / 602	4 / 3	286 / 151	316 / 138	0 / 1
	DIV	397 / 425	18	639 / 241	699 / 297	0 / 5
	SQRT	607 / 563	16	647 / 201	678 / 318	0 / 4
FP32	MUL	479 / 463	4	910 / 287	940 / 267	0 / 2
	DIV	312 / 432	24 / 23	1888 / 485	2400 / 764	0 / 9
	SQRT	522 / 506	17–18	1842 / 442	2195 / 716	0 / 9
FP64	MUL	364 / 375	6 / 12	3673 / 905	3660 / 1003	0 / 10
	DIV	362 / 318	28 / 34	7498 / 1241	7610 / 1460	0 / 22
	SQRT	388 / 459	23–27	6616 / 1464	6600 / 1488	0 / 17

Tabla 3.3: Resultados de VFloat con configuraciones LUT/DSP y PipeRegs recomendados por Vivado.

En las operaciones de MUL, las implementaciones con DSP reducen claramente el número de LUTs respecto a las realizadas con lógica, manteniendo frecuencias muy próximas. En DIV y SQRT, el uso de LUTs provoca un crecimiento significativo de recursos a medida que aumenta la precisión, mientras que con DSPs el área se reduce aunque la latencia se incrementa, especialmente en FP64. Estos resultados reflejan cómo la elección del recurso base condiciona directamente el equilibrio entre área, latencia y frecuencia en VFloat.

Quiero destacar que tanto la documentación como el código de los módulos contienen bastantes errores y son poco intuitivos. Las ROMs para las tablas (div/recip/sqrt) deben generarse como IP de memoria e inicializarse con ficheros `.coe/.mif` del repositorio y solo hay hasta FP64. Las constantes de retardo en `float_pkg.vhd` deben ajustarse a la latencia efectiva de los IP (multiplicadores/ROM) que selecciones.

Para terminar, quiero dar unas conclusiones finales para cada operación:

- ADD/SUB: HardFloat implementa suma/resta en un ciclo combinacional con bajo consumo de recursos. FPnew integra la suma dentro del bloque `ADDMUL`, compartiendo recursos con la multiplicación. VFloat dispone de un módulo ADD/SUB con latencia de 4 ciclos.
- MUL: HardFloat ofrece multiplicación combinacional de un ciclo. FPnew comparte la multiplicación en el bloque `ADDMUL`. VFloat depende de un multiplicador externo, con mayor latencia y consumo.
- DIV/SQRT: HardFloat utiliza un algoritmo iterativo secuencial con elevada latencia pero área contenida. FPnew emplea la variante `PulpDivsqrt`, con

latencias dependientes del formato de entre 7 - 41 ciclos totales. VFloat tiene las latencias según el multiplicador IP Core implementado bastante elevadas en precisiones superiores y un coste de área bastante mayor que el resto.

En resumen, **HardFloat** destaca por su eficiencia modular, latencia mínima en ADD/MUL y compatibilidad total con IEEE 754, lo que la convierte en una opción robusta y fácilmente integrable en arquitecturas modulares. **FPnew** proporciona flexibilidad arquitectónica y buen rendimiento en operaciones complejas como la división y la raíz cuadrada, gracias a la posibilidad de parametrizar la latencia mediante `PipeRegs`; sin embargo, al tratarse de una FPU completa y monolítica excede el alcance de este trabajo, donde se buscan bloques independientes. **VFloat**, pese a su precisión configurable y a que la latencia puede ajustarse mediante el pipeline de los IP Cores, depende en gran medida de estos y presenta una fuerte variabilidad en sus métricas, lo que dificulta obtener resultados consistentes y limita su idoneidad como alternativa práctica.

A la vista de estas consideraciones, se ha seleccionado **HardFloat** como base para el diseño de la unidad funcional de este trabajo.

Capítulo 4

Diseño y Evaluación de la FPU

El capítulo describe el proceso de diseño, implementación y evaluación de las unidades de punto flotante desarrolladas. Con el fin de garantizar la reproducibilidad, el código fuente completo del wrapper, controladores, bancos de pruebas y scripts se encuentra disponible en un repositorio: `https://github.com/helio-0/fpu-tfg`.

El repositorio se encuentra organizado de manera jerárquica para facilitar su uso:

- `src`: código fuente en SystemVerilog y VHDL de los wrappers y controladores diseñados en este trabajo.
- `tb`: bancos de pruebas empleados para FP16, FP32 y FP64.
- `third_party`: bibliotecas externas integradas en el diseño, diferenciadas en subcarpetas para **HardFloat** y **VFloat**.
- `scripts`: scripts de automatización para simulación y reportes.

El detalle sobre las licencias de las bibliotecas externas y del propio material desarrollado se presenta en el Apéndice A.2, de manera que se garantice el cumplimiento de las condiciones de distribución establecidas por cada autor.

4.1. Metodología y herramientas

La implementación se ha realizado con herramientas estándar del ecosistema AMD Xilinx:

Entorno de diseño
: Vivado 2020.2 en su interfaz gráfica. Cada paso de síntesis e implementación se lanza desde “Run Synthesis” y “Run Implementation”, lo que resulta más cómodo en esta fase de aprendizaje, ya que permite inspeccionar la jerarquía y los esquemáticos en tiempo real. Para no depender únicamente de la GUI,

tras cada pasada Vivado genera un journal con los comandos TCL utilizados; a partir de ese fichero se ha construido un script que permite repetir la generación de los reportes de manera más automatizada con cada iteración.

Base de código
El núcleo HardFloat se descargó como archivo `.zip` desde la página oficial del autor.[1] Se ha utilizado tal cual, sin modificar los archivos originales.

Plataforma destino
Se trabajó sobre la tarjeta Alveo U200, perteneciente a la familia de aceleradores de Xilinx, en concreto el modelo FPGA UltraScale+ `xcu200-fsgd2104-2-e`.

Aunque existen tarjetas más pequeñas, se eligió la Alveo U200 por tres motivos principales:

- **Aprender con una plataforma representativa.** La U200 pertenece a la gama Alveo orientada a entornos de datacenter, y es habitual en proyectos de aceleración de bases de datos, inteligencia artificial o vídeo. Trabajar con ella permite experimentar con un flujo de diseño equivalente al que se emplea en producción y familiarizarse con una de las herramientas que usa la industria.
- **Facilitar futuras pruebas en servidor.** El diseño que funcione aquí puede cargarse sin cambios en clouds públicos (como AWS F1 o Azure NP), ya que todas estas plataformas comparten una infraestructura de ejecución similar. Esto permite escalar el proyecto si fuera necesario, sin tener que adaptarlo a otro entorno.
- **Acceso a documentación y soporte.** AMD Xilinx ofrece guías paso a paso, ejemplos y foros específicos para la U200. Al estar incluida en la licencia gratuita de Vivado, es una plataforma ampliamente utilizada y cuenta con abundante material de referencia y soporte comunitario.

4.2. Diseño de la FPU

La `FPU_Wrapper` encapsula tres unidades HardFloat (ADD/SUB, MUL, DIV/SQRT) más los conversores `fNToRecFN`/`recFNToFN`.

Los módulos desarrollados en este trabajo son:

- *FPU_ Wrapper*: interfaz *ready/valid* con la CPU, conversión IEEE↔recFN, distribución de operandos a la unidad seleccionada y registro del resultado final junto con el tag y las flags de excepción, de forma que queden alineados temporalmente en la salida.

[1] Dirección consultada: `https://www.jhauser.us/arithmetic/HardFloat-1/doc/HardFloat-Verilog.html`.

- *FPU_Controller*: FSM de seis estados que arbitra las unidades y gestiona el protocolo con *divSqrtRecFN_small*. Al aceptar una operación, registra el operador, el modo de redondeo y la etiqueta para mantenerlos estables durante toda la ejecución. Genera las señales de control y un pulso de escritura para capturar el resultado en el registro de salida.

- *Demux*: distribuye los operandos recodificados a las tres rutas de datos (ADD/SUB, MUL, DIV/SQRT) con anchura REC_WIDTH.

- *Mux3to1*: selecciona el resultado final y las banderas correspondientes en función de la operación.

Este apartado describe la arquitectura interna de la unidad funcional de punto flotante desarrollada, así como los módulos que la componen y la lógica de control que gobierna su funcionamiento. El diseño se basa en integrar varios bloques del núcleo HardFloat con una estructura de control propia que permita seleccionar la operación, gestionar el flujo de datos y entregar el resultado de forma sincronizada.

El objetivo principal ha sido construir una FPU funcional, capaz de ejecutar operaciones básicas en coma flotante según el estándar IEEE 754, con una estructura modular, comprensible y fácil de extender en futuras versiones. Para ello, se ha seguido un enfoque simple pero escalable, donde cada componente tiene una función bien definida y se comunica a través de una interfaz clara.

La `FPU_Wrapper` integra tres unidades de HardFloat (suma/resta, multiplicación y división/raíz cuadrada) y los conversores `fNToRecFN`/`recFNToFN`. El diseño es parametrizable mediante `EXP_WIDTH` y `MAN_WIDTH`: internamente, todo el camino de datos opera en formato *recFN* y la interfaz externa utiliza formato IEEE. Una máquina de estados coordina la aceptación de operaciones, la ejecución y la entrega alineada de resultado, flags y etiqueta.

Este apartado describe la arquitectura interna de la unidad funcional de punto flotante desarrollada, así como los módulos que la componen y la lógica de control que gobierna su funcionamiento. El diseño se basa en integrar varios bloques del núcleo HardFloat con una estructura de control propia que permita seleccionar la operación, gestionar el flujo de datos y entregar el resultado de forma sincronizada.

4.2.1. Visión general

La unidad funcional se organiza en torno a un módulo superior (`FPU_Wrapper`) que encapsula la lógica de control, los operadores reutilizados de HardFloat y los módulos auxiliares de conversión y selección de datos. De este modo, la interfaz externa trabaja con operandos y resultados en formato IEEE 754, mientras que el cálculo interno se realiza sobre la representación recodificada *recFN* de HardFloat.

El diseño es parametrizable mediante los genéricos `EXP_WIDTH` y `MAN_WIDTH`, de los que se derivan las anchuras de los buses internos. En particular, el camino de datos opera con un ancho recodificado `REC_WIDTH` = `EXP_WIDTH` + `MAN_WIDTH` + 1,

que conecta directamente los conversores con los tres operadores principales y con la lógica de multiplexado.

La arquitectura distingue entre dos tipos de bloques:

- **Operadores reutilizados**: que ejecutan las operaciones aritméticas sobre el formato *recFN*.
- **Módulos propios**, encargados de distribuir los operandos, seleccionar el resultado correcto, coordinar la ejecución mediante una máquina de estados y registrar de forma sincronizada el valor final, las flags de excepción y el tag.

En cada operación, los operandos IEEE se convierten primero a *recFN*, se encaminan mediante el demultiplexor hacia la unidad correspondiente y el resultado es gestionado por el `FPU_Controller`, que coordina los protocolos de *handshake* y la sincronización. El valor seleccionado por el multiplexor se registra junto con las flags y el tag, y finalmente un conversor lo devuelve al formato IEEE para su entrega a la salida.

La Figura 4.9 muestra el esquema general de la FPU, donde se diferencian los bloques reutilizados del núcleo HardFloat y los módulos desarrollados específicamente en este proyecto.

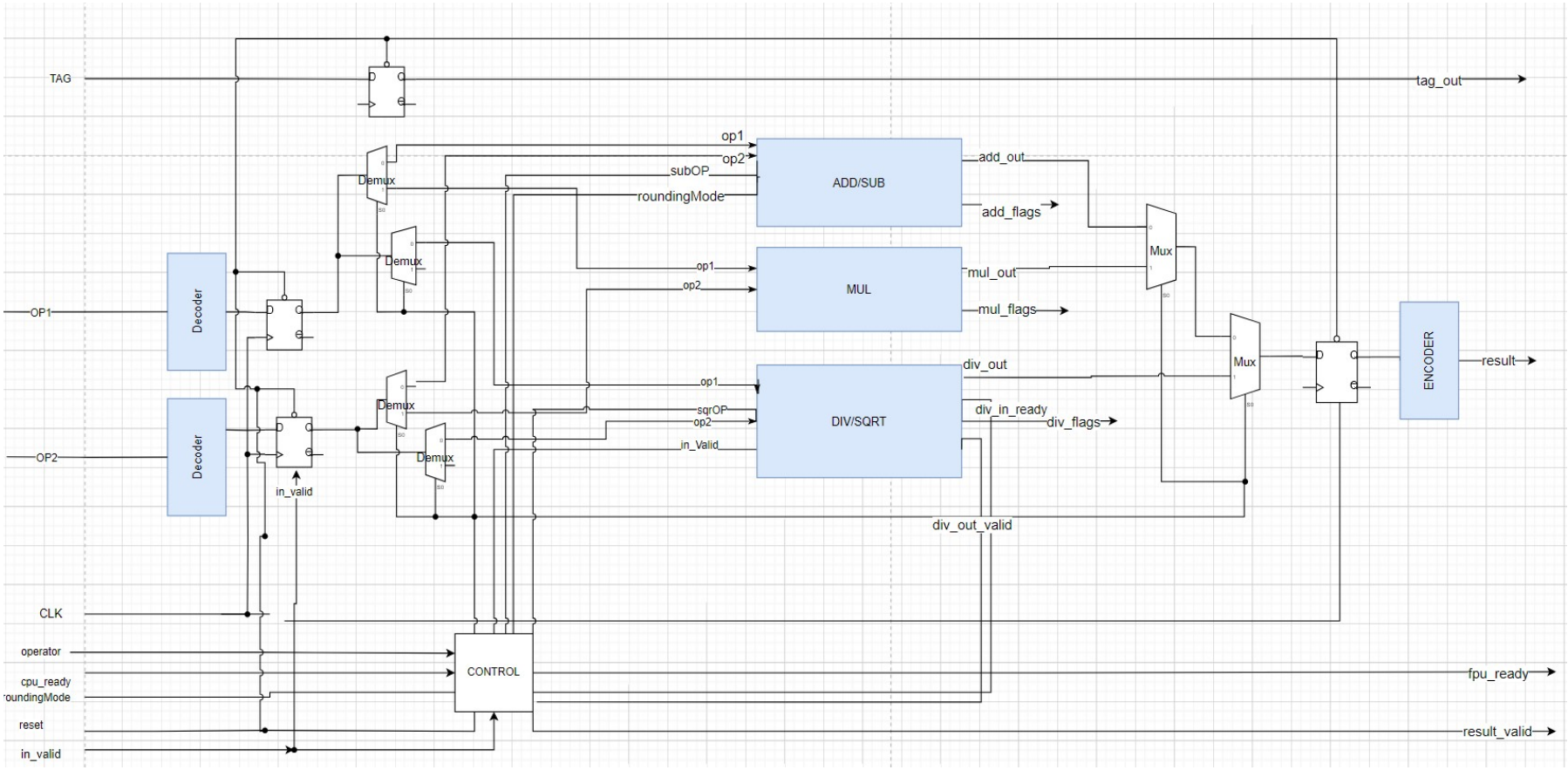

Figura 4.1: Diagrama de bloques de la FPU de módulos Hardfloat.

4.2.2. Componentes reutilizados

El diseño incorpora tres operadores de HardFloat que se instancian sin modificaciones y actúan como núcleo aritmético de la FPU:

- `addRecFN`: módulo combinacional de suma y resta en formato *recFN*. Implementa modos de redondeo y flags de excepción, y en este diseño su salida se registra para alinear el resultado con el resto del camino de datos.
- `mulRecFN`: operador combinacional de multiplicación en *recFN*, también registrado a la salida. Ofrece soporte completo para `NaN`, ceros e infinitos, y genera las flags IEEE correspondientes.
- `divSqrtRecFN_small`: unidad secuencial para división y raíz cuadrada. Emplea un algoritmo iterativo de varios ciclos y sigue un protocolo de *handshake* gestionado por la FSM. La operación concreta se selecciona mediante una señal de control; en el caso de raíz cuadrada el segundo operando no se utiliza. Se ha configurado la entrada `nReset` como activa en bajo.

Junto a ellos se utilizan los conversores fNToRecFN y recFNToFN, responsables de transformar los datos entre el formato IEEE 754 externo y la representación recodificada interna.

4.2.3. Módulos desarrollados

Además de los bloques de cálculo, el diseño incluye varios módulos creados específicamente para adaptar HardFloat a un entorno más completo y controlado. Estos módulos se encargan de la gestión de entradas y salidas, la selección de la operación y la coordinación de los bloques operativos.

- **FPU_Wrapper**: Módulo superior que conecta con la CPU y encapsula el camino de datos interno en formato recodificado. Expone los puertos para la entrada de los valores, la selección del operador, modo de redondeo, tag, `in_valid` y `cpu_ready`; y las salidas `result`, `exceptionFlags[4:0]`, `tag_out`, `result_valid` y `fpu_ready`. Internamente instancia los conversores fNToRecFN / recFNToFN, el `Demux` y el `Mux3to1`, además de los registros de salida para resultado, flags y tag. La lógica de selección (`demux_control`) y de captura (`reg_we`) proviene del controlador.
- **FPU_Controller**: Máquina de estados que arbitra las unidades operativas y coordina el protocolo con divSqrtRecFN_small. Al aceptar una operación, mantiene constantes el operador, el modo de redondeo y el tag guardándolos en registros. Genera `subOp`, `sqrOp`, `op_select_o`, el pulso `div_in_valid` para iniciar DIV/SQRT y `reg_we` para capturar el resultado en los registros de salida. Lanza `fpu_ready` cuando la FPU puede aceptar una nueva operación y `result_valid` cuando el resultado registrado está listo.

- **Demux**: Reparte los operandos hacia las unidades manteniendo el ancho interno `REC_WIDTH`, según `op_select`. La arquitectura utiliza dos conversores fNToRecFN en paralelo para preparar `inOp1` e `inOp2`.
- **Mux3to1**: Selecciona el resultado en *recFN* procedente de una de las tres unidades según `op_select`. Su salida alimenta el registro de resultado del FPU_Wrapper. El ancho es parametrizable y coincide con el del `Demux`.
- **we_register**: Registro con write-enable y reset síncrono activo en alto. Se emplea en tres instancias para almacenar el resultado en *recFN*, las banderas de excepción y el tag. Al activar `reg_we`, los tres valores se actualizan en el mismo ciclo, garantizando que `result`, `exceptionFlags` y `tag_out` queden alineados temporalmente tras la conversión final a IEEE.

A continuación se muestran todas las señales del módulo y su utilidad.

Señal	Dir.	Descripción
		Entradas de la CPU
`clk`	in	Reloj principal del sistema.
`reset`	in	Reset síncrono activo en alto.
`in_valid`	in	Indica que la CPU presenta operación y operandos válidos.
`cpu_ready`	in	Indica que la CPU está lista para emitir una operación.
`operator[2:0]`	in	Código de operación.
`rounding_mode[2:0]`	in	Modo de redondeo IEEE 754.
`tag[3:0]`	in	Identificador asociado a la operación.
`inOp1`	in	Operando A en formato IEEE.
`inOp2`	in	Operando B en formato IEEE.
		Salidas hacia la CPU
`fpu_ready`	out	La FPU puede aceptar una nueva operación.
`result_valid`	out	Indica que el resultado registrado es válido en este ciclo.
`result`	out	Resultado en formato IEEE.
`exceptionFlags[4:0]`	out	Banderas de excepción IEEE 754.
`tag_out[3:0]`	out	Identificador asociado al resultado.
		Señales internas de control
`op_select_o[1:0]`	out	Selección de la unidad operativa.
`subOp`	out	Diferencia ADD/SUB en la unidad de suma/resta.
`sqrOp`	out	Selecciona operación raíz cuadrada en la unidad DIV/SQRT.
`div_in_valid`	out	Pulso de inicio de operación en divSqrtRecFN_small.
`div_in_ready`	in	Indica que la unidad DIV/SQRT puede aceptar nuevos operandos.
`div_out_valid`	in	Resultado de la unidad DIV/SQRT listo.
`reg_we`	out	Habilita la escritura simultánea de registros de salida.
`rounding_mode_o[2:0]`	out	Modo de redondeo registrado.
`tag_o[3:0]`	out	Etiqueta registrada internamente, alineada al resultado.

Tabla 4.1: Interfaz externa e internas principales del módulo `FPU_Wrapper`.

4.2.4. Flujo de datos y control

El diseño de la unidad funcional sigue una arquitectura claramente separada entre ruta de datos y lógica de control. Esta división facilita la comprensión del

sistema y permite que cada parte se mantenga modular y escalable.

Ruta de datos

La ruta de datos comienza con la recepción de dos operandos (`inOp1`/`inOp2`) de `IEEE_W` = `EXP_WIDTH` + `MAN_WIDTH` bits y un código de operación. Ambos operandos se convierten al formato interno mediante dos instancias de fNToRecFN en paralelo (`outDecoOP1`/`outDecoOP2`), lo que permite procesarlos en el mismo ciclo y evita introducir estados adicionales para el control a cambio de un ligero aumento de área.

A continuación, se enrutan con `Demux` hacia una de las tres unidades de HardFloat en función de la señal de selección `op_select` y `subOp`, `sqrOp` activas si se trata de una resta o una raíz.

Las salidas de las unidades, el valor resultante y flags asociadas, se agrupan mediante dos Mux3to1 para generar `outMux` y `ex_flags_sel`. Luego se capturan en los registros (`result_reg`, `flags_reg`, `tag_reg`) cuando `reg_we`=1, de modo que resultado, flags y etiqueta queden alineados temporalmente.

Finalmente, se convierte el contenido de `result_reg` de vuelta a IEEE para llevar `result` a la interfaz externa.

Controlador

La ruta de control se implementa en el FPU_Controller, una máquina de estados que coordina la aceptación de operaciones, la selección de la unidad y la publicación de resultados. Su función principal es mantener estables los parámetros operación, modo de redondeo y etiqueta y generar las señales de control hacia el camino de datos y la unidad secuencial de división/raíz cuadrada.

La FSM se podría considerar híbrida ya que no es totalmente Moore, para no añadir estados cuando se calcula DIV/SQRT para guardar el valor. Recorre seis estados:

- S_INIT: estado inicial tras el `reset`, transición inmediata a `S_IDLE`.
- S_IDLE: la FPU está lista para aceptar una operación. Si las señales `in_valid` y `cpu_ready` están activos, se registran `operator`, `rounding_mode` y `tag`, y se pasa al estado correspondiente; para operaciones de división/raíz cuadrada también se requiere `div_in_ready`=1.
- S_START_CMB: para ADD, SUB o MUL, se genera un pulso `reg_we` que captura el resultado combinacional en un ciclo.
- S_START_DIV: para DIV o SQRT, se emite `div_in_valid` cuando la unidad indica `div_in_ready=1`.
- S_WAIT_DIV: el controlador permanece en espera hasta que la unidad activa `div_out_valid`; en ese momento se lanza `reg_we`.

- S_DONE: el resultado, las flags y la etiqueta registrados se envían a la salida activando `result_valid` durante un ciclo, tras el cual se retorna a `S_IDLE`.

Las señales de control principales son:

- op_select_o[1:0]: selecciona la unidad operativa.
- subOp y sqrOp: indican a las unidades si es resta o raíz cuadrada.
- reg_we: habilita la escritura simultánea en los registros de resultado, flags y etiqueta.
- div_in_valid, div_in_ready, div_out_valid: protocolo de handshake con la unidad divSqrtRecFN_small.
- rounding_mode_o y tag_o: valores registrados en la aceptación de la operación para garantizar coherencia en la salida.

En la Tabla 4.2 se detalla la codificación que emplea el controlador para activar la ruta adecuada en cada caso.

`operator`	Operación	`op_select_o`	Ruta
000	ADD	00	ADD/SUB
001	SUB	00	ADD/SUB
010	MUL	01	MUL
011	DIV	10	DIV/SQRT
100	SQRT	10	DIV/SQRT

Tabla 4.2: Codificación de operaciones y rutas de datos en la FPU.

A continuación se muestra el diagrama de estados de la FSM:

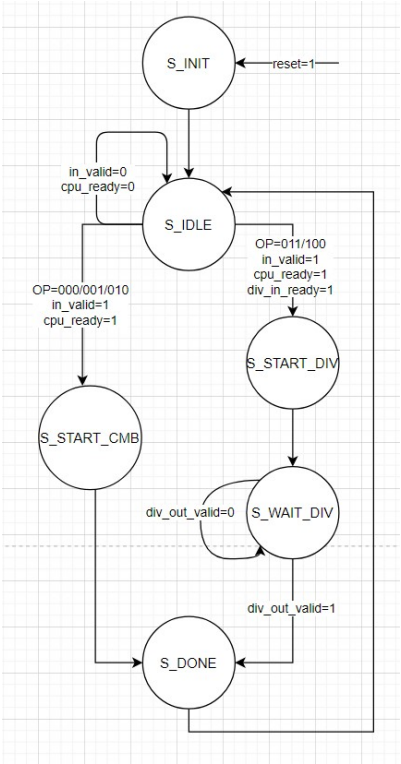

Figura 4.2: FSM del módulo de control.

4.2.5. Tratamiento de redondeo y excepciones

El diseño permite seleccionar dinámicamente el modo de redondeo a través de la señal de entrada `rounding_mode` de 3 bits. Esta señal se registra en el `FPU_Controller` al aceptar una operación y se mantiene constante durante toda su ejecución, propagándose después a las unidades aritméticas. Los modos soportados por HardFloat son los seis definidos por IEEE 754:

- `3'b000`: Round to Nearest, ties to Even
- `3'b001`: Round toward Zero
- `3'b010`: Round toward $-\infty$
- `3'b011`: Round toward $+\infty$
- `3'b100`: Round to Nearest, ties to Max Magnitude
- `3'b110`: round to odd

Cada operador produce además la salida `exceptionFlags[4:0]`, que codifica en orden {invalid, divide-by-zero, overflow, underflow, inexact}. Estas señales se multiplexan en función de la operación seleccionada y se registran junto con el resultado y la etiqueta, de modo que todos los valores se publican de forma alineada

en la salida. La configuración adoptada es *tininess after rounding*, por lo que el `underflow` únicamente se comprueba tras aplicar el redondeo.

Aunque en esta implementación las señales de excepción solo se propagan como salida, una posible mejora futura consistiría en integrar lógica interna que permita detectar condiciones específicas (como `invalid` o `overflow`) y generar respuestas a nivel de control, como bloquear la escritura del resultado, notificar a la CPU o activar mecanismos de depuración.

4.3. Simulación

Para validar el correcto funcionamiento de la FPU se realizaron simulaciones en Vivado, usando un testbench y monitorizando las principales señales de la interfaz junto con el estado de la FSM. A continuación se recogen ejemplos representativos de operaciones básicas y situaciones excepcionales.

Suma

La Figura 4.3 muestra la simulación de la operación 2.0 + 2.0. En el instante inicial, la CPU presenta los operandos en `inOp1` e `inOp2`, ambos con el valor `0x40000000`, correspondiente a 2.0 en formato IEEE 754, junto con el código de operación `000` (ADD) y la etiqueta `tag=0xA`.

Cuando las señales `in_valid` y `cpu_ready` están activas, el controlador acepta la operación y transita desde el estado `S_IDLE` a `S_START_CMB`. Al ser una operación combinacional, en el siguiente ciclo se habilita `reg_we` y el resultado se captura en los registros de salida.

El valor publicado en `result` es `0x40800000`, que corresponde a 4.0 en formato IEEE. La señal `result_valid` se activa durante un ciclo, indicando que tanto el resultado como `tag_out=0xA` y `exceptionFlags=00000` están listos para su lectura.

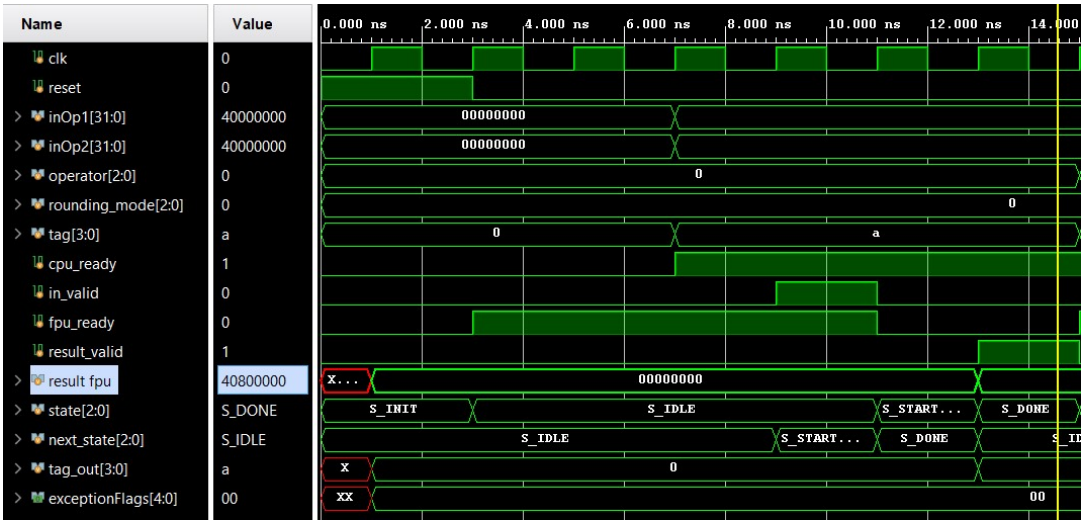

Figura 4.3: Simulación de la suma $2.0 + 2.0 = 4.0$ sin excepciones.

Multiplicación

La Figura 4.4 muestra la simulación de la operación 2.0×2.0. Los operandos `inOp1` e `inOp2` toman el valor `0x40000000` (2.0 en IEEE 754) y el código de operación es `010` (MUL).

El resultado obtenido en `result` es `0x40800000`, equivalente a 4.0, con `tag_out` coincidente y sin flags activas. Como en el caso de la suma, la operación es combinacional y el valor se publica en el ciclo siguiente a su aceptación.

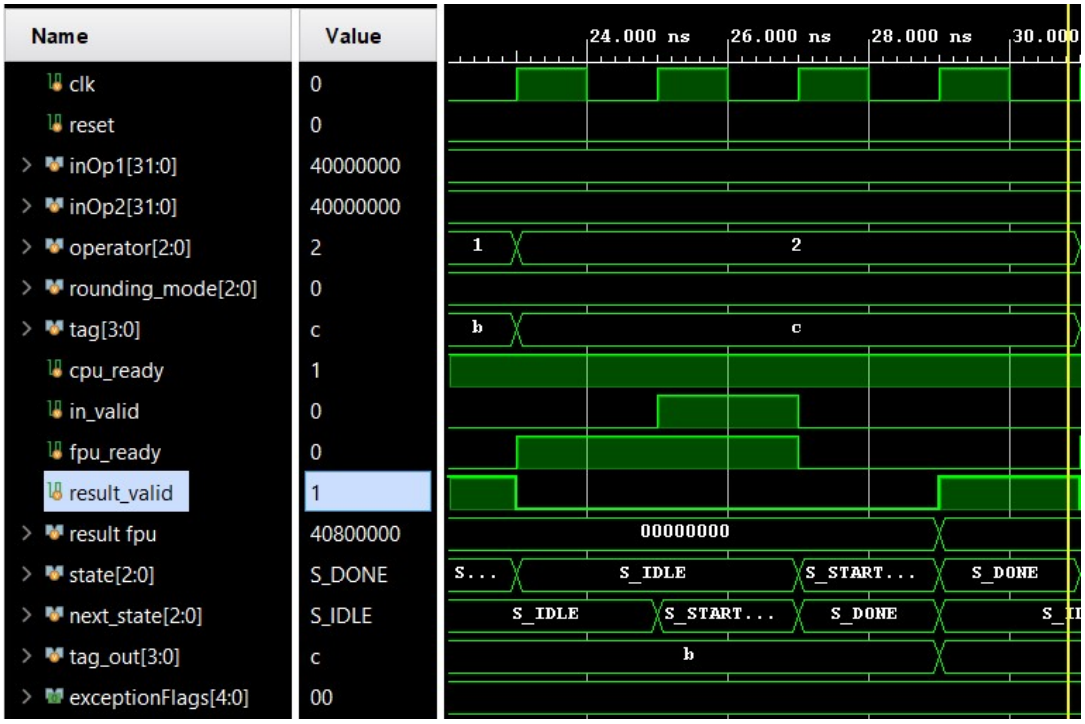

Figura 4.4: Simulación de la multiplicación $2.0 \times 2.0 = 4.0$ sin excepciones.

División

La Figura 4.5 muestra la operación 2.0/2.0. Los operandos `inOp1` y `inOp2` valen `0x40000000`, con `operator=011` (DIV).

A diferencia de ADD y MUL, esta unidad es secuencial y requiere varios ciclos más; tras activarse `in_valid`, el controlador emite `div_in_valid` y espera a que la unidad indique `div_out_valid`. En ese momento se captura el resultado. La salida es `0x3F800000`, correspondiente a 1.0.

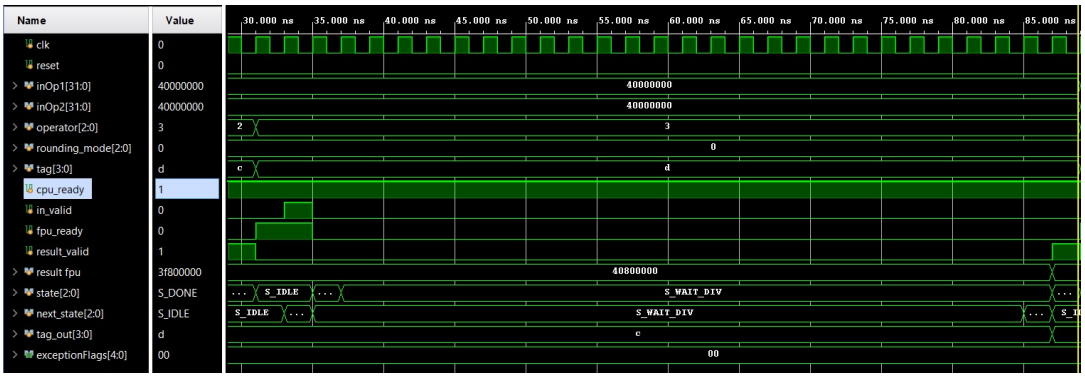

Figura 4.5: Simulación de la división 2.0/2.0 = 1.0 sin excepciones.

Raíz cuadrada

La Figura 4.6 muestra la operación $\sqrt{4.0}$. El operando `inOp1` se presenta con el valor `0x40800000`, mientras que `inOp2` no se utiliza. El código de operación es `100` (SQRT).

Como en la división, la unidad es secuencial: el controlador activa `div_in_valid` y espera a que se active `div_out_valid`. En ese ciclo se registra el resultado. El valor publicado en `result` es `0x40000000`, equivalente a 2.0.

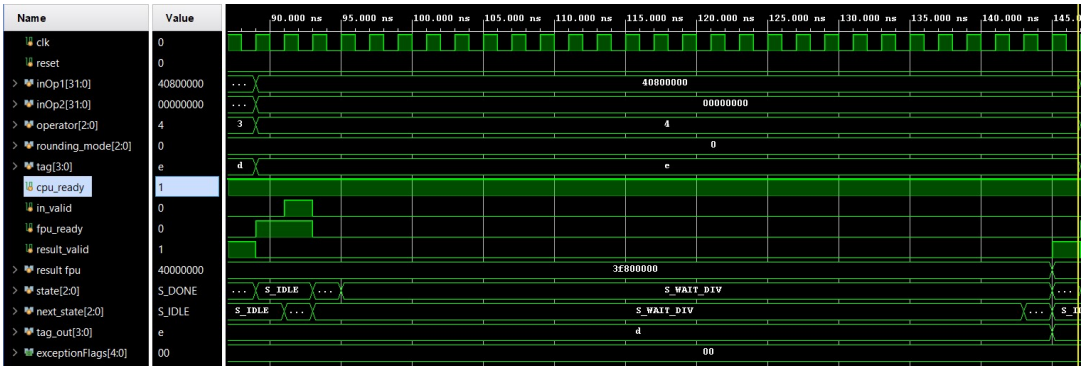

Figura 4.6: Simulación de la raíz cuadrada $\sqrt{4.0} = 2.0$ sin excepciones.

Excepción invalid

A continuación se muestra la operación 0.0/0.0. Ambos operandos se representan como `0x00000000` y el código de operación `011` (DIV).

La unidad produce como salida un `NaN` (`0x7FC00000`) y la señal `exceptionFlags` se activa con `10000`, correspondiente a la condición *invalid operation*.

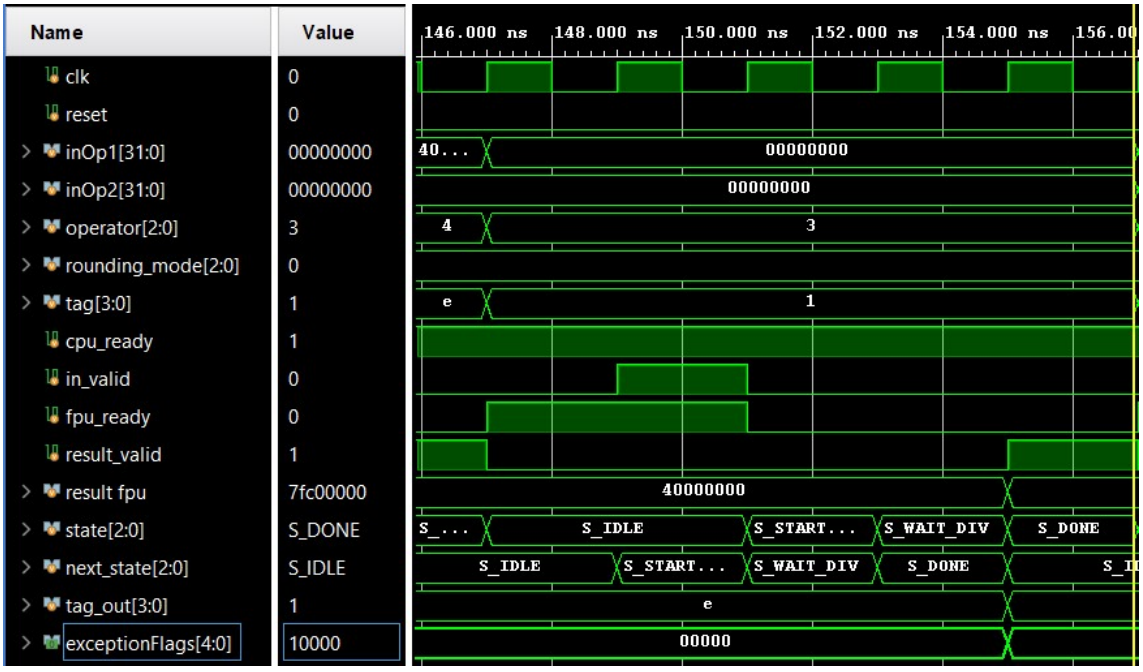

Figura 4.7: Excepción *invalid*: división 0.0/0.0 → NaN.

Excepción overflow

La Figura 4.8 corresponde a la operación max_finite × 2. El operando A es el mayor número finito representable en precisión simple (`0x7F7FFFFF`) y el operando B es 2.0 (`0x40000000`).

El resultado es $+\infty$ (`0x7F800000`), acompañado de `exceptionFlags=00101`, que codifica simultáneamente las condiciones de *overflow* e *inexact*.

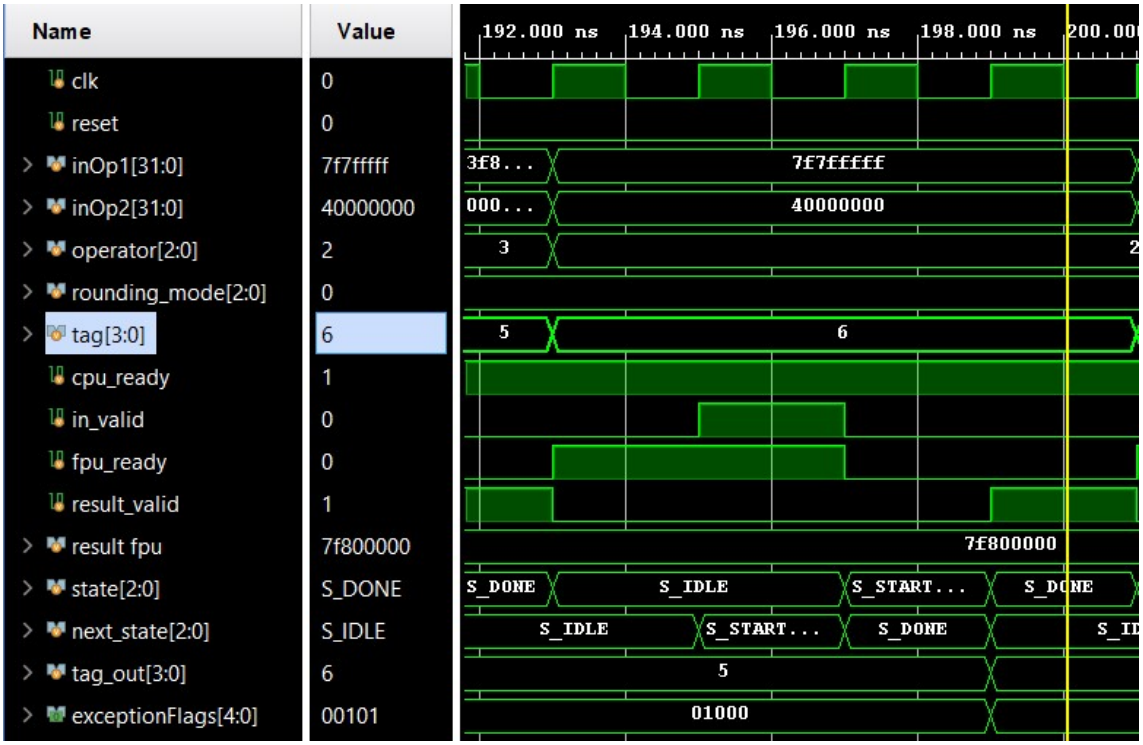

Figura 4.8: Excepción *overflow+inexact*: multiplicación de máximo número finito por 2.

4.4. Resultados de síntesis

La síntesis del diseño se realizó para los formatos FP16, FP32 y FP64 sobre la FPGA Xilinx Alveo U200 y a continuación, se resumen los recursos empleados y la frecuencia máxima alcanzada en cada configuración.

Formato	LUTs	FFs	C8	DSPs	Frecuencia (MHz)	Latencia (Ciclos)
FP16	836	99	16	1	100.0	1/1/15
FP32	1570	158	50	2	64.2	1/1/26
FP64	3688	281	105	9	63.8	1/1/56

Tabla 4.3: Resultados de síntesis de la FPU para distintas precisiones.

Para ADD, SUB y MUL el *throughput* efectivo es de una operación por ciclo a la frecuencia máxima alcanzada. En cambio, las operaciones DIV y SQRT presentan una latencia mayor, que aumenta con la precisión tanto por la naturaleza iterativa del algoritmo como por la frecuencia alcanzable en formatos más amplios. Estos resultados evidencian una escalabilidad predecible del diseño al variar `EXP_WIDTH`/`MAN_WIDTH`.

En los tres formatos, la ruta crítica identificada tras la implementación parte de registros del controlador y atraviesa la ruta aritmética activa (alineación, suma y

redondeo) hasta los registros de salida. El retardo está dominado por el *routing*, que representa entre un 70 % y un 77 % del total, sobre un camino combinacional muy largo, con una elevada cantidad de etapas lógicas en serie (≈ 37 en FP32). Con esta configuración se obtienen unos 64 MHz de frecuencia máxima. Una posible mejora consistiría en insertar una etapa de *pipeline* entre la suma y el redondeo para acortar el camino crítico y aumentar la frecuencia.

4.5. Diseño alternativo con módulos VFloat

Con el objetivo de contrastar el diseño principal basado en HardFloat, se desarrolló una FPU equivalente empleando los módulos de la biblioteca **VFloat**. Este núcleo, escrito en VHDL, usa un protocolo de *handshake* en todos sus módulos.

Esta FPU implementa las mismas operaciones que en el caso de HardFloat: ADD/SUB, MUL, DIV y SQRT. Para cada operación se instanció el bloque correspondiente de VFloat (`fp_add`, `fp_mul`, `fp_div`, `fp_sqrt`), conectando las señales de entrada y salida mediante una interfaz unificada. A diferencia de HardFloat, que requiere conversiones a formato recodificado (`recFN`), los módulos de VFloat aceptan directamente el empaquetado IEEE.

La suma y la resta se implementan reutilizando el mismo bloque `fp_adder`. Para diferenciar ambas operaciones, el controlador genera una señal auxiliar que indica si la operación solicitada es una resta. En ese caso, se modifica el operando B invirtiendo su bit de signo, de manera que la resta $A - B$ se convierte en la suma $A + (-B)$. Este mecanismo evita instanciar un bloque específico para SUB y simplifica la lógica.

Las operaciones de multiplicación, división y raíz cuadrada se resuelven mediante los módulos `variable_precision_multiplier`, `variable_precision_divider` y `variable_precision_squareroot`, respectivamente. En todos los casos las señales `STALL` y `EXCEPTION_IN` se fijan a cero, ya que el controlador asegura que sólo haya una operación activa y no se encadenan excepciones entre módulos.

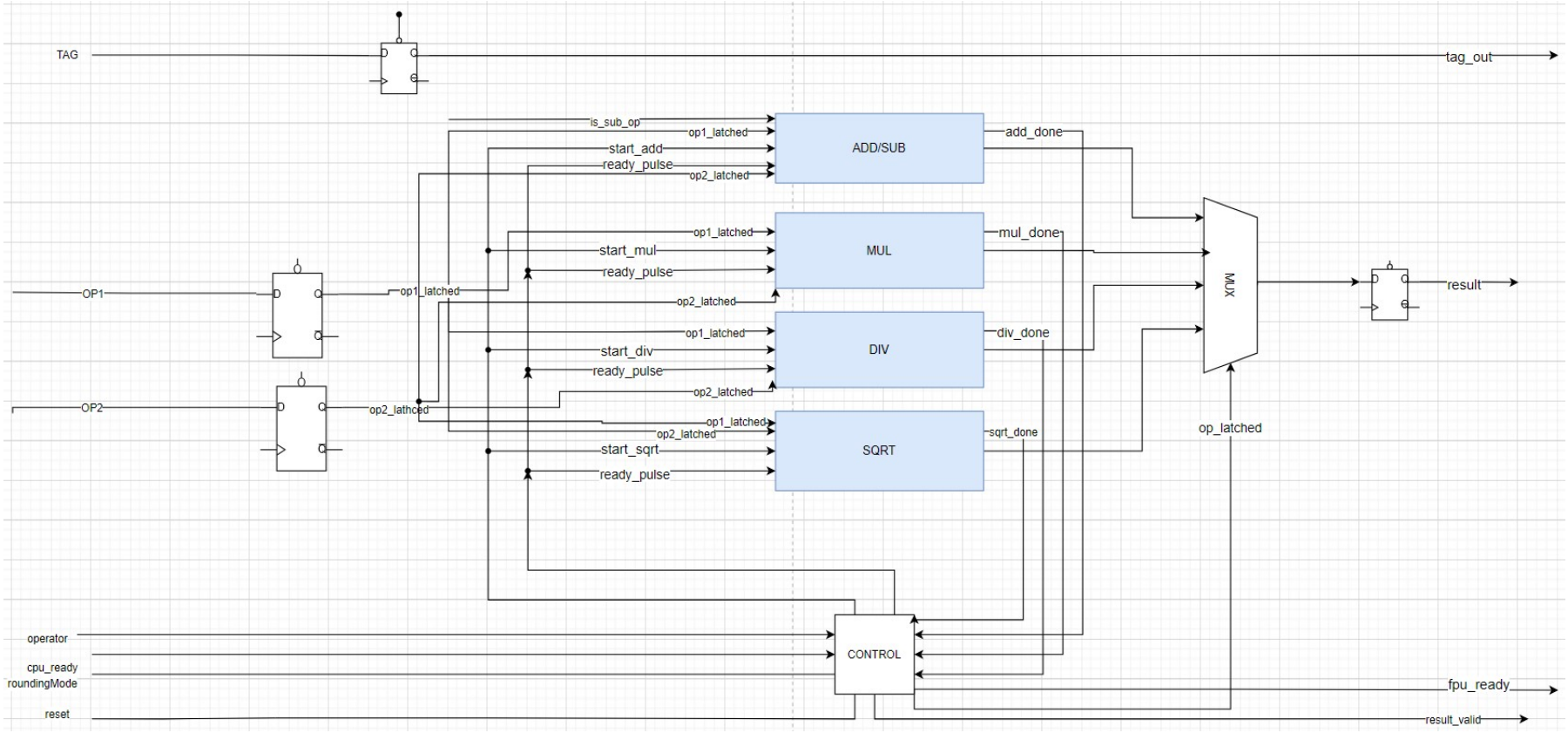

Figura 4.9: Diagrama de bloques de la FPU de módulos VFloat.

A continuación se explican las nuevas señales de esta FPU:

Señal	Dir.	Descripción
		Señales internas de control VFloat)
ready_pulse	out	Habilita la captura de operandos en los módulos VFloat.
start_add	out	Arranque del módulo de suma/resta.
start_mul	out	Arranque del módulo de multiplicación.
start_div	out	Arranque del módulo de división.
start_sqrt	out	Arranque del módulo de raíz cuadrada.
is_sub_o	out	Señal auxiliar que distingue SUB de ADD.
vfloat_round	out	Selección de redondeo simplificada (1=RNE, 0=RTZ).
a_latched	reg	Operando A registrado internamente.
b_latched	reg	Operando B registrado internamente.
		Conexión a los IPs VFloat
y_add/mul/div/sqrt	in	Resultado en formato IEEE 754 de cada módulo.
done_add/mul/div/sqrt	in	Señales de finalización de cada operación.
exc_add/mul/div/sqrt	in	Indicadores de excepción por operación.
STALL	const	Señal fijada a 0.
EXCEPTION_IN	const	Señal fijada a 0.

Tabla 4.4: Señales adicionales y específicas del módulo FPU_VFloat_Wrapper.

4.5.1. Flujo de datos y control

El flujo de datos con módulos VFloat se articula en torno a dos bloques principales: el FPU_VFloat_Wrapper, que integra las entidades VHDL de VFloat, y el FPU_VFloat_Controller, encargado de gestionar el protocolo de arranque y validación de resultados. La interfaz CPU↔FPU se ha mantenido idéntica a la usada con HardFloat (in_valid, cpu_ready, fpu_ready, result_valid, operator, rounding_mode, tag, inOp1, inOp2), lo que permite reutilizar el mismo banco de pruebas y comparar ambas arquitecturas en igualdad de condiciones.

Los operandos registrados (a_latched, b_latched) se difunden a todas las unidades, pero únicamente la unidad seleccionada recibe un pulso de entrada válido al cumplirse READY = ready_pulse ∧ start_*. De este modo no se requiere un demultiplexor adicional de operandos.

El módulo FPU_VFloat_Controller implementa una FSM con cinco estados: S_INIT, S_IDLE, S_START, S_BUSY y S_DONE.

- S_INIT: se inicializa el sistema tras un reset y se pasa a S_IDLE.
- S_IDLE, la señal fpu_ready=1 indica que la unidad está disponible. Si coinciden in_valid=1 y cpu_ready=1, se registran la operación (op_latched), el modo de redondeo (vfloat_round), el tag y los operandos (a_latched, b_latched).
- S_START: se genera un pulso de arranque común (ready_pulse=1) junto con la activación de la señal start_add, start_mul, start_div o start_sqrt,

según la operación latcheada. Este pulso tiene un ciclo de duración y equivale a la señal `READY` de los módulos VFloat.

- S_BUSY: el controlador permanece a la espera de que el bloque VFloat aserte `DONE`, que se detecta a través de la condición `op_done`.
- S_DONE: se publica el resultado con `result_valid=1` durante un ciclo y se regresa a S_IDLE.

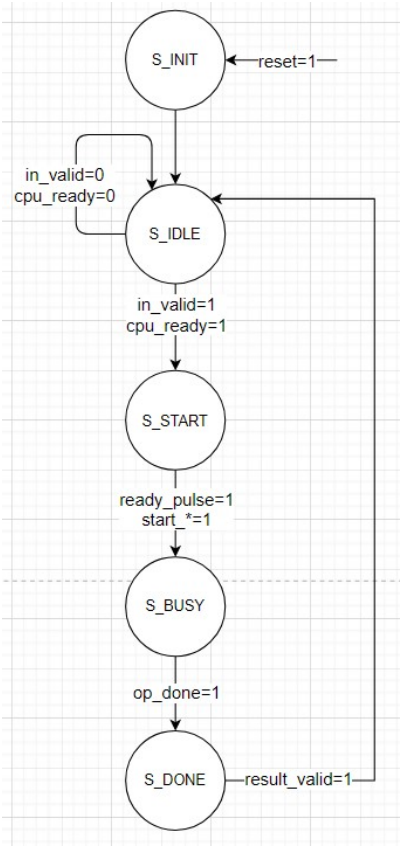

Figura 4.10: Diagrama de estados de la FSM de VFloat.

El campo `rounding_mode` de la CPU se traduce a una señal de un solo bit (`vfloat_round`) que distingue únicamente entre *Round to Nearest Even* y *Round toward Zero*. Esta limitación refleja las capacidades actuales de los módulos VFloat, y supone una diferencia frente a HardFloat, que soporta todos los modos de redondeo de IEEE 754.

Cada módulo VFloat devuelve un triplete `DONE`, `RESULT`, `EXCEPTION_OUT`. El controlador selecciona cuál propagar en función de `op_latched`. Así, las salidas `result` y `exception` siempre corresponden al bloque activo, mientras que `tag_out` refleja la etiqueta de la operación original. La señal `result_valid` se aserta exclusivamente en el estado S_DONE, durante un ciclo.

En conjunto, el flujo de datos puede resumirse en cuatro etapas: (1) aceptación de la operación en S_IDLE, (2) generación de un pulso de arranque hacia el módulo

VFloat en S_START, (3) espera de la señal `DONE` en S_BUSY, y (4) publicación del resultado con `result_valid` en S_DONE. Este mecanismo ilustra la filosofía de VFloat, basada en módulos secuenciales con *handshake* explícito, en contraste con el carácter más combinacional de HardFloat.

4.5.2. Simulación

Suma

La Figura 4.11 muestra la simulación de la operación 1.0+2.0 = 3.0. En el instante inicial, la CPU presenta los operandos en `inOp1` e `inOp2`, con valores `0x3f800000` (1.0) y `0x40000000` (2.0) en formato IEEE 754 de precisión simple, junto con el código de operación `000` (ADD) y la etiqueta `tag=0x1`.

Al activarse `in_valid` y `cpu_ready`, el controlador acepta la instrucción, latchea los operandos y genera un pulso en `ready_pulse` con `start_add=1`. Tras varios ciclos en estado `S_BUSY`, el bloque `fp_adder` aserta `done_add`, lo que provoca la transición a `S_DONE`.

El valor publicado en `result` es `0x40400000`, correspondiente a 3.0 en formato IEEE. La señal `result_valid` se activa durante un ciclo, junto con `tag_out=0x1` y `exception=0`.

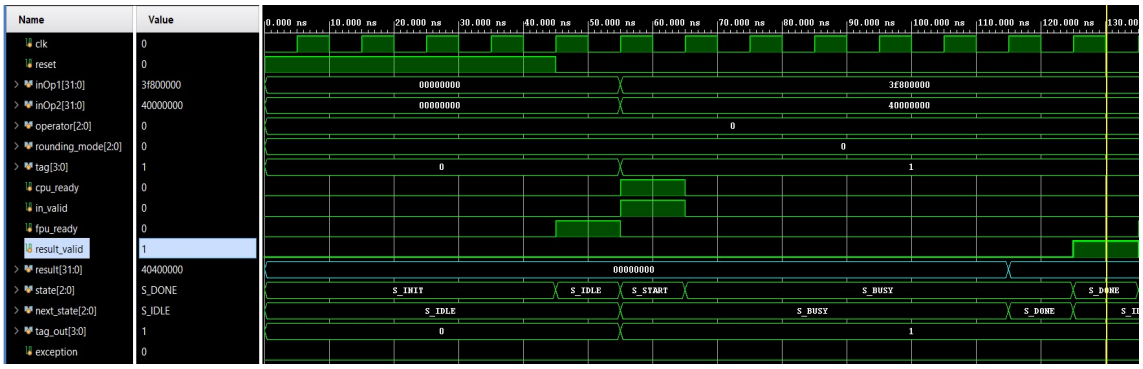

Figura 4.11: Simulación de la operación de suma 1.0 + 2.0 = 3.0 sin excepciones.

Resta

La Figura 4.12 muestra la simulación de la operación 5.5 − 2.5 = 3.0. Los operandos se cargan como `0x40b00000` (5.5) y `0x40200000` (2.5) con el código de operación `001` (SUB) y la etiqueta `tag=0x2`.

El controlador identifica la operación como resta y activa la señal auxiliar `is_sub_o`, lo que invierte el bit de signo de `inOp2`. De esta forma, la resta se implementa como una suma $A + (-B)$. Tras completarse el cálculo, el valor en `result` es `0x40400000` (3.0).

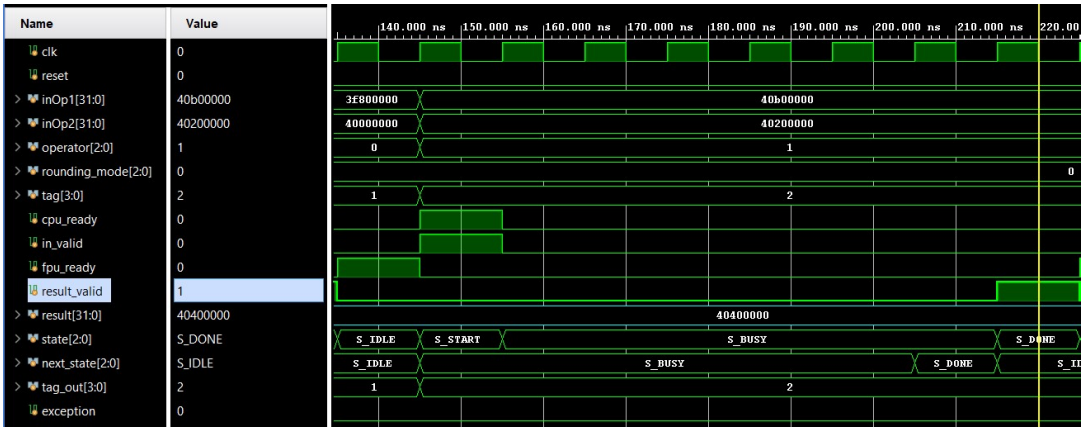

Figura 4.12: Simulación de la operación de resta $5.5 - 2.5 = 3.0$.

Multiplicación

La Figura 4.13 muestra la simulación de la operación $2.0 \times 2.0 = 4.0$. Los operandos se presentan como `0x40400000` (2.0), con el código de operación `010` (MUL) y la etiqueta `tag=0x3`.

El controlador aserta `start_mul` junto con el pulso `ready_pulse`, activando el bloque `variable_precision_multiplier`. El resultado publicado en `result` es `0x40800000`, equivalente a 4.0.

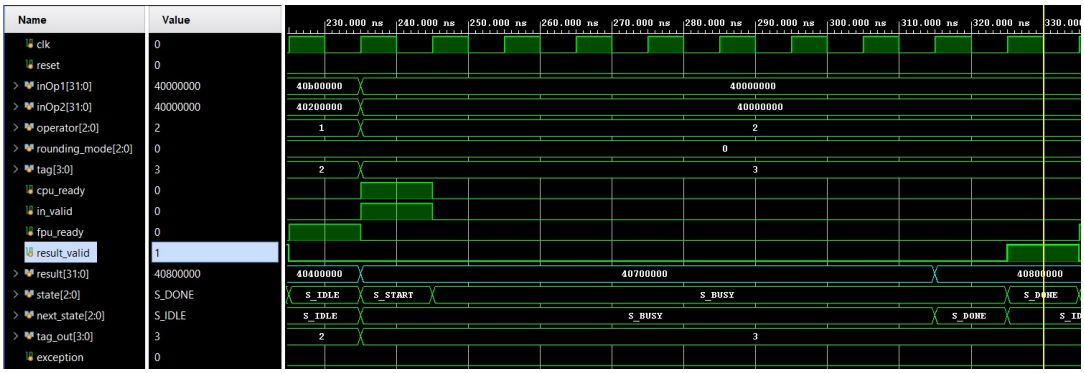

Figura 4.13: Simulación de la multiplicación $2.0 \times 2.0 = 4.0$.

División

La Figura 4.14 muestra la simulación de la operación $8.0 \div 2.0 = 4.0$. Los operandos son `0x41000000` (8.0) y `0x40000000` (2.0), con código de operación `011` (DIV) y etiqueta `tag=0x4`.

El controlador activa `start_div`, y tras la latencia propia del divisor, el resultado obtenido en `result` es `0x40800000` (4.0).

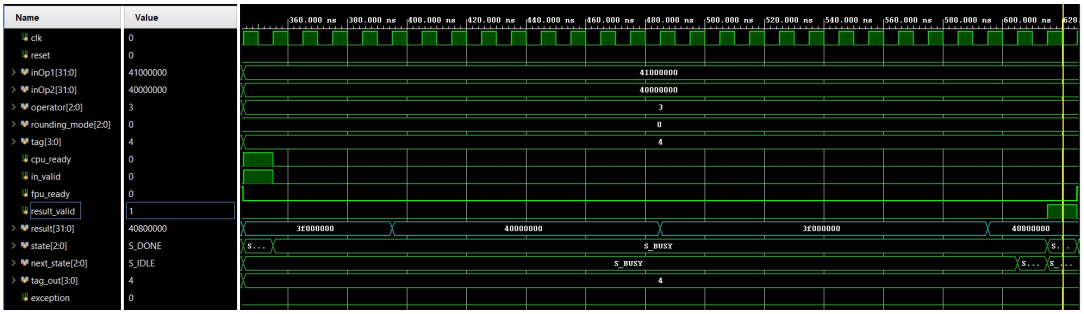

Figura 4.14: Simulación de la división $8.0 \div 2.0 = 4.0$.

Raíz cuadrada

La Figura 4.15 muestra la simulación de la operación $\sqrt{25.0} = 5.0$. El operando `inOp1` toma el valor `0x41C80000` (25.0), mientras que `inOp2` se ignora. Se utiliza el código de operación `100` (SQRT) y la etiqueta `tag=0x5`.

El controlador genera `start_sqrt` y, tras completarse el cálculo en el módulo `variable_precision_squareroot`, el valor de `result` es `0x40A00000`, correspondiente a 5.0.

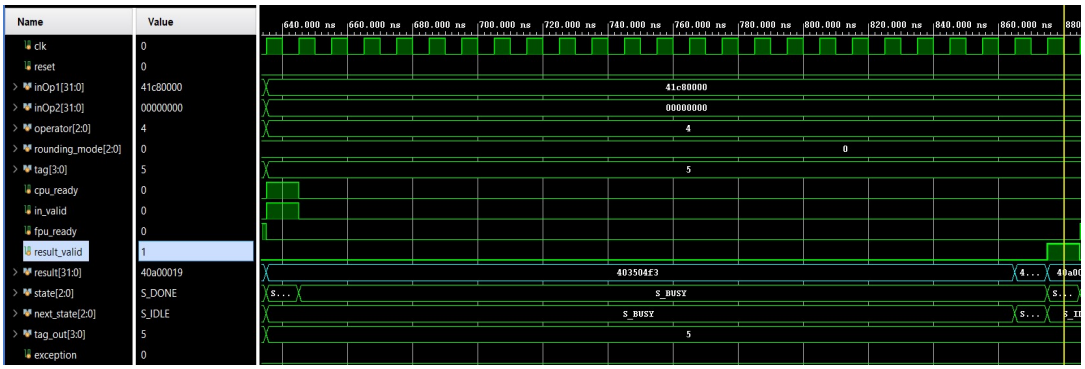

Figura 4.15: Simulación de la operación de raíz cuadrada $\sqrt{25.0} = 5.0$.

4.5.3. Resultados de síntesis

La integración en el wrapper resultó más sencilla en cuanto a interfaz. Supuso un aumento de la frecuencia, pero un mayor consumo de LUTs y FFs debido al pipeline interno de cada módulo.

Formato	LUTs	FFs	C8	DSPs	Frecuencia (MHz)	Latencia (Ciclos)
FP16	727	871	17	10	379	6/18/17
FP32	1477	1922	71	20	175	9/25/21
FP64	3810	3981	105	49	104	10/33/25

Tabla 4.5: Resultados de síntesis de la FPU VFloat.

En la Tabla 4.5 se aprecia el patrón esperado: el área aumenta con la precisión, reflejado en el mayor número de LUTs, FFs y DSPs, mientras que la frecuencia máxima se reduce progresivamente, alcanzando 379 MHz en FP16, 175 MHz en FP32 y 104 MHz en FP64. La columna Latencia muestra, en ciclos, el triplete **MUL/DIV/SQRT** para cada formato.

Estos resultados confirman el intercambio característico de VFloat: una interfaz de *handshake* y un *pipeline* interno que permiten lograr frecuencias elevadas, a costa de un mayor número de registros y una ocupación superior de recursos en comparación con HardFloat, además de una latencia más alta en las operaciones de división y raíz cuadrada.

Capítulo 5

Conclusiones y Trabajo Futuro

El objetivo principal de este proyecto era diseñar e implementar una unidad funcional de punto flotante reutilizando bloques *open source* y adaptándolos a una arquitectura modular y realista. Se han desarrollado dos variantes plenamente operativas: una FPU basada en **HardFloat** y otra basada en **VFloat**. Ambas cumplen los requisitos planteados: ejecutan correctamente las operaciones básicas del estándar IEEE-754, comparten una interfaz común y permiten comparar prestaciones y trade-offs de diseño.

Ambos diseños se han estructurado de manera clara y modular. La separación entre ruta de datos y ruta de control, el uso de conversores de formato y la encapsulación en módulos independientes facilitan el mantenimiento y permiten ampliar funcionalidades sin alterar la arquitectura general. La síntesis en distintas precisiones ha mostrado que la ocupación de recursos es baja en términos absolutos y que el diseño escala de forma predecible con el tamaño de los operandos.

Más allá de los resultados técnicos, el proyecto me ha permitido profundizar en los aspectos del estándar IEEE 754 y afianzar conocimientos tanto sobre aritmética en punto flotante como sobre el flujo de trabajo en FPGA.

Aunque el diseño alcanzado es plenamente funcional, existen varias líneas de mejora que podrían explorarse:

- **Añadir la operación FMA.** Permitirá completar el conjunto de instrucciones de la extensión RV32F y aprovechar el soporte nativo de HardFloat.
- **Introducir más etapas de *pipeline*.** El aumento de registros intermedios en la ruta de datos de la FPU HardFloat mejoraría la frecuencia máxima y, por tanto, el *throughput*.
- **Ampliar compatibilidad con IEEE 754.** Adaptar el diseño de VFloat para soportar la totalidad de modos de redondeo y el conjunto completo de excepciones del estándar.
- **Permitir operaciones simultáneas.** La duplicación parcial de unidades o

un arbitraje más avanzado abrirían la puerta a procesar varias instrucciones en paralelo.

- **Integración en un procesador RISC-V.** Conectar la FPU HardFloat como coprocesador o como parte del pipeline de un núcleo RISC-V validaría su uso en un contexto real.

En definitiva, este trabajo demuestra que es posible construir una FPU modular y escalable a partir de bloques reutilizables, con un comportamiento correcto en operaciones IEEE 754 y una estructura preparada para evolucionar hacia diseños más complejos y de mayor rendimiento. Además, la experiencia adquirida durante el desarrollo supone una base sólida para afrontar proyectos futuros en el ámbito del diseño digital y de arquitecturas hardware.

Introduction

Introduction to the subject area. This chapter contains the translation of Chapter 1.

In computing, the numerical representation plays a fundamental role because it defines how real numbers are stored and manipulated. This representation must be accurate and efficient. Today, the most widely used representations are fixed point and floating point, which will be chosen depending on the context and the needs of each application. As a general rule, the fixed-point representation provides a significant increase in performance, useful in real-time control systems, whereas if the project requires high precision, such as scientific simulations or statistical analysis, it is better to use floating point.

Floating-point arithmetic is fundamental in almost all areas of computer science and engineering. Basic operations such as addition, subtraction, multiplication and division are performed by specialised functional units known as Floating Point Units (FPUs). These FPUs are critical components in modern hardware architectures.

Due to the extensive use of floating point and with the adoption of the IEEE 754 standard, as of the year 2000 virtually all general purpose processors have a functional unit dedicated to the calculation of floating point operations. This unit is made up of different functional units for each operation and is responsible for performing arithmetic operations with decimal numbers accurately and efficiently.

There are several open source projects that implement different floating point operations. The different units do not have to interact with each other and can be treated independently, that is why to facilitate usability and encapsulation it is common to use an interface that encompasses the rest of the components, known as wrapper.

In this project I will focus on comparing the different open source floating point functional units in order to develop a complete FPU and offer a unified and efficient way to perform floating point operations.

5.1. Motivation

Given the great importance of the representation and manipulation of real numbers in the field of computing, it is essential to take into account different factors in order to operate with a numerical representation in an efficient way. The open source movement plays a key role in this field, as it encourages collaboration between researchers and developers and facilitates access to more advanced and suitable technologies for each project. However, this can also lead to confusion among researchers wishing to use one of these projects due to the wide variety of options available.

The choice and design of a suitable FPU can have a significant impact on the performance and efficiency of an application. With the widespread use of the IEEE 754 standard, FPUs have evolved considerably, providing high efficiency in floating-point calculations. However, each implementation can vary in terms of performance, accuracy, power consumption and complexity, making a detailed evaluation of them essential.

This project aims to address this issue by comparing different open source floating-point functional units. Using this information, I seek to develop a complete FPU that includes these units and is capable of operating efficiently. The aim is to provide a clear and understandable comparison that will make it easier for other researchers and developers to choose the most suitable functional units for their needs, as well as a complete FPU that is easy to use.

5.2. Objectives

The main objective of this work is to design a wrapper for floating-point operation modules, specifically addition/subtraction, multiplication and division/square root operations. To achieve this, the modules developed in open source from the Hardfloat project were used as a basis. Specific objectives include:

The main objective of this work is to design a complete FPU with floating-point modules that are easy to use. Specifically, it focuses on the operations of addition/subtraction, multiplication, and division/square root. To achieve this, the design is based on the open-source modules developed in the HardFloat and VFloat projects. The specific objectives include:

a. Investigate and evaluate the available open-source floating point processing units.

b. Compare the efficiency, cost, area and maximum frequency of the different available FPUs.

c. Select the most suitable floating-point operation modules for the development of the wrapper.

d. Design and implement a wrapper in SystemVerilog to integrate the selected modules into a hardware design environment.

5.3. Workplan

In order to carry out this work effectively, I have divided its structure into multiple stages. Each of these stages will allow me to keep track of the whole process and objectives.

1. Research on open source FPUs:
    - Conduct a comprehensive search of existing projects on available open source FPUs.
    - Identify the key features, advantages and limitations of each.
    - Collect relevant information on the efficiency, cost, area, latency and maximum operating frequency of each FPU.
2. FPU selection and comparison.
    - Select a set of candidate FPUs based on the research conducted.
    - Compare the selected FPUs in terms of efficiency, cost and maximum operating frequency.
    - Evaluate the suitability of each FPU for integration into the wrapper design.
3. **Initial fpu design.**
    - Define the wrapper input and output signals, including operand and result formats.
    - Design the wrapper control logic to coordinate the operations of the selected FPUs.
4. Implementation and simulation of the final design in SystemVerilog:.
    - Develop the code in SystemVerilog that implements the previously defined wrapper design.
    - Perform simulations to verify the correct operation of the wrapper under different input conditions.
    - Debug and resolve any compatibility or incompatibility issues that may arise during implementation.
5. **Verification and validation of the wrapper.**
    - Perform unit tests to verify the correct functioning of the wrapper in different input scenarios.
    - Perform integration tests to ensure compatibility and consistency of the wrapper with the selected FPUs.

Conclusions and Future Work

The main objective of this project was to design and implement a floating-point functional unit by reusing *open-source* blocks and adapting them to a modular and realistic architecture. The final result confirms that the proposed requirements have been met: the FPU correctly executes the basic operations of the IEEE 754 standard.

The design has been structured in a clear and modular way. The separation between data path and control path, the use of format converters, and the encapsulation into independent modules facilitate maintenance and allow functionalities to be extended without altering the overall architecture. Synthesis at different precisions has shown that the resource usage is low in absolute terms and that the design scales predictably with operand size.

Beyond the technical results, this project has allowed me to deepen my understanding of the IEEE 754 standard and to consolidate knowledge both on floating-point arithmetic and on the FPGA design flow.

Although the achieved design is fully functional, several lines of improvement could be explored:

- **Add the FMA operation.** This would complete the instruction set of the RV32F extension and take advantage of HardFloat's native support.
- **Introduce additional pipeline stages.** Increasing the number of intermediate registers in the data path would improve the maximum frequency and thus the overall throughput.
- **Enable simultaneous operations.** Partial duplication of units or more advanced arbitration would make it possible to process multiple instructions in parallel.
- **Integration into a RISC-V processor.** Connecting the FPU as a coprocessor or as part of a RISC-V core pipeline would validate its use in a real context.

In summary, this work demonstrates that it is possible to build a modular and scalable FPU from reusable blocks, with correct behavior for IEEE 754 operations

and a structure prepared to evolve towards more complex and higher-performance designs. Furthermore, the experience gained during the development provides a solid foundation for addressing future projects in the field of digital design and hardware architectures.

Bibliografía

Al-Eryani, J. Ieee 754 floating point unit. `https://opencores.org/projects/fpu100`, 2000. OpenCores project documentation.

Bruguera, J.. Radix-64 floating-point divider. En *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, páginas 84–91. 2018.

Dawson, J. Synthesiseable ieee 754 floating point library in verilog. `https://github.com/dawsonjon/fpu`, 2013.

Dinechin, F. y Pasca, B. Designing custom arithmetic data paths with flopoco. *IEEE Design Test of Computers*, vol. 28(4), páginas 18–27, 2011.

Ercegovac, M., Lang, T., Muller, J.-M. y Tisserand, A. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, vol. 49(7), páginas 628–637, 2000.

Guthmuller, E., Fuguet, C., Bocco, A., Fereyre, J., Alidori, R., Tahir, I. y Durand, Y. Xvpfloat: Risc-v isa extension for variable extended precision floating point computation. *IEEE Transactions on Computers*, vol. 73(7), páginas 1683–1697, 2024.

Hauser, J. Hardfloat – parameterized modules for floating-point arithmetic. `https://www.jhauser.us/arithmetic/HardFloat.html`, 2019.

IEEE Computer Society. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, páginas 1–84, 2019.

Mach, S., Schuiki, F., Zaruba, F. y Benini, L. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29(4), páginas 774–787, 2021.

Malossi, A. C.., Schaffner, M., Molnos, A., Gammaitoni, L., Tagliavini, G., Emerson, A., Tomás, A., Nikolopoulos, D.§., Flamand, E. y Wehn, N. The transprecision computing paradigm: Concept, design, and applications. En *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, páginas 1105–1110. 2018.

Muller, J.-M., Brunie, N., de, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N. y Torres, S. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2 edición, 2018a.

Muller, J.-M., Brunie, N., de, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N. y Torres, S. The ulp function. En *Handbook of Floating-Point Arithmetic, Chapter 2.3.2* (editado por J.-M. Muller), páginas 29–34. Birkhäuser, 2nd edición, 2018b.

Oberman, S. Floating point division and square root algorithms and implementation in the amd-k7/sup tm/ microprocessor. En *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*, páginas 106–115. 1999.

Obermann, S. y Flynn, M. Division algorithms and implementations. *IEEE Transactions on Computers*, vol. 46(8), páginas 833–854, 1997.

Patankar, U.§. y Koel, A. Review of basic classes of dividers based on division algorithm. *IEEE Access*, vol. 9, páginas 23035–23069, 2021.

Soderquist, P. y Leeser, M. Division and square root: choosing the right implementation. *IEEE Micro*, vol. 17(4), páginas 56–66, 1997.

Ussleman, R. Ieee 754 floating point unit. `https://opencores.org/websvn/filedetails?repname=fpu&path=%2Ffpu%2Ftrunk%2FFPU.pdf`, 2000. OpenCores project documentation.

Wang, X. y Leeser, M. Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware. *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3(3), 2010. ISSN 1936-7406.

Öksüz, T. Fpu single and double precision. `https://github.com/taneroksuz/fpu`, 2019.

Apéndice A

Apéndice A

A.1. Resultados completos de FPnew

En esta sección se incluyen los resultados detallados de síntesis para FPnew, variando los parámetros `Implementation` (MERGED/PARALLEL) y `PipeRegs` en cada formato (FP16, FP32, FP64 y FP128).

PipeRegs	**Config**	**FP16**			**FP32**			**FP64**			**FP128**		
		MHz	LUTs	DSP	MHz	LUTs	DSP	MHz	LUTs	DSP	MHz	LUTs	DSP
0	Parallel	75.5	565	1	35.9	1049	2	40.5	2377	9	25.1	5506	49
0	Merged	41.5	696	1	32.2	1269	2	38.7	2485	9	22.7	5698	49
1	Parallel	775	553	1	775	1228	2	775	2281	9	775	5563	49
1	Merged	775	659	1	775	1195	2	775	2551	9	358	5758	49
2	Parallel	775	555	1	775	1229	2	775	2283	9	775	5562	49
2	Merged	775	609	1	775	1197	2	775	2554	9	418	5756	49

Tabla A.1: Resultados completos de síntesis de FPnew (MHz redondeados).

En todas las configuraciones evaluadas, la frecuencia máxima alcanzable está limitada por la métrica `WPWS` (*Worst Pulse Width Slack*), mientras que `WNS` permanece positiva. Esto indica que el factor limitante no es la ruta más lenta del diseño, sino la anchura mínima de los pulsos del reloj. En la práctica, esto impone un techo a `Fmax` independiente de la lógica crítica del diseño.

A.2. Licencias y aspectos legales

En este trabajo se han utilizado dos bibliotecas externas con licencias diferentes: HardFloat, bajo BSD 3-Clause (licencia permisiva), y VFloat, bajo GPL (copyleft fuerte).

En consecuencia:

- La **memoria del TFG** se publica con licencia **CC BY 4.0**.

- El **código propio que integra HardFloat** se distribuye con licencia **MIT**.
- El **código propio que integra VFloat** se distribuye con licencia **GPLv3**.

De esta forma se respeta la licencia original de cada componente y se facilita la reutilización del trabajo.