UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de soporte de cálculo para Unidad Vectorial de RISCV

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Mas Faus, Ferran

Tutor/a: Flich Cardo, José

Cotutor/a: López Rodríguez, Pedro Juan

CURSO ACADÉMICO: 2023/2024

Resum

Aquest treball té com a objectiu el disseny, desenvolupament i verificació d'una unitat de càlcul vectorial segmentada per a una arquitectura vectorial basada en RISC-V. La unitat de càlcul proposta ofereix suport integral per a una varietat d'operacions que considerem claus en coma flotant. Entre estes operacions es troben la suma, la resta, la multiplicació, la divisió y *fused multiply-add* (FMA). Cada component ha sigut dissenyat amb l'objectiu d'oferir suport a excepcions i proporcionar varies opcions d'arredoniment que asseguren precisió i fiabilitat en els cálculs.

El marc principal de treball d'aquesta unitat son els valors en coma flotant de simple precisió (32 bits). A més, es compatible amb els formats de dades Tensor Float 32 (TF32) i Bfloat 16 (BF16), conseguint ampliar la seua versatilitat i aplicabilitat, especialment en cálculs relacionats amb el món de la inteligència artificial, on el us d'aquests formats està en apogeu.

Per a la verificació i validació del correcte funcionament de la unitat, hem dut a terme la seua integració en una unitat vectorial, junt amb una evaulació detallada emprant el software Xilinx Vivado. D'aquesta forma conseguim arreplegar informació necessaria per elaborar un análisis dels recursos necessaris per a la seua implementació. Aquest procés ens ha proporcionat una visió aproximada dels recursos requerits per la unitat en el cas de que s'implantara en un chip o en una FPGA.

Paraules clau: RISC-V, FPU, Verilog, Arquitectura de Computadors.

Resumen

El presente trabajo tiene como objetivo el diseño, desarrollo y verificación de una unidad de cálculo vectorial segmentada para una arquitectura vectorial basada en RISC-V. La unidad de cálculo propuesta ofrece soporte integral para una variedad de operaciones que consideramos clave en coma flotante. Entre estas operaciones están la suma, la resta, la multiplicación, la división y *fused multiply-add* (FMA). Cada componente ha sido diseñado con el objetivo de ofrecer soporte a excepciones y proporcionar varias opciones de redondeo que aseguran precisión y fiabilidad en los cálculos.

El marco principal de trabajo de esta unidad son los valores en coma flotante de simple precisión (32 bits). Además, es compatible con los formatos de datos Tensor Float 32 (TF32) y Bfloat 16 (BF16), lo que consigue ampliar su versatilidad y aplicabilidad, especialmente en cálculos relacionados con el mundo de la inteligencia artifical, donde el uso de estos formatos de datos está en auge.

Para la verificación y validación del correcto funcionamiento de la unidad, hemos llevado a cabo su integración en una unidad vectorial, junto con una evaluación detallada usando el software Xilinx Vivado. De esta forma conseguimos recopilar información necesaria para elaborar un análisis de los recursos necesarios para su

implementación. Este proceso nos proporcionó una visión aproximada de los recursos requeridos por la unidad en el caso de que se implantara en un chip o en una FPGA.

Palabras clave: RISC-V, FPU, Verilog, Arquitectura de Computadores.

Abstract

The aim of this work is to design, develop and verify a segmented vector computing unit for a RISC-V based vector architecture. The proposed computational unit provides support for a variety of operations that we consider key floating point operations. Among these operations are addition, subtraction, multiplication, division and fused multiply-add (FMA). Each component has been designed with the objective of supporting exceptions and providing various rounding options to ensure accurate and reliable results.

The main framework of this unit is single-precision (32-bit) floating point values. In addition, it supports the Tensor Float 32 (TF32) and Bfloat 16 (BF16) data formats, extending its versatility and applicability, especially in computations related to the world of artificial intelligence, where the use of these data formats is booming.

In order to verify and validate the correct functioning of the unit, we have carried out its integration into a vector unit, together with a detailed evaluation using the Xilinx Vivado software. In this way we were able to collect necessary information to prepare an analysis of the resources required for its implementation. This process provided us with an approximate view of the resources required by the unit in the case of an on-chip or FPGA implementation.

Key words: RISC-V, FPU, Verilog, Computer Architecture.

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En el mundo de la arquitectura y la tecnología de computadores, dos conceptos fundamentales han guiado la evolución constante en el diseño y el rendimiento de los chips computacionales: el Dennard-Scaling y la ley de Moore.

El Dennard Scaling[1], nombrado en honor al ingeniero Robert Dennard, describe cómo la eficiencia energética de los transistores mejora con el tiempo. Esta teoría sostiene que, a medida que se reducen las dimensiones de los transistores en un circuito integrado, la densidad de corriente permanece constante. Esto resulta en un menor consumo de energía y una disipación de calor más eficiente, permitiendo el desarrollo de dispositivos cada vez más potentes sin un aumento significativo en el consumo de energía.

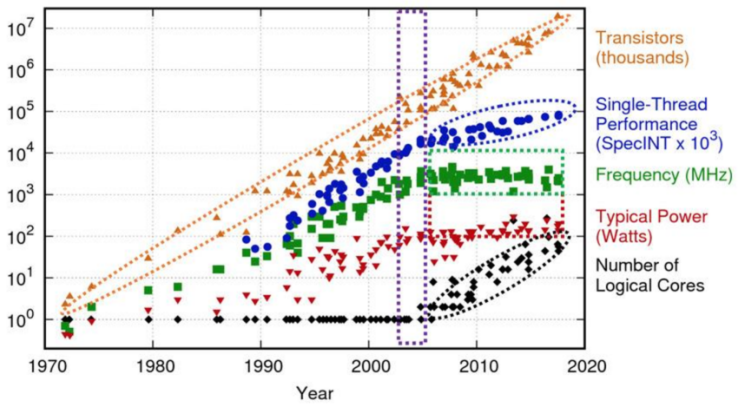

Figura 1.1: Tendencias en arquitectura de computadores.

Por otro lado, la Ley de Moore[2], formulada por Gordon Moore, co-fundador de Intel, predice un crecimiento exponencial en la densidad de los circuitos integrados. Según esta ley empírica, la cantidad de transistores en un chip se duplica aproximadamente cada dos años, lo que implica un aumento notable en la capacidad de procesamiento y almacenamiento de los dispositivos electrónicos. Esta tendencia ha permitido la creación de arquitecturas de chips más complejas y

[1] `https://www.rambus.com/blogs/understanding-dennard-scaling-2/`
[2] `https://ourworldindata.org/moores-law`

eficientes, y ha impulsado el desarrollo de tecnologías innovadoras que han revolucionado diversas industrias.

Sin embargo, como podemos ver en la figura 1.1 con los triángulos naranja representando el comportamiento de la Ley de Moore y en triángulos rojos el Dennard Scaling, a partir de los años 2000, el cumplimiento del Dennard Scaling comienza a disminuir. Esto se debe a que, al reducir las dimensiones de los transistores más allá de un cierto punto, los efectos de la corriente de fuga y otros fenómenos físicos provocan un incremento en el consumo de energía y la disipación de calor, limitando así la eficiencia energética que antes se lograba.

Gracias a estas leyes, hemos sido testigos de chips cada vez más pequeños y con un rendimiento mejorado. Estos avances han impulsado la rápida evolución de dispositivos informáticos, desde computadoras personales hasta dispositivos móviles y sistemas embebidos. Además, han facilitado el desarrollo de tecnologías emergentes como el Internet de las cosas (IoT)[3], la inteligencia artificial (IA)[4] y el aprendizaje automático (*machine learning*)[5], que requieren un procesamiento intensivo de datos en tiempo real.

En este contexto de progreso tecnológico continuo, surge la necesidad de desarrollar unidades de procesamiento más avanzadas y versátiles. Un ejemplo claro de esta evolución es la GPU (Unidad de Procesamiento Gráfico)[6]. Las GPU han ganado popularidad en el ámbito del procesamiento paralelo y la computación de alto rendimiento debido a su capacidad para realizar operaciones intensivas en paralelo. Estas unidades están diseñadas específicamente para tareas que requieren un alto grado de procesamiento, como la renderización de gráficos 3D, el procesamiento de video y la minería de criptomonedas.

Otro enfoque prometedor en el diseño de unidades de procesamiento es el uso de arrays sistólicos (figura 1.2)[7], siendo arquitecturas especializadas para ejecutar operaciones matriciales de manera eficiente. Estas estructuras se basan en la comunicación de datos a través de un conjunto de células de procesamiento interconectadas, lo que las hace particularmente adecuadas para aplicaciones de redes neuronales.

Un ejemplo destacado de este tipo de arquitectura es la TPU (*Tensor Processing Unit*) de Google[6], que combina arrays sistólicos con unidades vectoriales para optimizar el rendimiento en tareas de inteligencia artificial y aprendizaje automático. Esta combinación permite una mayor flexibilidad y eficiencia en el procesamiento de datos, ya que las unidades vectoriales pueden complementar las operaciones realizadas por el array sistólico, mejorando así el rendimiento general del sistema.

[3] https://www.redhat.com/es/topics/internet-of-things/what-is-iot
[4] https://planderecuperacion.gob.es/noticias/que-es-inteligencia-artificial-ia-prtr
[5] https://www.ibm.com/es-es/topics/machine-learning
[6] https://www.ibm.com/topics/gpu
[7] https://www.sciencedirect.com/topics/engineering/systolic-arrays

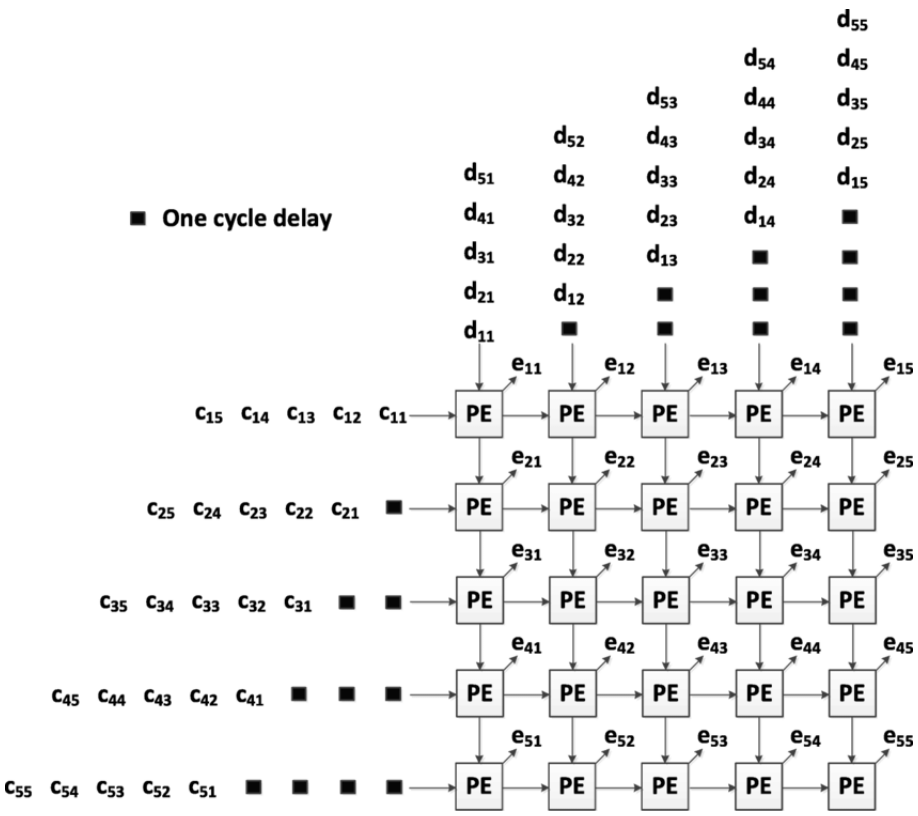

Figura 1.2: Array sistólico.

Estas unidades vectoriales están empezando a desempeñar un papel fundamental en la industria, al permitir operaciones simultáneas en conjuntos de datos, consiguiendo acelerar significativamente en tareas intensivas de cálculos con gran volúmenes de datos y mejorar su eficiencia.

En este escenario es donde se presenta nuestra unidad de cálculo de coma flotante segmentada, diseñada para proporcionar soporte de cálculo a una unidad vectorial basada en la arquitectura RISC-V[10]. Esta unidad representa un avance significativo en el diseño de unidades de procesamiento, ya que ofrece un conjunto completo de operaciones aritméticas esenciales, incluyendo sumador/restador, multiplicador, divisor y *fused multiply-add*.

Nuestra unidad de cálculo de coma flotante es compatible con una variedad de formatos de datos, incluyendo números en coma flotante de 32 bits, TF32[8] y BFloat16[9]. Esto permite una mayor flexibilidad y eficiencia en el procesamiento de datos en una amplia gama de aplicaciones, desde el procesamiento de señales hasta la inteligencia artificial y el aprendizaje automático. La segmentación de los operadores en nuestra unidad de cálculo garantiza una ejecución eficiente de operaciones vectoriales, lo que la hace especialmente adecuada para aplicaciones que requieren un alto rendimiento en el procesamiento de datos en paralelo.

La mejora de rendimiento causada por el uso de unidades vectoriales se debe a la mitigación del cuello de botella de Flynn[2]. En la arquitectura clásica de

[8] `https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/`
[9] `https://cloud.google.com/tpu/docs/bfloat16?hl=es-419`

computadores, el cuello de botella de Flynn se refiere a la limitación del rendimiento debido a la necesidad de buscar una instrucción por cada dato procesado. Aunque los procesadores superescalares pueden buscar y ejecutar varias instrucciones por ciclo de reloj, la detección y resolución de dependencias entre estas instrucciones limitan la cantidad de instrucciones que se pueden ejecutar simultáneamente.

El procesamiento vectorial mitiga este cuello de botella al permitir que una única instrucción vectorial se aplique a un conjunto de datos (vectores). La operación codificada en la instrucción se repite para cada elemento del vector, y las operaciones en cada componente no tienen dependencias entre sí. Tras decodificar la instrucción, la ejecución se vuelve altamente eficiente mediante el uso de múltiples unidades aritméticas o unidades aritméticas segmentadas. Si la instrucción implica acceso a memoria, se accede de manera eficiente a un bloque de datos, en este caso vectores, mejorando significativamente el rendimiento global del sistema.

Para maximizar aún más el rendimiento y la eficiencia de las unidades de procesamiento, los desarrolladores pueden integrar IP cores. Los IP cores de Xilinx[10] son módulos predefinidos y optimizados que se pueden incorporar directamente en FPGA[11], proporcionando un rendimiento superior y facilitando el desarrollo de sistemas complejos. Estos IP cores están diseñados para manejar tareas específicas con alta eficiencia, reduciendo la carga de trabajo del diseño personalizado y permitiendo a los desarrolladores centrarse en la integración y la funcionalidad del sistema en su conjunto.

Aunque la integración de IP cores de Xilinx podría proporcionar un rendimiento superior, hemos decidido seguir un enfoque independiente en el desarrollo de nuestra unidad de cálculo de coma flotante. Este enfoque nos brinda una libertad creativa y técnica sin restricciones, permitiéndonos diseñar y verificar cada operador de forma personalizada según nuestras necesidades específicas. Además, esta independencia nos ofrece la flexibilidad de adaptar nuestro diseño para su implementación tanto en ASIC(*application-specific integrated circuit*)[12] como en FPGA, según las exigencias del proyecto y las preferencias del cliente.

En el transcurso de este trabajo, nos dedicaremos a explorar en detalle el proceso de diseño, verificación y evaluación de rendimiento de nuestra unidad de cálculo vectorial de coma flotante.

[10] `https://www.xilinx.com/products/intellectual-property.html`

[11] `https://digilent.com/blog/what-is-an-fpga/`

[12] `https://www.utmel.com/blog/categories/integrated20circuit/what-is-an-asic-chip`

1.1 Contexto y Motivación

El Grupo de Arquitecturas Paralelas (GAP)[13]de la Universitat Politècnica de València (UPV) se ha consolidado como un centro de investigación de referencia en el ámbito de la computación paralela y arquitecturas avanzadas. Con una trayectoria sólida y reconocida, el GAP se ha destacado por su contribución al avance de la computación paralela a nivel nacional e internacional.

Además de su labor investigadora, el GAP desempeña un papel activo en la comunidad académica y profesional al colaborar en la creación de herramientas y recursos para la enseñanza y la formación en el campo de la computación paralela. Esto incluye la elaboración de materiales didácticos, la presencia en multitud de eventos y la participación en iniciativas de divulgación y educación en tecnología.

En este contexto, la arquitectura RISC-V emerge como una plataforma fundamental en el ámbito de la investigación y desarrollo de sistemas de computación avanzada. RISC-V, desarrollado en la Universidad de California, Berkeley, es una arquitectura de conjunto de instrucciones (ISA) abierta y modular que ha ganado una creciente atención en la comunidad de diseño de chips debido a sus características únicas y su enfoque flexible.

Una de sus principales ventajas es la capacidad de personalización, permitiendo a los diseñadores crear procesadores altamente especializados para una amplia gama de aplicaciones, desde dispositivos integrados hasta servidores de alto rendimiento.

Dentro del repertorio de instrucciones de RISC-V, las instrucciones vectoriales juegan un papel importante. Estas permiten realizar operaciones simultáneas en conjuntos de datos, acelerando significativamente el procesamiento en aplicaciones intensivas en cálculos, como la inteligencia artificial y el aprendizaje automático. Las unidades vectoriales basadas en RISC-V aprovechan estas capacidades para mejorar el rendimiento y la eficiencia en el manejo de datos.

La unidad de cálculo vectorial que se desarrolla en este Trabajo de Final de Grado se enmarca dentro de este entorno innovador y está diseñada para proporcionar soporte a una unidad vectorial basada en esta arquitectura de conjunto de instrucciones. Esta unidad representa una contribución significativa a la investigación en arquitecturas de computación paralela.

Una de las plataformas clave para el desarrollo y prueba de estas unidades es la FPGA (*Field-Programmable Gate Array*). Estos dispositivos (figura 1.3) proporcionan un entorno altamente flexible y reconfigurable que permite a los diseñadores probar y depurar sus diseños en tiempo real. Esta capacidad de testeo iterativo y exploración de una amplia gama de arquitecturas de hardware es fundamental para el desarrollo de sistemas complejos. Las FPGA permiten evaluar y comparar múltiples versiones de un diseño, identificar la mejor solución en términos de

[13] `https://www.gap.upv.es/`

rendimiento, consumo de energía y costo, y realizar ajustes y mejoras de manera rápida y eficiente.

En este sentido, la herramienta Xilinx Vivado juega un papel fundamental al proporcionar un entorno integrado para el diseño, implementación y verificación de sistemas en FPGA. Con características avanzadas de síntesis y simulación. Vivado simplifica y agiliza el proceso de desarrollo de hardware, permitiendo a los diseñadores centrarse en la innovación y la optimización de sus diseños.

Figura 1.3: FPGA propietaria de AMD y utilizada dentro del grupo de investigación.

1.2 Objetivos

El proyecto tiene como objetivo principal el desarrollo de una unidad de coma flotante segmentada de 32 bits, destinada a brindar soporte de cálculo a una unidad vectorial basada en la arquitectura RISC-V. Esta unidad estará compuesta por varios componentes esenciales, cada uno diseñado para proporcionar un buen rendimiento y una precisión adecuada en una variedad de operaciones aritméticas clave.

Dentro de este objetivo principal, podemos encontrar los siguientes objetivos específicos:

- **Diseño de la unidad de coma flotante segmentada:** El primer objetivo del proyecto es diseñar cada componente de la unidad de coma flotante de manera segmentada, asegurando una ejecución eficiente de operaciones aritméticas como suma/resta, multiplicación, división y *fused multiply-add* (FMA).
- **Garantizar la precisión y el redondeo correcto**: Se buscará implementar mecanismos para garantizar el redondeo correcto y un tratamiento adecuado de excepciones, conforme al estándar IEEE 754 para la aritmética de coma flotante.
- **Soporte para operaciones con máscaras según RISC-V**: Se desarrollará la capacidad de la unidad para ejecutar operaciones condicionales basadas en

máscaras de datos, siguiendo las especificaciones marcadas por la extensión vectorial de la arquitectura RISC-V.

- **Verificación de rendimiento y eficiencia:** Se llevarán a cabo varias pruebas para evaluar el rendimiento y la eficiencia de la unidad en diferentes escenarios de uso. Se analizarán métricas como la utilización de recursos para asegurar un funcionamiento adecuado.

- **Flexibilidad de implementación:** Se diseñará la unidad con la capacidad de ser implementada tanto en dispositivos ASIC como en FPGA, brindando así la flexibilidad necesaria para adaptarse a diferentes requisitos de aplicación y entornos de implementación.

- **Ampliar soporte a formatos de datos de redes neuronales:** Este objetivo se centra en expandir la capacidad de la unidad de coma flotante para adaptarse a los formatos de datos más recientes en coma flotante, especialmente aquellos utilizados en redes neuronales para entrenamiento e inferencia. Se trabajará en la integración de los formatos TF32 y BFloat16, que son cada vez más prevalentes en aplicaciones de inteligencia artificial y aprendizaje profundo. El objetivo es garantizar una compatibilidad total con los estándares actuales y futuros de procesamiento de datos neuronales.

- **Integración exitosa de la unidad de cálculo en una unidad vectorial:** Este objetivo busca la incorporación efectiva de la unidad de cálculo de coma flotante dentro de una unidad vectorial, asegurando su correcto funcionamiento y su armonización con el procesamiento vectorial. Se llevará a cabo un proceso de diseño e integración, respaldado por pruebas para verificar la compatibilidad y el rendimiento de ambas unidades en conjunto. El objetivo es garantizar un correcto desempeño en diversos escenarios de aplicación.

1.3 Estructura de la Memoria

- **Capítulo 1, Introducción:** En este capítulo se presentará el contexto y la motivación que han servido de base para el desarrollo del proyecto, así como sus principales objetivos y las colaboraciones que han contribuido a su exitosa conclusión.

- **Capítulo 2, Conceptos Previos:** Se proporcionará una presentación concisa de los conocimientos fundamentales necesarios para comprender los temas tratados en este trabajo, con el objetivo de que el lector pueda contextualizar y asimilar el contenido que se desarrollará en los capítulos siguientes.

- **Capítulo 3, Análisis del Problema:** En este capítulo se expondrán las dificultades o desafíos inherentes al diseño, implementación y funcionamiento de la unidad vectorial de coma flotante.

- **Capítulo 4, Diseño de los componentes:** Se analizará la arquitectura y la estrategia adoptada para crear cada componente que forma parte de la unidad. Se describirán las decisiones de diseño que tomamos y las razones que

las respaldan, centrándonos especialmente en buscar la mejor relación posible entre eficiencia y rendimiento.

- **Capítulo 5, Verificación y resultados:** En este capítulo, se documentarán las pruebas llevadas a cabo en cada componente para confirmar su correcto desempeño. Se detallarán los métodos de verificación empleados, que incluyen pruebas funcionales y simulaciones, así como los resultados obtenidos en términos de precisión.
- **Capítulo 6, Conclusiones:** Se destacarán los objetivos alcanzados durante el desarrollo del proyecto, las dificultades enfrentadas, la relación con los estudios cursados y además, se discutirán las posibles áreas de mejora y las oportunidades para trabajos futuros, incluyendo posibles ampliaciones o extensiones del trabajo actual.

1.4 Colaboraciones

En el apartado de colaboraciones, es imprescindible destacar el valioso aporte realizado por Daniel Chinesta Nuñez en el marco de este proyecto. Chinesta no solo ha sido mi compañero de prácticas de empresa en el Grupo de Arquitecturas Paralelas, sino también un colaborador cercano y un amigo de confianza. Su profundo compromiso y dedicación hacia el desarrollo tecnológico han sido evidentes a lo largo de nuestra colaboración.

Este proyecto de unidad de coma flotante es una parte de un proyecto mucho más ambicioso: el desarrollo de una unidad vectorial basada en la arquitectura RISC-V. Dada la complejidad inherente al diseño, hemos decidido dividir la unidad en dos partes claramente diferenciadas, asegurándonos de evitar dependencias entre ellas para facilitar su desarrollo y mantenimiento.

Por un lado, Chinesta ha sido responsable de la faceta de control de datos, registros y memoria. Su trabajo abarca todo lo relacionado con la gestión y organización de la información dentro del sistema, garantizando que los datos se manejen de manera eficiente y precisa.

Por otro lado, mi responsabilidad ha sido desarrollar el soporte de cálculo vectorial en coma flotante. Esta parte del proyecto se enfoca en implementar las operaciones matemáticas necesarias para manejar vectores de números en coma flotante, optimizando el rendimiento y garantizando la precisión de los resultados.

CAPÍTULO 2
Conceptos Previos

En este capítulo, proporcionaremos una visión general de los conceptos esenciales necesarios para comprender a fondo nuestro trabajo, centrándonos en los números en coma flotante y el estándar IEEE 754, especialmente en su formato de 32 bits, así como las reglas de redondeo y el manejo de excepciones. También abordaremos los formatos adicionales soportados que van más allá del estándar, y exploraremos cómo se integran en el contexto del procesamiento vectorial.

Dentro del procesamiento vectorial, destacaremos dos técnicas cruciales: el enmascaramiento y la técnica de segmentación, que desempeñan un papel fundamental en la ejecución eficiente de las instrucciones vectoriales.

2.1 Números de Coma Flotante

La representación de los números en coma flotante, como se observa en la figura 2.1, se basa en la notación científica, un método poderoso y versátil que permite representar números muy grandes o muy pequeños de manera compacta y precisa. Esta forma de representación, fundamental en el diseño de sistemas numéricos, ha sido empleada a lo largo de la historia de la computación para abordar una amplia gama de problemas en diversas disciplinas, desde la ingeniería hasta la física y las ciencias de la computación.

En la notación científica, la coma decimal no se encuentra en una posición fija en la secuencia de bits, sino que su posición se indica como una potencia de la base. Todos los números en coma flotante están compuestos por tres componentes esenciales: el signo, la mantisa y el exponente. Esta estructura modular permite representar tanto números muy grandes como muy pequeños de manera eficiente, adaptándose a las necesidades de cálculo de diversas aplicaciones.

En la figura 2.2, se presenta de manera gráfica el rango de valores representables en coma flotante. Es importante comprender que la representación en coma flotante utiliza un número finito de bits, lo que implica que el rango de valores que pueden ser representados también es limitado. Este rango se extiende desde el valor máximo posible del exponente hasta el valor mínimo del mismo, abarcando todas las posibles combinaciones de exponente y mantisa en este intervalo.

Es importante destacar la existencia de los números denormalizados dentro de este espectro. Estos números se encuentran en un rango especial, situado entre cero y el número más pequeño (tanto positivo como negativo) que puede ser representado por números normales en el formato de coma flotante.

Por último, es necesario considerar los desbordamientos positivos (*overflows)* y negativos (*underflows*), los cuales ocurren cuando los resultados de las operaciones exceden los límites máximo y mínimo del rango de representación en coma flotante, respectivamente.

Figura 2.1: Número en coma flotante.

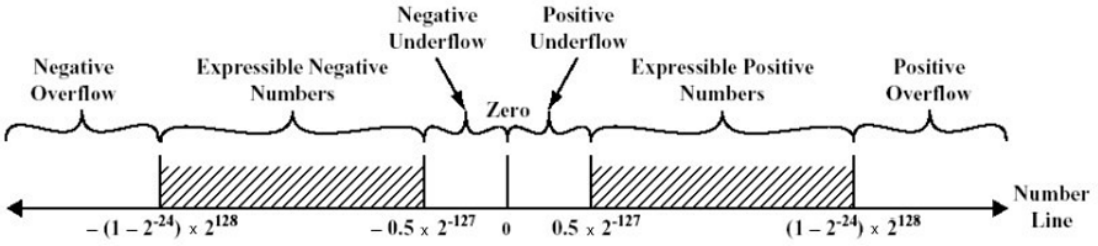

Figura 2.2: Rango de valores en coma flotante.

La variedad en la implementación de sistemas de coma flotante, como los utilizados por computadoras icónicas como el PDP10[1] con base 8, así como el IBM 360[2] con base 16, puso de manifiesto la necesidad urgente de un estándar unificado en la representación de números en coma flotante. Esta diversidad de formatos dificultaba la interoperabilidad entre diferentes sistemas y complicaba el desarrollo de software y hardware compatibles.

Ante esta situación, surgió la necesidad de establecer un formato claro y conciso que garantizara su utilidad para todos los desarrolladores, permitiendo una implementación coherente y consistente en una amplia gama de plataformas computacionales. La adopción de un estándar unificado en la representación de números en coma flotante no solo simplificó el proceso de desarrollo y facilitó la interoperabilidad entre sistemas, sino que también sentó las bases para el avance y la estandarización en el campo de la computación numérica y científica.

[1] `https://es.wikipedia.org/wiki/PDP-10`
[2] `https://en.wikipedia.org/wiki/IBM_System/360`

2.2 El Estándar IEEE 754

IEEE, o Instituto de Ingenieros Eléctricos y Electrónicos (*Institute of Electrical and Electronics Engineers*)[3], es una organización profesional internacional dedicada al avance de la tecnología en diversas áreas, incluyendo la ingeniería eléctrica, electrónica, informática, telecomunicaciones, y muchas otras disciplinas relacionadas. Este instituto es conocido principalmente por desarrollar estándares técnicos, incluyendo el famoso estándar IEEE 754 para la representación de números de coma flotante en sistemas informáticos.

IEEE 754[4] es un estándar internacional que define el formato de representación de números de coma flotante en sistemas informáticos. Este estándar especifica la representación binaria de números de coma flotante, así como las reglas para realizar operaciones aritméticas con estos números, incluyendo sumas, restas, multiplicaciones y divisiones. Además, establece normas para el redondeo de resultados y el manejo de casos especiales como el cero, infinito y NaN (*Not a Number*).

Dentro de este estándar existen un gran número de formatos de representación de valores. En nuestro caso, el formato principal elegido para la unidad de coma flotante es el de simple precisión de 32 bits (*binary32*):

- 1 bit para el signo (S)
- 8 bits para el exponente (E)
- 24 bits para la Mantisa (M) (23 bits almacenados explícitamente)

Como podemos ver en la figura 2.3, el bit de signo determina el signo del número, "0" para un número positivo y "1" para un número negativo. La mantisa real que pertenece al formato incluye 23 bits que se encuentran a la derecha de la coma binaria y un bit implícito (a la izquierda de la coma). Este bit tendrá el valor de "1" con la única excepción de que todos los bits del exponente tengan el valor "0", en ese caso el bit implícito pasará a valer cero y estaremos hablando de un número denormalizado.

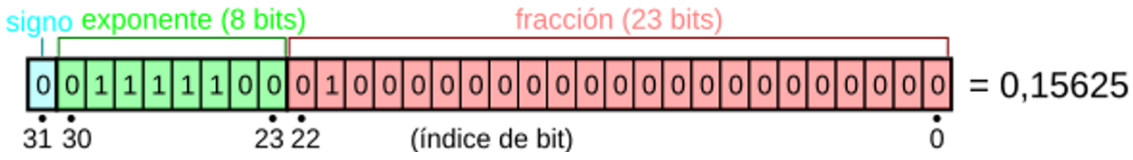

Figura 2.3: Formato de coma flotante 32 bits.

Según el estándar IEEE 754, se requiere que el resultado de una operación en coma flotante sea equivalente al que se obtendría con un cálculo de precisión absoluta y un redondeo adecuado. Con ese objetivo se definen 4 reglas de redondeo que se muestran en la tabla 2.1:

[3] `https://www.ieee.org/`

- **Redondeo al más cercano** (en caso de empate, al número par). Este es el método de redondeo por defecto del estándar. Redondea al número de coma flotante más cercano a x.

- **Redondeo a +∞**. Redondea al número de coma flotante (posiblemente +∞) mayor o igual a x.

- **Redondeo a -∞**. Redondeo al número de coma flotante (posiblemente -∞) menor o igual a x.

- **Truncado**. Redondeo al número de coma flotante más cercano a x cuya magnitud no sea más grande (es equivalente a redondear a -∞ si $x \geq 0$ o a +∞ si $x \leq 0$).

Modo de Redondeo	**Valores de ejemplo**			
	+11.5	**+12.5**	**-11.5**	**-12.5**
Al más cercano, empate al par	+12.0	+12.0	-12.0	-12.0
Al + ∞	+12.0	+13.0	-11.0	-12.0
Al - ∞	+11.0	+12.0	-12.0	-13.0
Truncado	+11.0	+12.0	-11.0	-12.0

Tabla 2.1: Ejemplos de redondeo en decimal.

Para comprender los casos especiales definidos por el estándar, es obligatorio referirse a la figura 2.2. Aquí, se evidencia que los valores que sobrepasan los límites tanto a la izquierda como a la derecha de la línea de valores son clasificados como infinito. Por otro lado, aquellos que se acercan significativamente a cero son designados como números denormalizados. Por último, cualquier valor que no se ajuste al formato establecido será considerado como NaN (*Not a Number*).

2.3 Formatos para Aprendizaje Automático

Los avances recientes en inteligencia artificial y aprendizaje automático han generado una demanda creciente por formatos de datos que cumplan mejor con los requisitos de rendimiento y precisión de estos sistemas. En respuesta a esta necesidad, los desarrolladores han creado nuevos formatos de coma flotante, como TF32 y Bfloat16, diseñados para optimizar tanto la eficiencia computacional como la precisión de cálculo en entornos de aprendizaje automático de alto rendimiento.

Estos nuevos formatos han demostrado ser especialmente importantes en el contexto de las GPU y TPU, donde la velocidad y la eficiencia en el procesamiento de datos son cruciales. Al reducir el número de bits en el formato, se logra mantener una precisión adecuada mientras se incrementa significativamente la velocidad de procesamiento.

2.3.1. Tensor Float 32

El formato de coma flotante Tensor Float 32 (TF32) de NVIDIA[4] es una innovación en la arquitectura Ampere[5], diseñada para mejorar el rendimiento en tareas de *deep learning* y supercomputación. Como podemos ver en la figura 2.4, TF32 utiliza un formato de 19 bits que incluye 1 bit de signo, 8 bits de exponente (idénticos a los del formato FP32) y 10 bits de mantisa, combinando el rango dinámico de FP32 con la precisión de la mantisa del formato FP16[6], que pertenece al estándar IEEE 754. Esto permite mantener un alto rango dinámico, evitando errores de *overflow* y *underflow* en los cálculos, mientras se mejora significativamente el rendimiento computacional en las unidades de procesamiento tensorial.

Figura 2.4: Formato TF32.

La compatibilidad de TF32 con FP32 facilita su integración en flujos de trabajo existentes sin necesidad de modificar el código. Durante las operaciones, las entradas en FP32 se redondean en sus mantisas antes de las multiplicaciones y acumulaciones, que se realizan con la precisión de TF32. Alcanzando un rendimiento mejorado respecto a las operaciones con FP32 puro. Este aumento significativo en el rendimiento se traduce en entrenamientos de modelos de deep learning mucho más rápidos y eficientes [9].

Además de su compatibilidad y eficiencia, TF32 es particularmente ventajoso en términos de uso de recursos de hardware. Al reducir la cantidad de bits necesarios para las operaciones de precisión, TF32 permite que las unidades de procesamiento puedan manejar más operaciones en paralelo, maximizando la utilización de los recursos disponibles.

2.3.2. Bfloat 16

El formato Bfloat16 (*Brain Floating Point*) desarrollado por Google[7] ha revolucionado las tareas de aprendizaje profundo y procesamiento de inteligencia artificial debido a su eficiencia y practicidad. Una de las características más notables de Bfloat16 es su estructura. Como podemos ver en la figura 2.5 conserva 1 bit de signo, 8 bits de exponente y 7 bits de mantisa. Esta configuración permite a Bfloat16 mantener el mismo rango dinámico que FP32, lo que es crucial para evitar problemas de *overflow* y *underflow* que pueden afectar negativamente al entrenamiento de modelos.

[4] https://www.nvidia.com/es-es/
[5] https://nvidia.com/es-es/data-center/ampere-architecture/
[6] https://en.wikipedia.org/wiki/Half-precision_floating-point_format
[7] https://ai.google/

Figura 2.5: Formato Bfloat16.

Otra ventaja significativa del Bfloat16 es su compatibilidad con hardware y software existentes. Esto facilita la implementación de Bfloat16 en sistemas ya establecidos sin requerir cambios drásticos en el código o la arquitectura. Los cálculos pueden realizarse en Bfloat16 y luego convertirse a FP32 sin un coste elevado al compartir el mismo tamaño de exponente, ahorrando tiempo de cálculo al trabajar con menos bits. Este equilibrio entre rendimiento y precisión ha hecho que Bfloat16 sea ampliamente adoptado en la industria [1].

Además, Bfloat16 mejora notablemente la eficiencia energética y la velocidad de entrenamiento de los modelos de aprendizaje profundo. Al reducir la cantidad de bits necesarios para representar los números, se disminuye el consumo de memoria y se acelera el procesamiento en unidades de procesamiento tensorial (TPU) y GPU. Esto permite realizar cálculos más rápidos y eficientes, lo cual es esencial para entrenar modelos complejos en menos tiempo y con menor costo energético.

2.4 Procesamiento Vectorial

El procesamiento vectorial representa una innovación esencial para potenciar el rendimiento y la eficiencia en diversas aplicaciones de computación de alto rendimiento. Esta técnica se basa en la capacidad de realizar operaciones en paralelo en conjuntos de datos utilizando vectores, lo que conlleva múltiples beneficios significativos[3].

En contraste con el procesamiento escalar, donde cada instrucción opera en un solo dato a la vez, el procesamiento vectorial aprovecha la capacidad de ejecutar una sola instrucción en múltiples elementos de datos simultáneamente, conocidos como vectores. Esto permite realizar cálculos complejos de manera más eficiente y rápida, ya que cada operación se aplica a un conjunto de datos en lugar de a datos individuales como vemos en la figura 2.6.

Una de las principales ventajas del procesamiento vectorial es su capacidad para acelerar significativamente el rendimiento de aplicaciones que implican operaciones intensivas en datos. Al realizar múltiples operaciones en paralelo, se reduce el tiempo necesario para completar la tarea, lo que se traduce en mejoras de rendimiento. Por ejemplo, en el procesamiento de imágenes o en simulaciones científicas, donde se realizan operaciones repetitivas en grandes conjuntos de datos, el procesamiento vectorial puede proporcionar grandes mejoras de rendimiento.

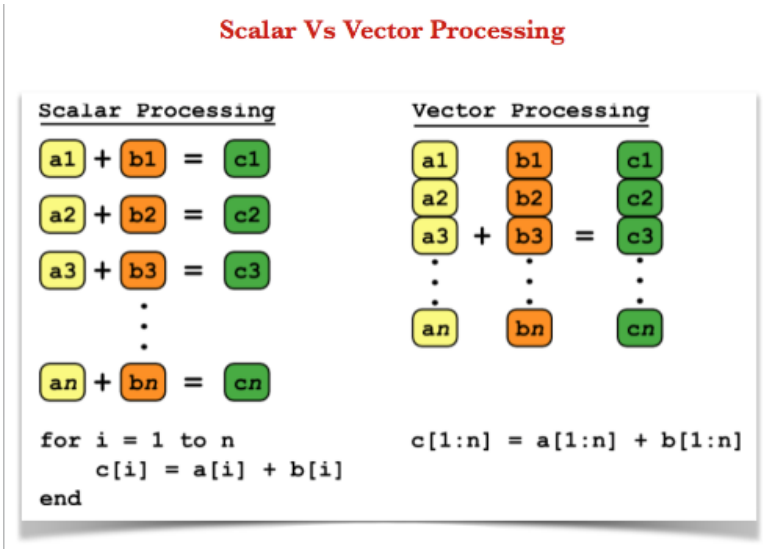

Figura 2.6: Diferencia entre procesamiento escalar y vectorial.

Otra ventaja clave del procesamiento vectorial es su capacidad para potenciar el rendimiento en aplicaciones especializadas, como el aprendizaje automático y la inteligencia artificial. Estas áreas suelen requerir operaciones intensivas en datos, y el procesamiento vectorial permite manejar eficazmente grandes volúmenes de datos en paralelo, lo que resulta en mejoras notables en el rendimiento y la capacidad de procesamiento.

La comprensión detallada del uso del enmascaramiento en los operadores y la aplicación de técnicas de segmentación es esencial para una implementación eficaz del procesamiento vectorial.

En los siguientes apartados, exploraremos en detalle cómo estas técnicas contribuyen a optimizar el rendimiento y la eficiencia del sistema al ejecutar operaciones en conjuntos de datos.

2.4.1. Enmascaramiento en los Operadores

Las operaciones con máscaras son una técnica fundamental en el procesamiento de datos que permite seleccionar, manipular o condicionar elementos específicos dentro de un vector. Una máscara es una secuencia de bits que actúa como un filtro, donde cada bit determina si se debe aplicar una operación en el elemento correspondiente del vector.

Cuando se aplican máscaras a una operación aritmética, el resultado es que la operación se realiza sobre los elementos seleccionados por la máscara y los elementos no seleccionados son aquellos que no se modifican, manteniendo su valor.

Estas operaciones con máscara toman una gran importancia en el contexto del procesamiento vectorial, ya que estas se utilizan para realizar operaciones condicionales en paralelo en un conjuntos de datos. Esto permite, por ejemplo, aplicar una operación solo a los elementos que cumplen cierta condición, o seleccionar y

modificar un subconjunto específico de datos según lo especificado por la máscara.

A continuación podemos ver la equivalencia en C a las operaciones con máscaras en el procesamiento vectorial. En este, podemos observar como estamos realizando una suma de dos vectores y solo almacenamos el valor de la suma en el vector en el caso de que el elemento de A sea mayor que 0.

```c
#include <stdio.h>

#define SIZE 10

int main() {
    int A[SIZE] = {1, -2, 3, 4, -5, 6, 7, -8, 9, 10}; // Vector A
    int B[SIZE] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};   // Vector B
    int C[SIZE]; // Vector resultante

    for (int i = 0; i < SIZE; i++) {
        if (A[i] > 0) {
            C[i] = A[i] + B[i];
        } else {
            C[i] = 0;
        }
    }
        return C;
}
```

Las operaciones con máscaras funcionarían de una manera análoga, los valores de A que sean mayores que 0 serían los selecionados por la máscara para ser usados en la suma, y aquellos que no sean seleccionados serían los valores de A menores o iguales que 0.

2.4.2. Segmentación de Operadores

La segmentación es una técnica fundamental para mejorar el rendimiento y la eficiencia de los operadores sin aumentar significativamente la complejidad del diseño. Esta descompone operaciones complejas en etapas más pequeñas y manejables.

El objetivo principal de la segmentación es dividir las operaciones aritméticas y lógicas en una serie de etapas secuenciales, donde cada etapa realiza una parte específica de la operación. Estas etapas pueden ejecutarse de forma concurrente y en paralelo, lo que resulta en una mejora significativa en la velocidad de procesamiento.

La segmentación en los operadores permite optimizar el rendimiento de los procesadores modernos. Ofrece la capacidad de aprovechar al máximo los recursos de hardware disponibles y mejorar la eficiencia del procesador en general.

Un ejemplo ilustrativo sobre la técnica de segmentación se puede ver en la figura 2.7. Aquí, se representan dos instrucciones(una en azul y otra en rojo) que requieren de tres ciclos de reloj para su ejecución completa. Al implementar la técnica de segmentación, estas instrucciones se introducen en el *pipeline*, un canal de procesamiento que divide la ejecución de las instrucciones en tres etapas.

Cada etapa del *pipeline* se resuelve en un solo ciclo de reloj. Lo importante es que diferentes etapas de las instrucciones se ejecuten en paralelo. Este enfoque resulta en una mejora sustancial del rendimiento, ya que se logra una mayor eficiencia al aumentar la frecuencia a la que se ejecutan las instrucciones y aprovechar al máximo los recursos de la CPU.

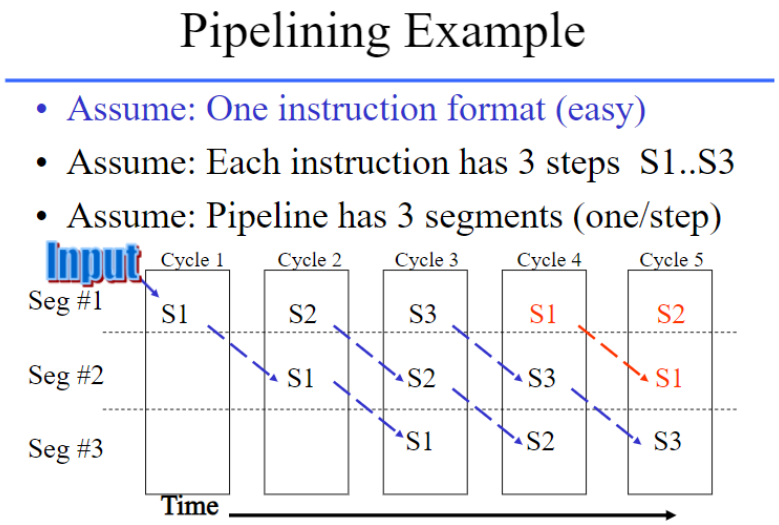

Figura 2.7: Segmentación de operadores

CAPÍTULO 3
Análisis del Problema

En este capítulo, presentaremos las herramientas de desarrollo empleadas para implementar la solución y analizaremos diversas opciones de diseño para construir los distintos módulos que componen la unidad de coma flotante. Comentaremos las ventajas y desventajas de cada opción, decantándonos por la variante que nos ofrezca el mejor rendimiento y se adapte a los objetivos a cumplir.

El desarrollo de una unidad de cálculo vectorial de coma flotante presenta varios desafíos técnicos y de diseño. Es fundamental, en primer lugar, realizar una distinción clara entre diseños secuenciales y combinacionales, ya que estas son las dos principales estrategias disponibles.

Los diseños secuenciales, por un lado, implican la ejecución de operaciones de manera ordenada y en etapas sucesivas, utilizando el hardware disponible de forma eficiente. Sus implementaciones son normalmente más sencillas de desarrollar pero tienen como desventaja la mayor dificultad para poder aplicar correctamente la técnica de segmentación.

Por otro lado, los diseños combinacionales permiten realizar todas las operaciones necesarias en un solo paso, lo que facilita implementar la técnica de segmentación y ofrece mejoras en términos de rendimiento y latencia, pero pueden requerir más recursos de hardware y una mayor optimización para evitar retardos excesivos.

Analizando las ventajas de cada tipo de diseño, nos hemos decantando por la solución combinacional ya que nuestro objetivo es proporcionar soporte al procesamiento vectorial, lo que requiere una arquitectura segmentada y con la capacidad de soportar operaciones con máscaras.

La segmentación es esencial para permitir que cada ciclo de reloj realice una operación en diferentes componentes del vector, mejorando significativamente el rendimiento en aplicaciones paralelas.

Además, las operaciones con máscaras permiten un control preciso sobre qué elementos del vector se procesan y sobre cuales no, proporcionando soporte a operaciones condicionales sobre un gran conjunto de datos (vector) de una manera eficiente como mencionamos en el capítulo 2

3.1 Elección de Herramientas de Desarrollo

3.1.1. Verilog

El lenguaje de descripción de hardware Verilog[5] ha sido seleccionado para el desarrollo de nuestra unidad de cálculo vectorial debido a su prevalencia y adopción dentro de nuestro grupo de investigación. Al utilizar Verilog, garantizamos una cohesión y compatibilidad óptimas con los proyectos ya existentes, facilitando así la integración y el mantenimiento del código.

Desarrollado en la década de 1980, Verilog es uno de los lenguajes de descripción de hardware más populares en la industria de diseño de sistemas digitales. Su sintaxis y estructura, similares a las de los lenguajes de programación convencionales, facilitan su aprendizaje y uso para aquellos con experiencia previa en programación.

Una de las principales ventajas de Verilog es su capacidad para modelar y simular sistemas complejos con alta precisión. Permite la descripción detallada de cada componente y su interconexión como se muestra en la figura 3.1. Además, su capacidad para realizar simulaciones a nivel de comportamiento, transferencia de registros y puertas lógicas permite a los diseñadores verificar y optimizar sus diseños antes de la implementación física.

Verilog también destaca por su capacidad de soporte para la síntesis automatizada. Los diseños descritos en este lenguaje de descripción de hardware pueden ser directamente convertidos en circuitos físicos mediante herramientas de síntesis, como Xilinx Vivado[1] . Esta integración entre la descripción de alto nivel y la implementación física agiliza el proceso de desarrollo y reduce los errores. Adicionalmente, Verilog facilita la realización de pruebas automatizadas mediante *testbench*, permitiendo validar el comportamiento del diseño en diversas condiciones y escenarios.

Finalmente, la amplia aceptación y soporte de Verilog en la industria ofrece numerosos recursos, bibliotecas y herramientas compatibles que facilitan el diseño y la verificación de sistemas complejos. La activa comunidad de usuarios y desarrolladores garantiza un flujo constante de mejoras y actualizaciones, así como un gran repositorio de conocimientos y buenas prácticas accesibles para cualquier ingeniero.

[1] `https://www.xilinx.com/products/design-tools/vivado.html`

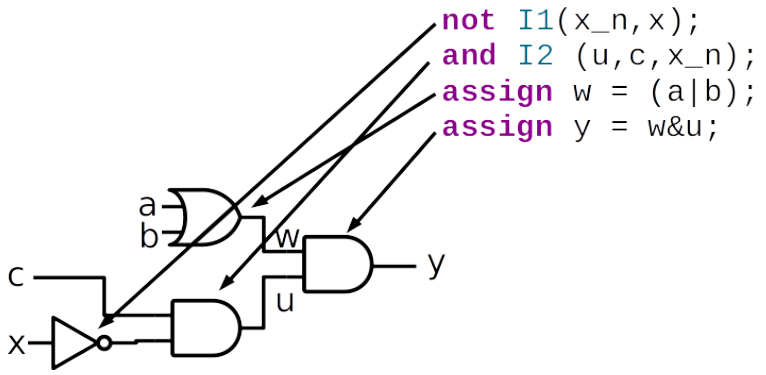

Figura 3.1: Equivalencia entre Verilog y hardware.

3.1.2. Xilinx Vivado

Xilinx Vivado es una herramienta integral para el diseño y síntesis de hardware, muy utilizada en la industria para el desarrollo de sistemas digitales avanzados. Creada por Xilinx(comprada por AMD)[2], ofrece un entorno completo que abarca desde la concepción del diseño hasta su implementación y verificación final en FPGA. Su capacidad para gestionar todo el flujo de diseño la convierte en una herramienta de gran utilidad.

Una de las principales ventajas de Xilinx Vivado es su potente motor de síntesis y optimización. Vivado permite a los diseñadores convertir descripciones de alto nivel, como las escritas en Verilog, en configuraciones físicas optimizadas para FPGA. Este proceso automatizado no solo acelera el ciclo de desarrollo, sino que también asegura que los diseños sean altamente eficientes en términos de recursos y rendimiento. La capacidad de Vivado para optimizar automáticamente el uso de LUT (*Look-Up Tables*), *flip-flops* y otros recursos críticos de la FPGA ofrece ventajas para desarrollar hardware que maximice el rendimiento sin exceder las limitaciones de espacio.

A su vez, Xilinx Vivado proporciona herramientas avanzadas de simulación y verificación que permiten a los ingenieros validar sus diseños antes de la implementación física. Vivado incluye un simulador integrado que soporta simulaciones a nivel de comportamiento, RTL (*Register Transfer Level*) y puertas lógicas, permitiendo una verificación exhaustiva en cada etapa del diseño. Estas capacidades de simulación son cruciales para detectar y corregir errores en la fase inicial del desarrollo, lo que reduce el tiempo y los costos asociados con el ciclo de diseño iterativo.

Otra característica destacada de Xilinx Vivado es su entorno de desarrollo integrado (IDE) intuitivo y fácil de usar. El IDE de Vivado proporciona un conjunto completo de herramientas y características que facilitan la gestión de proyectos de diseño, la visualización de resultados de síntesis y la depuración de problemas. La interfaz gráfica de usuario permite a los ingenieros navegar fácilmente por los distintos módulos del diseño, realizar análisis de temporización y ajustar parámetros para optimizar el rendimiento del hardware. Esta facilidad de uso es

[2] `https://www.amd.com/en.html`

particularmente beneficiosa en proyectos complejos, donde la gestión eficiente de múltiples componentes y conexiones es crítica.

Xilinx Vivado también soporta un amplio rango de FPGA y dispositivos programables de Xilinx, lo que proporciona a los diseñadores la flexibilidad necesaria para elegir la plataforma de hardware que mejor se adapte a sus necesidades específicas. La compatibilidad con dispositivos de última generación asegura que los diseños puedan aprovechar las características más avanzadas y los mayores niveles de integración disponibles en el mercado.

3.2 Análisis y Comparación de Implementaciones Hardware

En esta sección, realizaremos un análisis detallado de las diversas implementaciones hardware consideradas para la unidad de coma flotante. Exploraremos las diferentes opciones disponibles, evaluando sus características, ventajas y desventajas. Además, compararemos estas implementaciones entre sí para identificar cuál se ajusta mejor a los objetivos y requisitos del proyecto. Al final de este análisis, se tomará una decisión justificada sobre la implementación a seguir, con el objetivo de optimizar el rendimiento y la eficiencia de la unidad de coma flotante.

En este apartado, es crucial diferenciar claramente entre tres formas de diseño completamente distintas: el uso de IP cores de Xilinx, los operadores nativos del entorno Vivado, y nuestras propias implementaciones modulares.

Por un lado, tenemos los operadores nativos del entorno de Vivado. A pesar de su facilidad de uso, estos operadores no son adecuados para números en coma flotante, por lo que los descartamos de nuestra comparación de inmediato. Estos operadores están diseñados para operaciones de números enteros y no pueden manejar las complejidades de la aritmética en coma flotante. La aritmética en coma flotante requiere un manejo especializado de los valores del signo, exponente y mantisa, lo cual no es soportado por los operadores básicos proporcionados por Vivado.

Por otro lado, tenemos la posibilidad de utilizar IP cores de Xilinx como el mostrado en la figura 3.2 para realizar las operaciones de coma flotante soportadas por la unidad vectorial. Estos IP cores están altamente optimizados por Xilinx y ofrecen latencias mínimas y un muy buen rendimiento. Sin embargo, esta elección limita nuestra flexibilidad y portabilidad, ya que nos vincula a la plataforma FPGA de Xilinx.

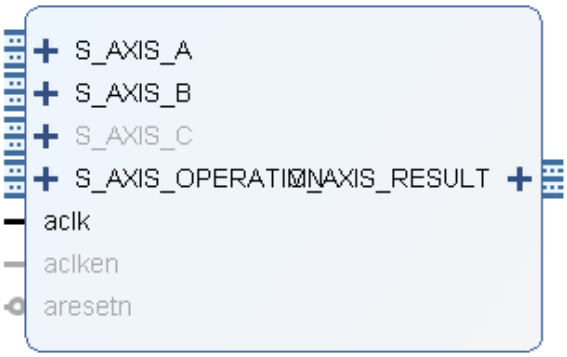

Figura 3.2: IP core de suma/resta de coma flotante.

Una de nuestras metas principales es desarrollar una solución que sea independiente de la tecnología FPGA específica, permitiendo la exportación tanto a ASIC como a FPGA. Por lo tanto, a pesar de su rendimiento superior, el IP core de Xilinx no cumple con este objetivo crucial de nuestro proyecto. La dependencia de una solución específica de un proveedor limita nuestra capacidad para migrar nuestro diseño a otras plataformas, afectando negativamente la versatilidad y adaptabilidad de nuestro proyecto.

Finalmente, consideramos una implementación modular y totalmente personalizada. Esta opción nos permite tener un control total sobre el diseño y su funcionamiento, asegurando que se ajuste a nuestros requerimientos específicos. Al desarrollar un diseño propio, podemos ajustar cada operador para satisfacer nuestras necesidades de rendimiento y precisión. Aunque esta opción puede no ofrecer las bajas latencias del IP core de Xilinx, nos proporciona la flexibilidad que buscamos. Podemos implementar y probar nuestro diseño en diferentes plataformas, asegurando que se mantenga funcional y eficiente en diversos entornos. Además, al entender completamente el funcionamiento interno de nuestra implementación, podemos realizar ajustes y mejoras continuas para mejorar su rendimiento y adaptabilidad.

Por las razones justificadas anteriormente, elegimos nuestra propia implementación modular de los operadores de coma flotante porque nos permite mantener la independencia tecnológica, comprender profundamente el funcionamiento interno del cálculo y adaptar la solución según los requerimientos específicos del proyecto. Esta decisión asegura que nuestro diseño sea flexible y que esté alineado con nuestros objetivos.

Una vez definida nuestra elección y sus motivos, procedemos a discutir las opciones que tenemos para desarrollar cada tipo de operador:

3.2.1. Sumador/Restador

La operación de suma y resta de números es esencial en una unidad vectorial de coma flotante. Estas operaciones son más complejas que sus contrapartes en aritmética entera debido al hecho de que constan de muchas más etapas como extraer el signo, exponente y mantisa del operando, calcular el signo final, alinear las mantisas de los dos operandos, sumarlas, normalizar el resultado, redondearlo y volverlo a normalizar en caso de que sea necesario.

Entre todas esas etapas, en este apartado centraremos nuestra atención en la parte más costosa y que tiene más variedad de implementaciones posibles, la suma de mantisas. A continuación realizaremos una comparación entre dos tipos de sumadores que podrían ser utilizados en la implementación de esta operación: el *Carry Propagate Adder* (CPA) y el *Carry Lookahead Adder* (CLA).

El CPA es una de las implementaciones más sencillas y clásicas para sumar números binarios. Su funcionamiento se basa en la propagación del acarreo a través de cada bit del sumador de manera secuencial. Internamente, cada bit del sumador calcula su suma utilizando un conjunto de *full adders* y genera un acarreo que se propaga al siguiente bit. Este proceso continúa hasta el bit más significativo y se puede ver claramente en la figura 3.3. El tiempo de retardo en un CPA es linealmente proporcional al número de bits, ya que cada bit debe esperar a que el acarreo de los bits anteriores sea calculado.

La simplicidad del CPA es una de sus mayores ventajas. Es fácil de entender e implementar, lo que lo hace adecuado para aplicaciones sencillas o donde el retardo no es crítico. Además, en diseños con un número limitado de bits, el CPA puede ser más eficiente en términos de recursos. Sin embargo, su principal inconveniente es el retardo acumulativo, que aumenta linealmente con el número de bits, haciéndolo ineficiente para operaciones con un gran número de bits, como las mantisas en coma flotante.

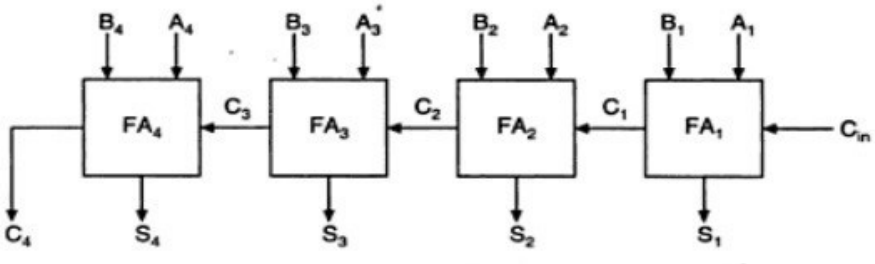

Figura 3.3: CPA con operadores de 4 bits.

Por otro lado, el CLA es una implementación más avanzada diseñada para superar las limitaciones del CPA mediante el cálculo anticipado de los acarreos. A diferencia del CPA, este sumador utiliza una lógica adicional para predecir si se generará un acarreo en cada bit, lo que permite calcular los acarreos de manera anticipada. La lógica de generación y propagación de acarreos en el CLA utiliza dos tipos de señales: *Generate* (G), que indica que un acarreo será generado por un bit específico, y *Propagate* (P), que indica que un acarreo será propagado a través de un bit específico.

Usando estas señales, el CLA puede calcular la suma de cada bit y el acarreo final sin esperar a que se propaguen los acarreos de los bits anteriores, lo que reduce significativamente el tiempo usado para producir un resultado. Este sumador tiene un retardo que es mucho menos dependiente del número de bits, ya que calcula los acarreos en paralelo, haciéndolo ideal para operaciones con un número elevado de bits, como las mantisas en coma flotante. Aunque el CLA utiliza más lógica para calcular los acarreos, esta inversión en recursos adicionales se traduce en una reducción significativa del tiempo de cómputo.

Relacionado con el diseño de un CLA, existen diversas variedades de implementación según la jerarquía de los generadores de acarreo:

- **CLA de 1 nivel:** Un CLA de 1 nivel se caracteriza por su arquitectura simple, sin ninguna clase de jerarquía. En esta configuración, existe una única etapa encargada de calcular simultáneamente todas las señales de Generar (G) y Propagar (P) para todos los sumadores completos (*fulladders*).

  En la figura 3.4 se puede observar la estructura interna de un CLA de un nivel, el cual produce un resultado de 4 bits. En esta estructura, los acarreos generados por los sumadores completos se calculan en paralelo mejorando así significativamente el rendimiento.

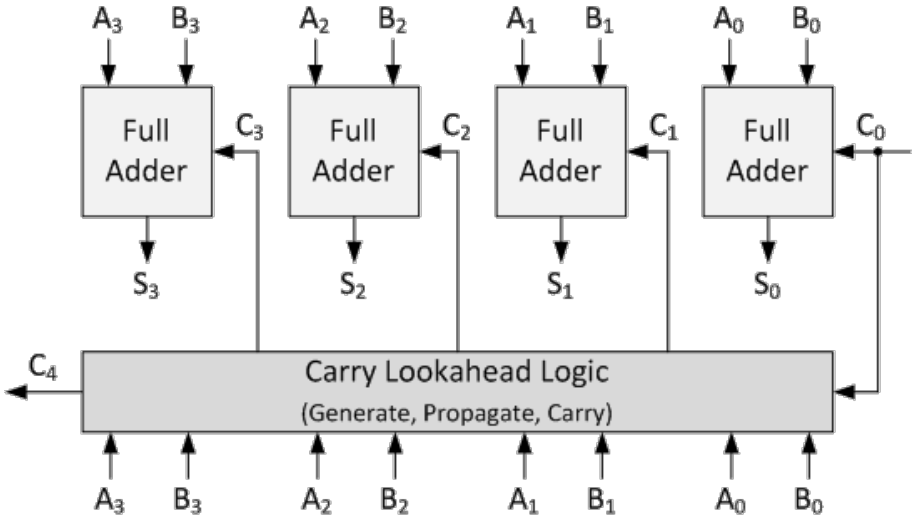

Figura 3.4: CLA de 1 nivel.

- **CLA de 2 niveles:** La causa del nombre es porque divide el problema de la propagación del acarreo en dos etapas jerárquicas como se puede ver en la figura 3.5. En la primera etapa, el sumador se divide en bloques más pequeños, por ejemplo de 4 bits. Cada uno de estos bloques calcula sus propias señales de G y P. La señal de generación indica si un bloque generará un acarreo independientemente de la entrada de acarreo, mientras que la señal de propagación indica si un bloque propagará un acarreo desde su entrada a su salida.

  En la segunda etapa, varios bloques de 4 bits se agrupan para formar bloques superiores más grandes, como por ejemplo de 16 bits. Los bloques superiores combinan las señales G y P de los bloques de 4 bits para calcular las señales G y P del bloque de 16 bits. Esta combinación se realiza mediante lógica adicional que genera el acarreo de salida para cada bloque superior. En esencia, los bloques de 4 bits calculan las señales de generación y propagación y luego estos resultados se combinan a nivel superior para determinar los acarreos finales de manera más rápida.

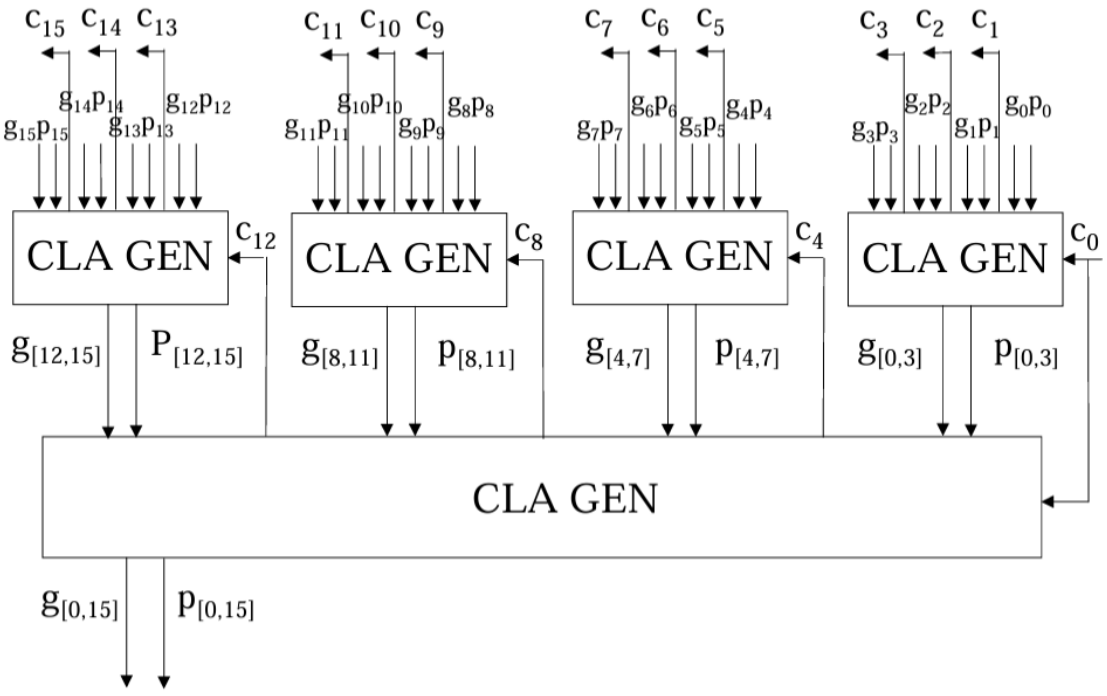

Figura 3.5: Generador de acarreo de 2 niveles.

El CLA de 2 niveles presenta varias ventajas significativas. La más notable es la reducción del ratio *fan-in*, *fan-out*, siendo estos el número máximo de señales de entrada y salida respectivamente, que un componente lógico puede manjear sin una pérdida de rendimiento. Esto conlleva que, a pesar de contar con un tiempo de propagación mayor, resulte en un retardo menor en comparación a un CLA de 1 nivel.

Otra ventaja importante es la escalabilidad. El diseño jerárquico del CLA de 2 niveles permite que sea fácilmente adaptado para sumar números de mayor tamaño aumentando el número o el tamaño de los bloques básicos. Esto hace que el CLA de 2 niveles sea una elección adecuada para aplicaciones que requieren alta velocidad de procesamiento, como las operaciones en unidades de coma flotante en arquitecturas vectoriales.

Sin embargo, una desventaja del CLA de 2 niveles frente al CLA de 1 nivel es la mayor complejidad del diseño. Debido a la jerarquía adicional y las etapas intermedias de cálculo, el CLA de 2 niveles requiere más recursos de hardware y una lógica de control más elaborada. Esto puede traducirse en un aumento del área necesaria para su implementación, lo que puede no ser justificado en aplicaciones donde el ahorro de recursos y la simplicidad del diseño son prioritarios.

- **CLA de 3 niveles:** El CLA de 3 niveles es una extensión del diseño del CLA de 2 niveles, introduciendo una tercera capa jerárquica en la estructura de cálculo de acarreo como podemos ver en la figura 3.6. En este diseño, los sumadores se agrupan en bloques más pequeños, y estos bloques a su vez se agrupan en bloques más grandes, creando así una jerarquía de tres niveles. Cada nivel de la jerarquía se encarga de pre-calcular los acarreos para su nivel y pasar esta información al siguiente nivel superior.

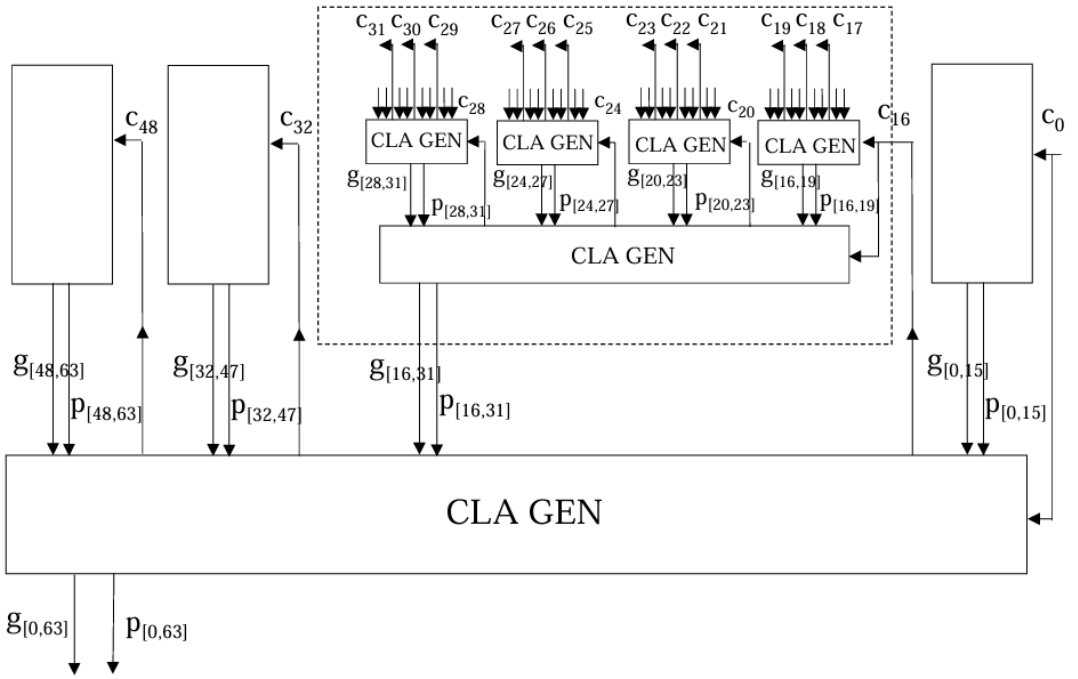

Figura 3.6: Generador de acarreo de 3 niveles.

La principal ventaja del CLA de 3 niveles respecto a los de 1 y 2 niveles es la reducción del ratio de *fan-in* y *fan-out*. Aunque las características del diseño incrementen el tiempo de propagación, esto conlleva a tener un menor retardo total del sumador. Además, el diseño con más etapas jerárquicas

mejora significativamente su escalabilidad, permitiendo sumar números de mayor tamaño de manera más eficiente.

Sin embargo, el CLA de 3 niveles también tiene desventajas en comparación con los CLA de 1 y 2 niveles. La principal desventaja es que incrementa la complejidad del diseño. Con una jerarquía más profunda, la lógica de control se vuelve más compleja y requiere más recursos de hardware. Esta complejidad adicional puede resultar en un aumento en el área necesaria para su implementación. Además, la implementación de un CLA de 3 niveles puede ser más difícil y costosa en términos de diseño y verificación, lo que podría no ser justificable en aplicaciones donde la simplicidad y la eficiencia de recursos son más importantes que la velocidad absoluta.

Cada diseño presenta una serie de ventajas y desventajas, lo que dificulta la selección de una opción basada únicamente en el marco teórico. Para encontrar la solución más óptima, es necesario realizar pruebas empíricas. Por ello, hemos llevado a cabo un caso de estudio comparando las diferentes opciones de diseño en el contexto específico de los operadores de coma flotante.

En este estudio, examinaremos detalladamente los retardos, tanto de lógica como de enrutamiento y la cantidad de LUT que ocupa cada diseño. Esta evaluación nos permitirá determinar cuál opción es el más eficiente y adecuado para nuestra aplicación.

En primer lugar, realizaremos la comparación de los sumadores de números de 24 bits, que son los candidatos a ser usados en el sumador de mantisas del sumador/restador de coma flotante que se muestra en el apartado 4.2.

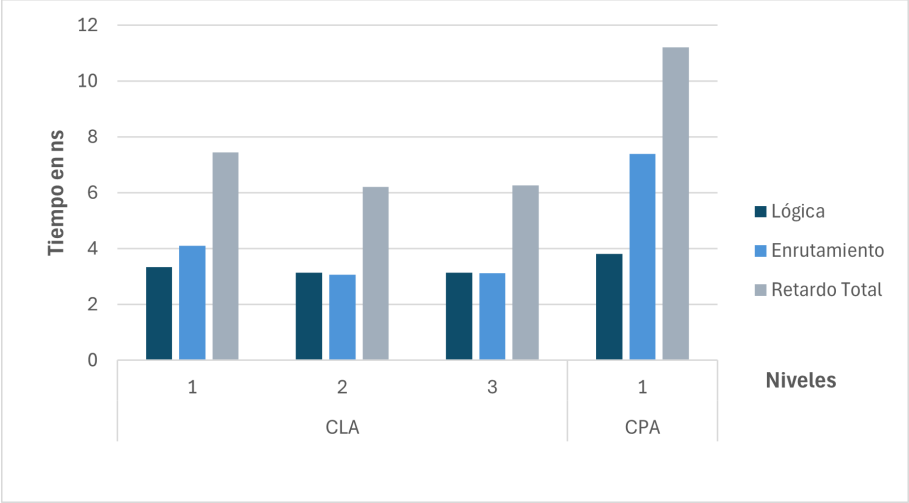

Figura 3.7: Retardos en sumadores de 24 bits.

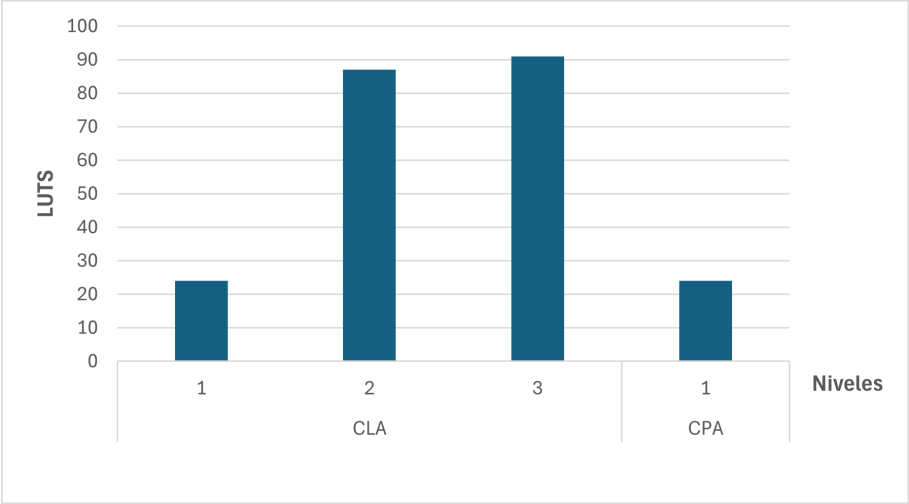

Figura 3.8: Recursos requeridos en sumadores de 24 bits.

Como hemos mencionado anteriormente, para realizar la suma de las mantisas en nuestra unidad de coma flotante, necesitamos sumadores de 24 bits. En la figura 3.7 se puede observar claramente que, en términos de retardos, el CPA es el menos eficiente al perder una gran cantidad de tiempo en el enrutamiento, resultando en tiempos de propagación significativamente mayores en comparación con los CLA de 2 y 3 niveles, que requieren de menor tiempo. Esto demuestra que los CLA de 24 bits tienen una ventaja considerable en cuanto a la velocidad de operación respecto a los CPA de 24 bits.

A su vez, la figura 3.8 revela que los sumadores que ofrecen los mejores tiempos de operación son también aquellos que requieren un mayor número de LUTS para su implementación. Este hallazgo subraya la relación directa entre la velocidad de los sumadores y los recursos de hardware necesarios.

Nuestro objetivo principal es encontrar el sumador más rápido posible para evitar que se convierta en un cuello de botella que limite las frecuencias operativas de la unidad de coma flotante. Después de evaluar los resultados, hemos decidido optar por el uso de un CLA de 2 niveles debido a su equilibrio entre velocidad y utilización de recursos. Aunque los CLA de 3 niveles ofrecen tiempos de operación ligeramente mejores, el CLA de 2 niveles utiliza menos recursos proporcionando una buena relación entre rapidez y eficiencia en el uso de hardware. Lo que lo convierte en la mejor opción para nuestra aplicación específica.

A continuación, mostraremos los datos referentes a los sumadores de 48 bits, estos son los candidatos potenciales a ser usados en el operador de multiplicación de coma flotante mostrado en el apartado 4.3.

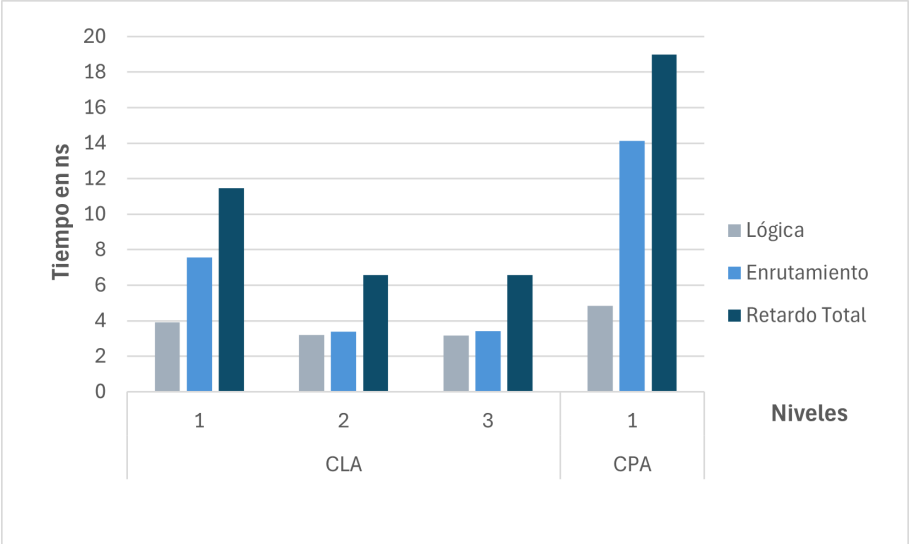

Figura 3.9: Retardos en sumadores de 48 bits.

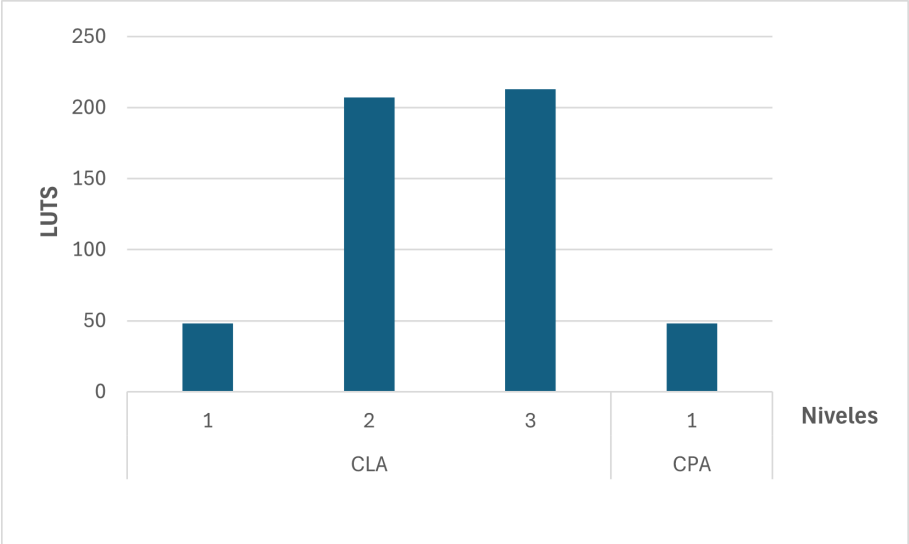

Figura 3.10: Recursos requeridos en sumadores de 48 bits.

Al analizar la figura 3.9, se puede observar un patrón similar al observado con los sumadores de 24 bits: El CPA se descarta de inmediato debido a sus retardos significativamente mayores, presentando retardos de hasta casi tres veces superiores en comparación con los CLA de 2 y 3 niveles. Esta ineficiencia se debe a la creciente pérdida de tiempo de enrutamiento debido a la estructura interna del CPA.

Por otro lado, la figura 3.10 confirma y acentúa el patrón observado anteriormente con los sumadores de menor tamaño. La diferencia en la cantidad de LUT necesarias se vuelve aún más pronunciada, reafirmando la hipótesis de que existe una relación directa entre el menor retardo y el mayor uso de recursos. Los CLA de 2 y 3 niveles, aunque requieren más LUT, ofrecen tiempos de operación significativamente mejores.

Reforzando nuestro objetivo de obtener los sumadores más rápidos y eficientes, en esta comparativa hemos optado nuevamente por el CLA de 2 niveles para nuestro sumador de 48 bits, que se integrará en el multiplicador de coma flotante. Esta elección se basa en su buena relación entre retardo y recursos, ofreciendo un rendimiento superior sin comprometer la eficiencia en el uso de hardware.

Por último, hemos decidido llevar a cabo el caso de estudio para sumadores de 64 bits. A pesar de que este tamaño de sumadores no tiene una aplicación directa a ninguno de los operadores que se utilizan para el desarrollo de la unidad de coma flotante, hemos considerado necesario realizar el estudio de estos sumadores, para así apoyar de forma empírica todas las afirmaciones que hemos ido exponiendo a lo largo del caso de estudio.

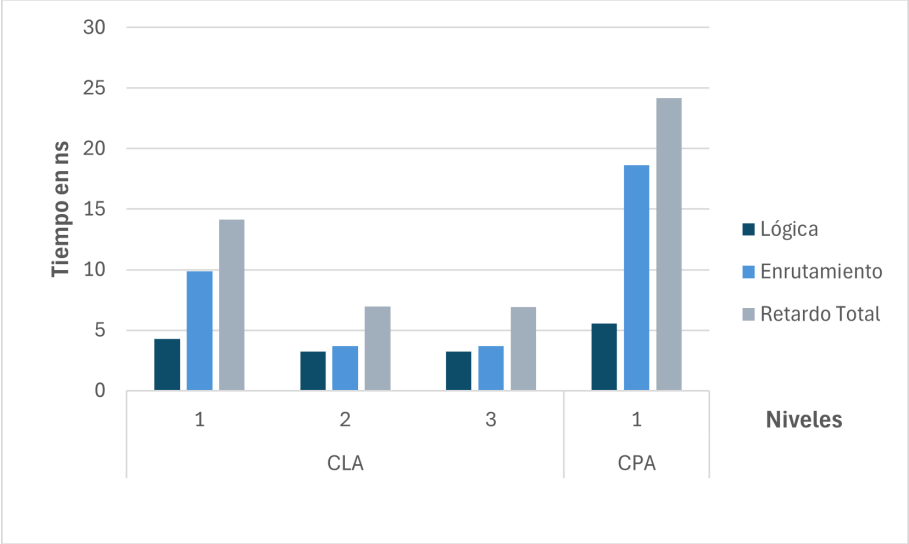

Figura 3.11: Retardos en sumadores de 64 bits.

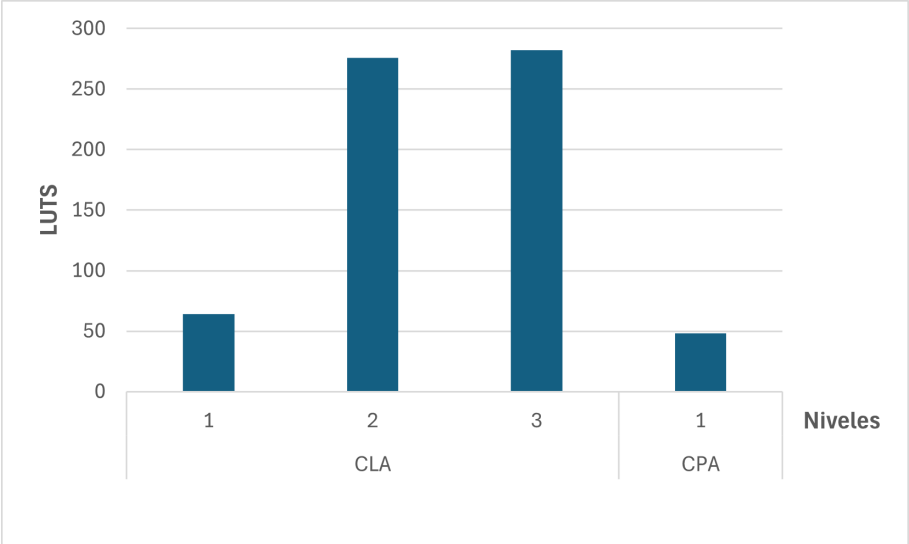

Figura 3.12: Recursos requeridos en sumadores de 64 bits.

Al analizar detenidamente las figuras 3.11 y 3.12, se revela claramente la tendencia de los sumadores conforme se incrementan los bits de los operandos. El tiempo de enrutamiento de los CPA se muestra como su principal desventaja en comparación con los CLA, dado que, en términos de lógica, los retardos no son tan distantes.

Desde una perspectiva de diseño, este análisis comparativo ha permitido destacar que los CLA de 2 y 3 niveles gozan de una ventaja indiscutible en entornos donde el rendimiento y la velocidad son el principal objetivo, gracias a su capacidad para minimizar los tiempos de enrutamiento. No obstante, esta superioridad se ve contrarrestada por el elevado consumo de recursos, lo que puede constituir un factor limitante en ciertos contextos de diseño.

En conclusión, este estudio respalda la afirmación de que en un diseño donde los recursos no sean una limitación, los CLA de 2 y 3 niveles representan la opción óptima, al proporcionar un equilibrio adecuado entre velocidad y eficiencia. Sin embargo, en escenarios donde la reducción en el uso de recursos sea crucial, podría ser necesario sacrificar algo de velocidad de cómputo a cambio de una menor demanda de recursos, optando por un CLA de 1 nivel o incluso un CPA. Esta decisión debe fundamentarse en una evaluación exhaustiva de las prioridades del diseño y los requisitos específicos del sistema en cuestión.

3.2.2. Multiplicador

La multiplicación de números en coma flotante presenta varios desafíos técnicos y de diseño, especialmente cuando se trata de cumplir con los objetivos de rendimiento y eficiencia. Para abordar estos desafíos es necesario identificar claramente las diferentes etapas que han de componer al multiplicador.

Entre estas etapas podemos encontrar el extraer el signo, exponente y mantisa de la entrada, calcular el signo final, sumar los exponentes, restar el exceso al resultado de esa suma, multiplicar las mantisas, normalizar el resultado y redondear y renormalizar el resultado en caso de que sea necesario.

En este apartado pondremos el foco en la parte más demandante tanto en recursos como en tiempo de cómputo, la multiplicación de mantisas. Para implementarla existen dos grandes variantes: las implementaciones secuenciales y las implementaciones combinacionales.

Cada una de estas categorías tiene sus propias características, ventajas y desventajas que hemos descrito anteriormente.

Un ejemplo destacable de implementaciones secuenciales para la multiplicación de mantisas es la implementación secuencial del algoritmo de Booth, que utiliza una técnica de codificación específica para reducir el número de operaciones necesarias en el proceso de multiplicación.

El algoritmo de Booth[7] (figura 3.13) es un método que permite la multiplicación binaria de números con menos operaciones aritméticas, especialmente útil para números con secuencias largas de bits iguales. La codificación de Booth agrupa los bits del multiplicador en pares o triples y determina la acción a tomar basándose en la secuencia de estos bits. Dependiendo de los bits observados, se pueden realizar operaciones de suma, resta o ninguna operación en el acumulador, seguido de un desplazamiento de los registros. Este método puede ser ventajoso en términos de ahorro de área, ya que requiere menos hardware activo en cada ciclo de reloj.

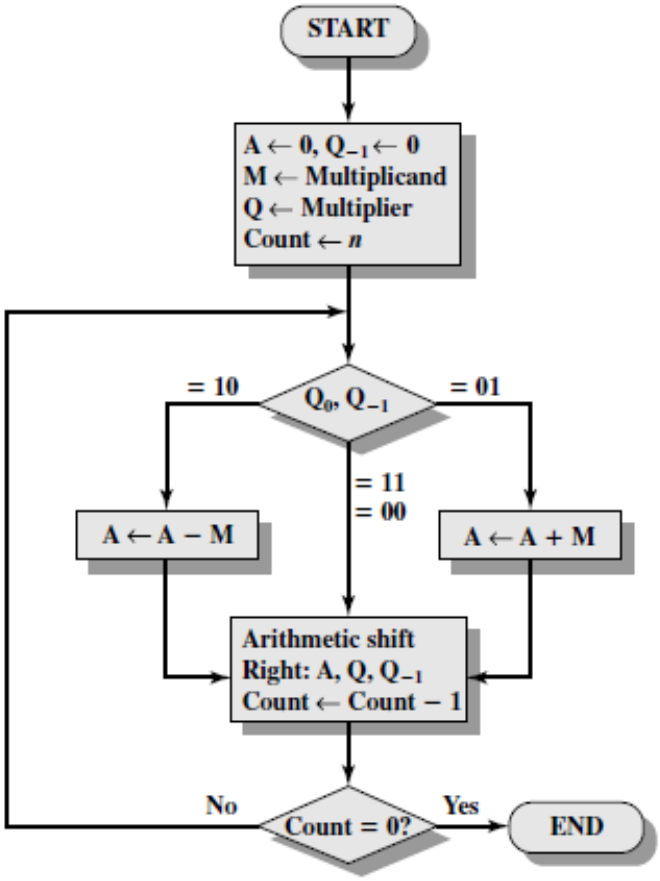

Figura 3.13: Algoritmo de booth en secuencial.

Como se ha detallado previamente, al descartar las opciones secuenciales, nos enfrentamos a la decisión de seleccionar la implementación combinacional más adecuada para las características de nuestro diseño. En este contexto, debemos elegir entre dos alternativas principales: el multiplicador en array y el generador de productos parciales acompañado de un árbol de Wallace. Cada opción tiene sus propias ventajas y consideraciones que comentaremos a continuación:

Multiplicador en Array

El multiplicador en array (figura 3.14) es una implementación combinacional que organiza los multiplicadores en una matriz bidimensional de celdas. Cada celda realiza una operación de multiplicación de bits y suma parcial, permitiendo que todas las operaciones se realicen en paralelo. Los productos parciales generados se desplazan y se alinean correctamente para la suma final. Además, las celdas propagan los acarreos necesarios durante las operaciones de suma parcial. Al final, las sumas parciales y los acarreos se combinan para producir el resultado final de la multiplicación.

Las ventajas del multiplicador en array incluyen una estructura regular y simple a base de *fulladders*, lo que facilita su implementación y diseño inicial. Esta simplicidad estructural permite una fácil escalabilidad y una comprensión clara del flujo de datos a través del circuito. Además, la uniformidad del diseño implica que el proceso de desarrollo puede ser optimizado, ya que los mismos bloques básicos se repiten a lo largo del array, reduciendo así el tiempo y el costo de desarrollo. Otra ventaja de este tipo de multiplicador es su capacidad para realizar operaciones de multiplicación de manera paralela, lo que puede resultar en una latencia relativamente baja para ciertas aplicaciones.

Su diseño también permite una fácil integración con otros componentes del sistema, gracias a su estructura modular. Esta ofrece la ventaja de que el array puede ser ajustado o ampliado para manejar diferentes tamaños de operandos sin necesidad de rediseñar completamente el circuito. Asimismo, la naturaleza paralela del array puede ser explotada para aumentar el rendimiento en tareas específicas que beneficien de la ejecución simultánea de múltiples operaciones.

Sin embargo, a pesar de estas ventajas, el multiplicador en array presenta desventajas significativas cuando se trata de tamaño y de facilidad para aplicar la técnica de segmentación. En concreto, este necesita n^2 *fulladders*, siendo n el número de bits de los operandos, que en concreto para nuestro diseño para una n=32, serían un total de 1024 *fulladders*, lo cual puede resultar en un uso considerable de espacio en el chip. Para adaptarse a un diseño segmentado, se requeriría una segmentación diagonal de los módulos del diseño, como se muestra en la figura 3.14, para que esta fuera del todo óptima, en lugar de la forma horizontal habitual, generando esto dificultades adicionales para aplicar correctamente la técnica de segmentación.

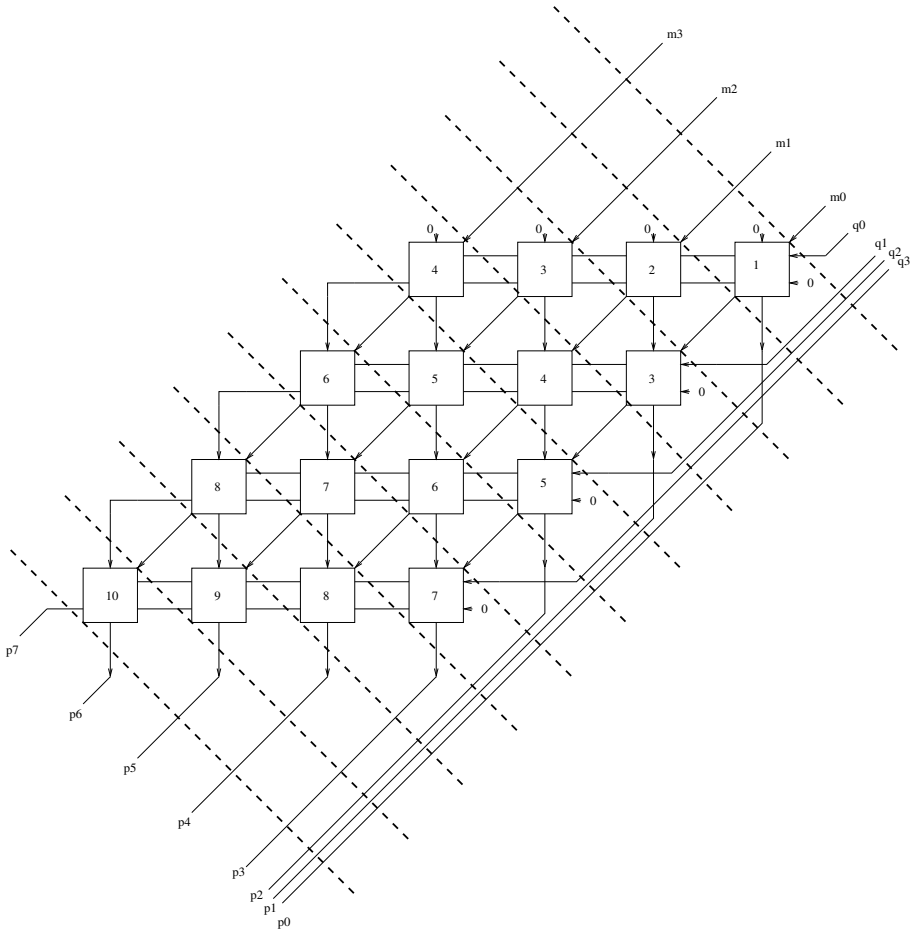

Figura 3.14: Multiplicador en array segmentado.

Generador de Productos Parciales junto a un Árbol de Wallace

El generador de productos parciales es un componente que crea múltiples productos parciales simultáneamente a partir de los bits de los operandos. Cada producto parcial representa el resultado de multiplicar un bit específico de un operando con todos los bits del otro operando.

El árbol de Wallace (figura 3.15) es una estructura combinacional que organiza la suma de los productos parciales en una jerarquía de sumadores que reduce progresivamente el número de términos hasta llegar a un resultado final. Esta estructura es altamente eficiente porque permite la reducción simultánea de múltiples sumas parciales en un solo ciclo de reloj. La eficiencia del árbol de Wallace radica en su capacidad para minimizar el número de niveles de suma necesarios, lo que resulta en una menor latencia comparada con otros métodos de reducción de sumas parciales.

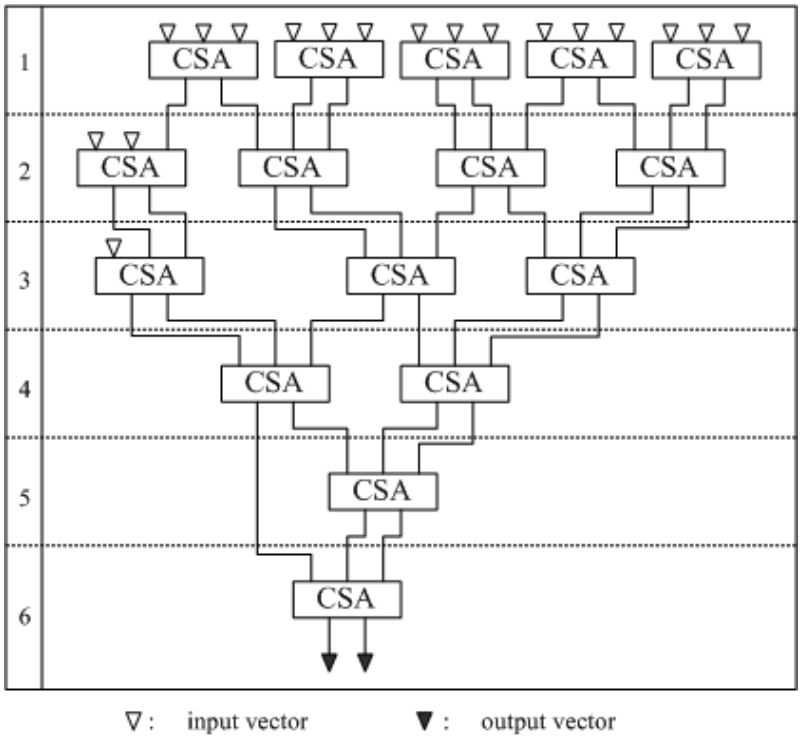

Figura 3.15: Árbol de Wallace de 16 bits segmentado en 6 etapas.

Las ventajas de utilizar un generador de productos parciales y un árbol de Wallace son numerosas. Primero, este enfoque permite una alta velocidad de operación, ya que todas las sumas parciales se procesan en paralelo, reduciendo significativamente la latencia de la multiplicación. Además, la estructura jerárquica del árbol de Wallace facilita la segmentación horizontal del diseño, lo que es esencial para nuestro objetivo de soportar el procesamiento vectorial. La segmentación permite que diferentes partes del multiplicador operen en diferentes ciclos de reloj, mejorando la eficiencia y el rendimiento general del sistema.

Otra ventaja significativa de esta implementación es su capacidad para escalar con el tamaño de los operandos. El generador de productos parciales y el árbol de Wallace pueden ser diseñados para manejar operandos de diferentes tamaños siguiendo la misma estructura de sumadores en árbol.

Adicionalmente, el uso de un árbol de Wallace mejora la distribución de carga dentro del multiplicador, equilibrando el número de operaciones de suma en cada nivel del árbol. Esto no solo mejora la velocidad sino que también optimiza el consumo de energía, ya que las operaciones pueden ser distribuidas más uniformemente a lo largo del diseño.

En términos de implementación práctica, la estructura modular del árbol de Wallace facilita su integración en un diseño más amplio. Esta modularidad también simplifica el proceso de verificación y prueba del multiplicador, asegurando que cada parte del diseño funcione correctamente antes de integrarlo en el sistema completo.

Después de analizar las ventajas y desventajas de los dos diseños, hemos decidido utilizar la implementación con un generador de productos parciales y un árbol de Wallace debido a sus ventajas en términos de velocidad, eficiencia y capacidad de segmentación. Este enfoque no solo cumple con nuestros objetivos de diseño combinacional, sino que también garantiza un buen rendimiento en procesamiento vectorial. La capacidad para procesar productos parciales en paralelo, la eficiencia en la reducción de sumas parciales y la facilidad de segmentación hacen de esta implementación la opción ideal para nuestro proyecto.

3.2.3. Multiplicador-Sumador Fusionados

El desarrollo de una unidad especializada en el operador de *Fused Multiply-Add* (FMA) marca un hito en la evolución de la computación de alto rendimiento y la inteligencia artificial. En la actualidad, prácticamente todos los procesadores modernos, desde CPUs convencionales hasta GPUs y aceleradores especializados, integran instrucciones dedicadas para realizar operaciones FMA. Entre los ejemplos más destacados, se incluyen las arquitecturas x86[3] de Intel[4] y AMD, así como las GPU de NVIDIA, que incorporan instrucciones FMA en sus conjuntos de instrucciones, aprovechando al máximo su eficiencia y potencia de procesamiento.

La amplia adopción del FMA se sustenta en su eficacia demostrada en una gran variedad de aplicaciones. En el ámbito de la inteligencia artificial, donde la velocidad de cálculo es crítica para el entrenamiento e inferencia de modelos complejos, el FMA proporciona una ventaja significativa al reducir drásticamente el tiempo de computación necesario para llevar a cabo operaciones matemáticas intensivas. Por ejemplo, en el procesamiento de redes neuronales profundas, que implica operaciones repetitivas de multiplicación y suma, el uso de FMA puede acelerar considerablemente el proceso de cálculo y mejorar los tiempos de entrenamiento de los modelos, lo que lleva a avances significativos en el aprendizaje automático (figura 3.16).

Además de su impacto en la inteligencia artificial, el FMA también es crucial en otras áreas de la computación de alto rendimiento. En simulaciones científicas, procesamiento de gráficos y cálculos financieros, la capacidad de realizar operaciones FMA de manera eficiente permite manejar grandes volúmenes de datos y complejas operaciones matemáticas con mayor rapidez. Esta mejora en el rendimiento se traduce en resultados más rápidos, beneficiando una amplia gama de aplicaciones científicas e industriales.

La operación FMA combina una multiplicación y una suma en una sola instrucción. Esta característica reduce la latencia y el consumo de energía al minimizar el número de instrucciones necesarias para llevar a cabo una tarea. Al fusionar múltiples operaciones en una sola, se disminuye la carga en la unidad de ejecución y se optimiza el uso de los recursos disponibles, lo que resulta en un rendimiento mejorado y una mayor eficiencia del sistema en general.[11].

[3] `https://es.wikipedia.org/wiki/X86`
[4] https://www.intel.la/content/www/xl/es/homepage.html

Otra ventaja clave del FMA es su versatilidad. Además de realizar operaciones FMA estándar, este operador puede adaptarse para realizar operaciones de multiplicación o suma por separado, según las necesidades del sistema. Por ejemplo, si todos los operadores de multiplicación y de suma convencionales están ocupados, este operador tiene la capacidad de ajustarse fácilmente para actuar como una de esas dos operaciones por separado. Ya sea sumando un "0" en el caso de que se necesite realizar una multiplicación o multiplicando por "1" en el caso de que se quiera realizar una suma.

Esta flexibilidad proporciona una mayor eficiencia en la asignación de recursos y optimiza el uso del hardware disponible, lo que resulta en una mayor capacidad de adaptación a diversas cargas de trabajo y requisitos de rendimiento. La versatilidad del FMA lo convierte en un componente valioso en cualquier diseño de unidad de cálculo, mejorando su utilidad en una amplia variedad de escenarios.

Dada la amplia gama de ventajas que hemos discutido previamente, concluimos que la integración de una unidad FMA puede optimizar el rendimiento y la eficiencia en diversas aplicaciones. Por lo tanto, hemos tomado la decisión de incorporarla en nuestro diseño, reconociendo su capacidad para acelerar cálculos intensivos y optimizar la utilización de recursos hardware. Esta decisión está motivada no solo por las mejoras en velocidad y eficiencia, sino también por la flexibilidad y adaptabilidad que el FMA aporta a nuestro sistema, asegurando que podamos manejar eficientemente una amplia gama de cargas de trabajo y requisitos de procesamiento.

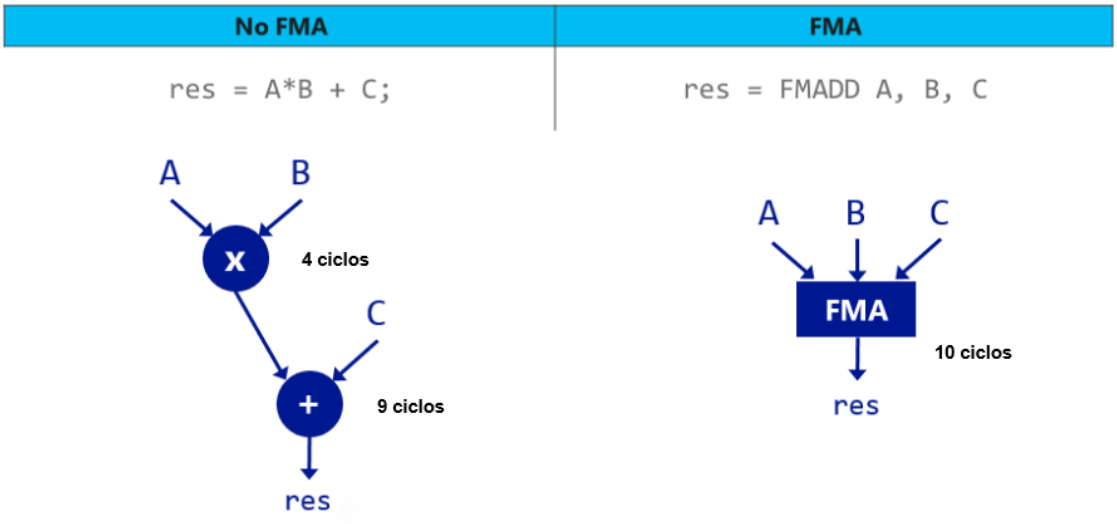

Figura 3.16: FMA vs multiplicaciones y sumas encadenadas.

3.2.4. Divisor

La división en coma flotante es una de las operaciones más costosas en términos de recursos y retardos debido a su gran complejidad y a las exigencias de precisión que impone el estándar IEEE 754. La correcta implementación de esta operación es crucial para desarrollar una unidad de coma flotante funcional y eficiente.

Entre todas las etapas para realizar correctamente la división, podemos encontrar: extraer el signo, exponente y mantisa de los operandos, restar los exponentes, añadir el resultado del exceso a la resta, dividir las mantisas, normalizar el resultado y redondear y renormalizar el resultado en caso de que sea necesario.

Entre las etapas mencionadas, nos centraremos en la división de mantisas, ya que es la etapa más costosa en tema de recursos y retardos. Hemos considerado dos tipos principales de métodos de división: el denominado *Fast Division*, destacando el método de *Newton-Raphson*, y el denominado *Slow Division*, que incluye la división con restauración (*Restorative*) y la división sin restauración (*Non-Restorative*).

Los métodos *Fast Division* se caracterizan por conseguir el resultado de la división a base de aproximaciones. El método de Newton-Raphson es un ejemplo representativo de esta categoría. Este método utiliza una serie de aproximaciones sucesivas para converger hacia el resultado de la división.

Se inicia con una estimación inicial del recíproco del divisor y, mediante multiplicaciones y sumas sucesivas, refina esta estimación hasta que alcanza un nivel de precisión aceptable. La ventaja principal del método de Newton-Raphson es su capacidad para generar resultados de forma rápida sin necesidad de hardware adicional complejo. Sin embargo, este método no garantiza un resultado exacto, lo que puede ser una limitación significativa para aplicaciones que necesiten de una precisión exacta.

Por otro lado, los métodos *Slow Division* incluyen procedimientos que aseguran una mayor precisión, aunque a costa de un mayor tiempo de cómputo y complejidad de hardware. Dentro de esta categoría, se encuentran la división con restauración (*Restorative*) y la división sin restauración (*Non-Restorative*). La división con restauración es un método tradicional que implica realizar una serie de restas sucesivas y, después de cada resta, verificar si se necesita restaurar el valor original del dividendo parcial. De esta forma conseguimos obtener un resto preciso.

La división sin restauración (figura 3.17) optimiza el proceso eliminando las operaciones de restauración en cada paso. En este método, se utilizan restas y desplazamientos para aproximar el cociente de forma más directa y eficiente. La ausencia de operaciones de restauración reduce la latencia y la complejidad del hardware, haciendo este método más adecuado para diseños que requieren un equilibrio entre precisión y eficiencia.

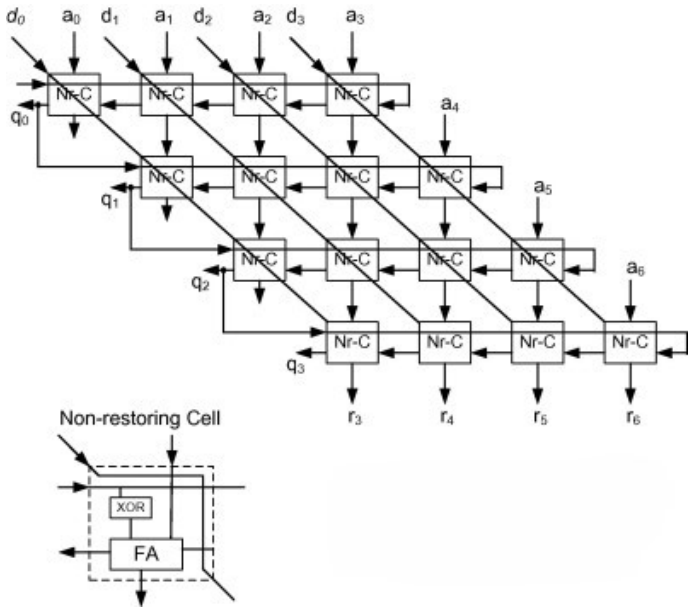

Figura 3.17: Divisor sin restauración.

Sin embargo, según un estudio comparativo[8], se ha observado que no hay una gran diferencia de retardos entre la división con restauración y sin restauración. Esto sugiere que la división con restauración puede ofrecer resultados precisos sin comprometer significativamente el rendimiento en términos de latencia. A su vez, el mismo estudio indica que la división con restauración consigue un resto exacto, a diferencia del método sin restauración. De esta forma, abrimos las puertas a una posible implementación de la operación de resto.

Al comparar estos métodos y teniendo en cuenta que implementar la restauración no presenta una gran diferencia en retardos, hemos decidido que la división con restauración ofrece la mejor combinación de ventajas para nuestra unidad de coma flotante. Aunque la *Fast Division* proporciona rapidez y simplicidad de hardware, su falta de precisión hace que no sea adecuada para cumplir con los estrictos requisitos del estándar IEEE 754. Por lo tanto, no puede ser el método usado para dar soporte a la operación de división en nuestra unidad de coma flotante.

Como conclusión, la división con restauración se destaca como la opción más viable para nuestro diseño. Este método garantiza la precisión necesaria tanto en el resultado de la división de mantisas como en el residuo.

CAPÍTULO 4
Diseño de Componentes

4.1 Estructura General de un Operador

En este apartado se describe la estructura general de los operadores de la unidad de coma flotante, incluyendo las señales y estructuras comunes que permiten su funcionamiento en sintonía con la unidad vectorial. La correcta integración y funcionamiento de estos operadores es crucial para aprovechar al máximo las capacidades de procesamiento paralelo y vectorial. Los operadores de la unidad de coma flotante están diseñados para realizar múltiples operaciones simultáneamente, permitiendo operaciones en vectores de datos.

Figura 4.1: Entradas y salidas comunes de los operadores

4.1.1. Entradas del Operador

- **CLK:** La señal de reloj (CLK) sincroniza todas las operaciones internas del operador, asegurando que los registros de segmentación funcionen correctamente y transmitan los datos en el momento adecuado. Cada ciclo de reloj marca un punto específico en el tiempo en el que se introducen datos en el operador, los datos avanzarán a través de las etapas y se obtienen los resultados en las salida del operador.
- ***Reset*:** Esta entrada de control tiene una función esencial en la inicialización y el mantenimiento del estado del operador. Cuando se activa esta señal, se pone a "0" la salida de los registros de segmentación, asegurando que el

operador comience desde un estado conocido y definido. El *reset* garantiza que no haya datos residuales o indeterminados en los registros, lo que podría afectar negativamente a las operaciones subsecuentes.

- **Operandos:** Cada operador recibe dos entradas principales de 33 bits, donde el bit de mayor peso (MSB) de cada entrada es el bit de dato válido, y los otros 32 bits son el valor numérico (figura 4.2). El bit de válido indica si el valor es apto para ser procesado. Ese bit se propagará por el operador asociado al valor númerico.

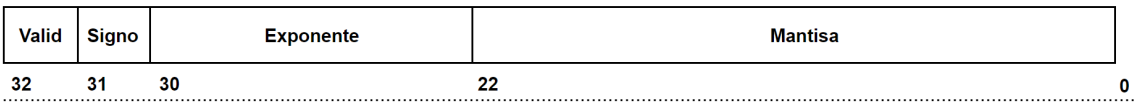

Figura 4.2: Bus de operandos para FP32.

- **Máscara:** El bit de máscara se propaga a través del operador hasta llegar a la etapa final. Este bit está asociado a los operandos y acompaña a los mismos durante todo el proceso de operación. Las operaciones con máscaras permiten realizar cálculos selectivos donde solo se procesan los elementos del vector que cumplen ciertas condiciones. La máscara actúa como un filtro que determina qué valores son relevantes para la operación actual y cuáles deben ser ignorados. Este mecanismo es especialmente útil en el procesamiento vectorial, donde la flexibilidad para manejar datos de forma condicional permite la vectorización de bucles con sentencias condicionales

- **Código de redondeo:** Esta entrada define el tipo de redondeo que se aplicará en la operación, estos códigos son los siguientes:

  - **00 -> Truncado** El resultado se trunca, eliminando cualquier dígito fraccional adicional sin considerar su valor.
  - **01 -> Redondeo al más cercano, empate al par** Redondea el número al valor más cercano. En caso de empate (cuando el número está exactamente en la mitad), se redondea al número par más cercano.
  - **10 -> Redondeo al +** ∞ El resultado se redondea al valor más cercano que sea mayor o igual al número original.
  - **11 -> Redondeo al -** ∞ El resultado se redondea al valor más cercano que sea menor o igual al número original.

- ***Start:*** La señal de inicio indica el comienzo de una operación específica. Hasta que esta señal no se establece en "1" , el operador permanece inactivo y no comenzará a procesar los datos. Esta característica garantiza que el operador solo empiece a ejecutar las operaciones cuando se le indique explícitamente, evitando así cualquier procesamiento prematuro. El uso de la señal de inicio asegura que los operadores estén sincronizados con el resto del sistema y entre ellos mismos.

4.1.2. Salidas del Operador

- **Resultado:** El resultado del operador se transmite a través de un bus de 34 bits, estructurado de la siguiente manera: el bit de mayor peso es el bit de válido, seguido por el bit de máscara y por los 32 bits del resultado.

  El bit de válido es una señal que indica si el resultado actual es correcto y está listo para ser utilizado. Si este bit está en "1" , el resultado en los 32 bits restantes es fiable y puede ser usado en cálculos posteriores. Si está en "0" , significa que el resultado no es válido y no debe ser utilizado.

  Este bit toma más importancia en la parte de control de la unidad vectorial que se menciona en la sección 1.4, allí, este bit sirve como contador para llevar la cuenta de los elementos que se van procesando hasta que se llega al valor del VLR(*Vector Length Register*) de la instrucción.

  A continuación, tenemos el bit de máscara. Este bit es usado para indicar al control si el resultado es significativo y se debe considerar en cálculos futuros. Nuestro diseño pondrá el bit de máscara a "0" si durante la operación ocurre alguna excepción, como *overflow*, *underflow* o NaN y además, si estamos realizando operaciones condicionales, y es un resultado significativo que no queremos almacenar. De esta forma indicamos que el resultado no es válido para el procesamiento futuro. Poniendo el valor del bit de máscara a "1" en caso contrario.

  Los restantes 32 bits del bus contienen el resultado de la operación. Esta estructura garantiza que solo los resultados válidos sean considerados, mejorando la robustez y precisión de las operaciones vectoriales. En ausencia de excepciones, el bit de máscara permanece con el valor a "1" , permitiendo el uso del resultado en operaciones futuras.

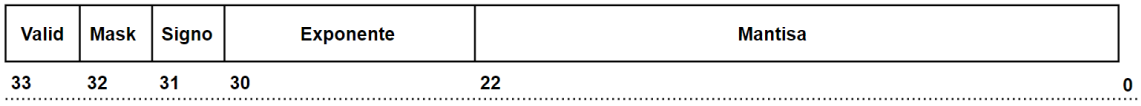

Figura 4.3: Bus de resultado para FP32.

- ***busy:*** Esta señal indica si el operador está actualmente ocupado procesando una operación. Es útil para realizar una gestión eficiente de los recursos del sistema y para evitar conflictos de datos. Cuando la señal de *busy* está activa, indica que el operador está realizando una operación y no está disponible para iniciar nuevas tareas. Esto asegura que no se inicien nuevas operaciones hasta que el operador haya terminado completamente la operación actual.

  La señal de *busy* facilita la sincronización entre diferentes componentes del sistema, permitiendo a la unidad de control determinar cuándo el operador está listo para recibir nuevas instrucciones.

4.2 Sumador/Restador

El sumador/restador es responsable de ejecutar operaciones aritméticas de sumas y restas de números en coma flotante. Su diseño maneja varios aspectos como la alineación de exponentes, ajuste de mantisas, manejo de signos y tratamiento de casos especiales, tales como *overflow* y *underflow*, conforme al estándar IEEE 754.

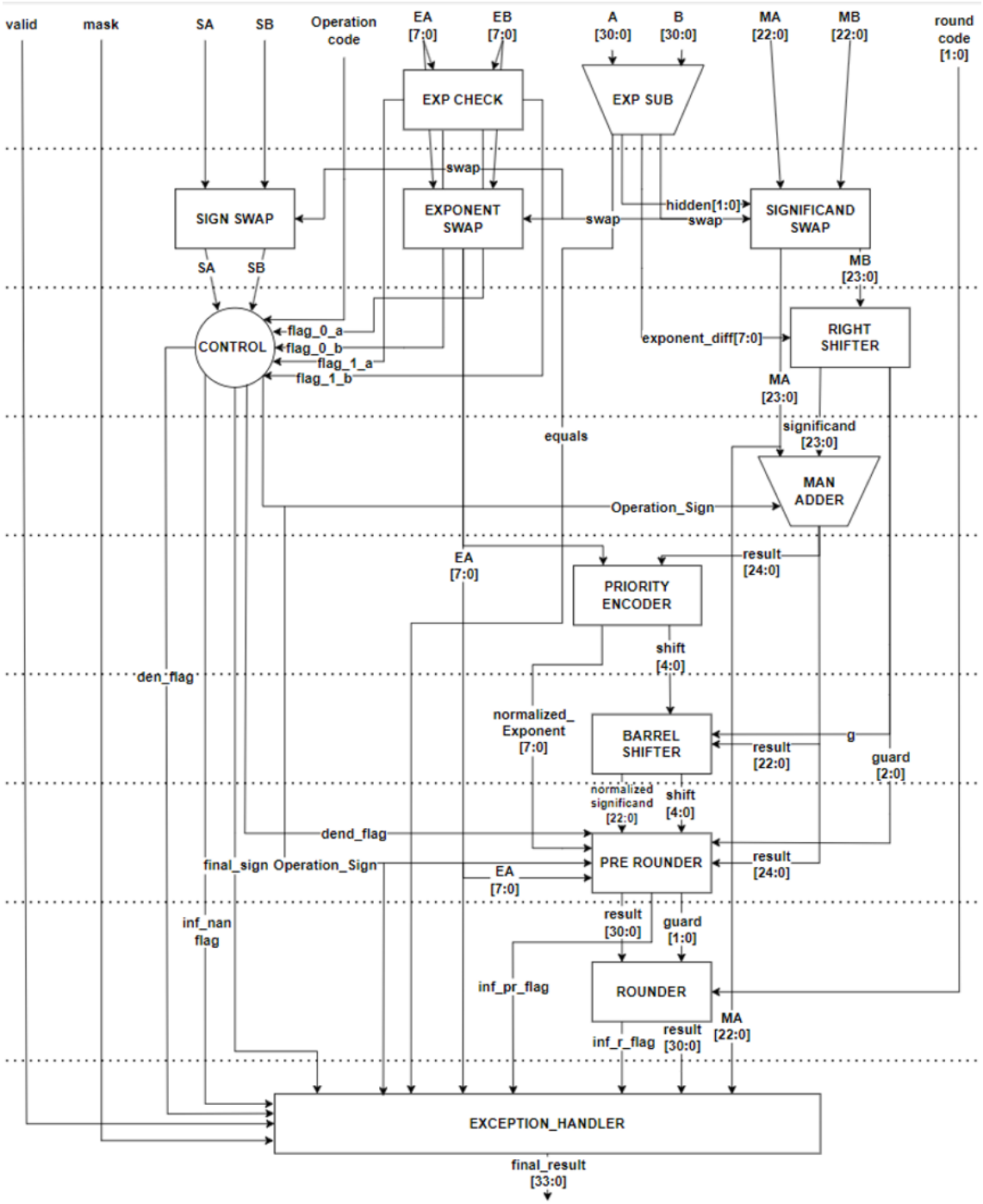

Figura 4.4: Sumador/Restador de coma flotante de 32 bits.

El operador, además de recibir las entradas mencionadas en la sección 4.1, recibe el código de operación, "0" para realizar la operación suma y "1" para la resta. Este codigo de operación se recibe una vez y se aplica para todos los operandos del vector.

Además de trabajar con operandos con formato FP32, a su vez, es capaz de operar con otros tipo de formatos de datos, entre los que destacamos TF32 y BF16, mencionados en el capítulo 2. Para gestionar estos formatos, el operador recibe por la entrada los parámetros que definen la longitud de cada campo del valor (exponente y mantisa) y adapta sus componentes internos para funcionar de forma correcta con el formato de datos específicado.

A continuación, se detallan los componentes que forman parte de un operador de suma/resta en coma flotante para el formato FP32 (figura 4.4) y su aportación a la obtención del resultado:

- **Exp Check:** Este módulo se encarga de detectar las excepciones en las entradas del operador, procesándolos para identificar casos especiales como infinito, NaN y números denormalizados. Este módulo toma como entrada los exponentes de los dos operandos y analiza sus valores para determinar el tipo de excepción, si la hubiera.

  Para detectar si un exponente corresponde a una excepción de infinito o NaN, el módulo verifica si el exponente es una secuencia completa de "1". La distinción entre estos dos casos se realiza mediante el análisis de la mantisa, pero en nuestro caso no es necesario ya que se tratarán de la misma forma.

  Por otro lado, para identificar una excepción de número denormalizado, el módulo examina si el exponente es todo "0". Un exponente completamente en "0", combinado con una mantisa no nula, indica un número denormalizado.

  Las señales resultantes de estas comprobaciones se envían a la unidad de control, la cual maneja las excepciones. Esta unidad asegura que en cualquier operación que se involucren estos valores se procese de manera conforme a las reglas establecidas.

- **Exp Sub:** Este módulo desempeña varias funciones más allá de lo que sugiere su nombre. En primer lugar, este módulo recibe los dos operandos sin el signo y procesa estos valores para determinar el bit implícito de las mantisas.

  Este paso es necesario para el correcto manejo de números tanto normalizados como denormalizados. Para los números normalizados, el bit implícito de la mantisa es siempre "1" , lo que indica que el número está en una forma estándar. Sin embargo, para los números denormalizados, donde el exponente es todo ceros, el bit implícito es "0" , lo que requiere un tratamiento especial.

Además de determinar el bit implícito de las mantisas, el módulo también se encarga de realizar la resta de exponentes, esta diferencia será utilizada por módulos posteriores para desplazar las mantisas antes de sumarlas y asegurar que el resto de etapas se realicen correctamente.

Otra función importante del módulo es realizar la comparación de los operandos para generar una señal de intercambio. Si el operando B resulta ser mayor que el operando A, se activa una señal de intercambio. Esto garantiza que, para el correcto funcionamiento del operador, el operando A sea siempre el más grande de los dos.

Finalmente, el módulo también genera una señal para manejar el caso especial donde los operandos son iguales y se debe realizar una resta. Esta situación requiere un tratamiento específico para evitar errores en el cálculo y asegurar que el resultado sea correcto, ya que este caso no es cubierto por el caso base, al siempre tener en el resultado el exponente más grande en valor.

- **Sign Swap:** Este módulo recibe los signos de los dos operandos junto con la señal de intercambio (swap) que proviene del módulo restador de exponentes. El modulo intercambia los signos de los operandos cuando la señal de swap lo indique.

- **Exponent Swap:** Este módulo desempeña una función similar al módulo Sign Swap, pero se encarga de intercambiar los exponentes en lugar de los signos. Este módulo recibe los exponentes de los dos operandos junto con la señal de intercambio (*swap*) que proviene del módulo restador de exponentes. El módulo intercambia los exponentes de los operandos cuando la señal de *swap* lo indique y saca por su salida el exponente más grande.

- **Significand Swap:** Con este módulo completamos la trilogía de componentes encargados de gestionar el intercambio de elementos del operador para asegurar un resultado preciso. En este caso, nos enfocamos en las mantisas. El módulo recibe como entrada las mantisas de los dos operandos, la señal de intercambio (*swap*) y la señal de bit implícito (*hidden*), que provienen del restador de exponentes.

  El módulo no solo intercambia las mantisas si la señal de swap así lo indica, sino que también añade el bit implícito en la parte más significativa de la mantisa según lo indique la señal de *hidden*. Este bit implícito, que puede ser un "1" o un "0", dependiendo de si el número es normalizado o denormalizado respectivamente.

- **Control:** Este módulo desempeña varias funciones importantes dentro de nuestro operador. Entre ellas, destaca el empaquetamiento de las excepciones detectadas desde el input principal y la señal de *underflow* proveniente del la resta de exponentes, para su posterior tratamiento en el manejador de excepciones.

Otra función a destacar es informar al sumador de mantisas para que realice la operación correcta (suma o resta) y calcular el signo final de la operación analizando el código de operación, además de los signos de los operandos.

- **Right Shifter:** Este módulo, como su nombre indica, se encarga de realizar desplazamientos a la derecha en la mantisa más pequeña para alinear los exponentes.

  Durante este proceso de desplazamiento, los bits que se descartan se convierten en bits de guarda, que se utilizan posteriormente para redondear el resultado final con mayor exactitud. Específicamente, de estos bits descartados se derivan el bit de guarda (*guard*), el bit de redondeo (*round*) y los *sticky bits*. Estos bits adicionales proporcionan información valiosa que permite decidir si es necesario ajustar el resultado final para mantener la precisión en las operaciones de punto flotante.

- **Man Adder:** Este módulo es el encargado de realizar la suma o resta de las mantisas de los operandos, según lo indicado por la unidad de control.

  Para ejecutar estas operaciones, el módulo emplea un sumador CLA de dos niveles jerárquicos de acarreo, como se explicó detalladamente en el capítulo 3. Este sumador es parametrizable, lo que permite adaptar su funcionamiento a las necesidades específicas de cada formato de datos soportado simplemente cambiando los parámetros de entrada.

- **Priority Encoder:** La función principal de este módulo consiste en identificar la posición en la que se encuentra el bit más significativo de la mantisa con valor "1", de esta forma conseguimos normalizar el exponente desplazando al mismo la posición calculada.

- **Barrel Shifter:** Este módulo trabaja en conjunto con el Priority Encoder, ya que recibe la cantidad de desplazamientos que se deben realizar a la mantisa resultado, calculada en el módulo anterior. Con esta información, junto con el bit de guarda, el Barrel Shifter efectúa los desplazamientos a la izquierda en la mantisa resultado.

  Durante el primer desplazamiento, el bit de guarda se introduce en la mantisa. En los desplazamientos subsecuentes, se introducen ceros, garantizando que la mantisa se ajuste correctamente para su normalización.

- **Pre Rounder:** Este módulo se encarga de recopilar los resultados proporcionados por el Priority Encoder y el Barrel Shifter, junto con los bits adicionales necesarios para el redondeo, y empaquetar estos resultados para enviarlos a la unidad de redondeo final.

  Dentro de esta unidad, se tiene en cuenta la cantidad de desplazamientos realizados en la mantisa por el Barrel Shifter para determinar los bits utilizados para el redondeo. Se diferencian tres casos específicos: cuando no se han realizado desplazamientos, cuando se ha realizado uno, y cuando se han realizado más de uno.

En paralelo, este módulo maneja correctamente las operaciones con dos números denormalizados, detectando los casos mediante los análisis a los operandos que se han ido realizando en etapas anteriores y evitando que los resultados pasen por etapas como el Priority Encoder o el Barrel Shifter que pueden estropear el resultado. Además, realiza comprobaciones de normalización en casos de suma analizando los bits de acarreo en la suma de mantisas para asegurar la exactitud del resultado final.

- **Rounder:** Este módulo tiene la responsabilidad de efectuar el redondeo del valor recibido del módulo anterior. Tal como se ha detallado en la sección 4.1, el módulo recibe un código de entrada que especifica el modo de redondeo a utilizar para la operación.

  Con esta información, el módulo aplica el redondeo correspondiente, asegurando además que el resultado se mantenga normalizado incluso si el redondeo provoca una denormalización del valor. Además, el módulo está diseñado para detectar y gestionar la excepción de desbordamiento (*overflow*) que podría surgir durante el proceso de redondeo.

- **Exception Handler:** Este módulo tiene la responsabilidad de gestionar las excepciones que se generan a lo largo de todo el operador. La mayoría de estas excepciones han sido previamente empaquetadas y enviadas por la Unidad de Control para su procesamiento final.

  Dentro del mismo se abordan casos específicos, tales como operaciones con números mixtos, es decir, entre números normalizados y denormalizados, así como situaciones particulares como la resta entre dos números iguales,las operaciones entre dos números denormalizados y las excepciones como NaN o infinito. Estas excepciones se resuelven poniendo el bit de máscara a "0" para que así la unidad vectorial no tenga en cuenta el valor para sus cálculos.

  El módulo está diseñado para identificar y procesar adecuadamente cada tipo de excepción, aplicando las correcciones necesarias o señalando condiciones especiales que deben ser manejadas de manera distinta. Por ejemplo, en el caso de la resta entre dos números iguales, se implementa una lógica específica para garantizar que el resultado sea correcto y que cualquier posible desbordamiento o subdesbordamiento sea debidamente detectado y tratado.

4.3 Multiplicador

El multiplicador es la unidad responsable de ejecutar las operaciones de multiplicación de números en coma flotante. Su diseño abarca diversos aspectos tales como el cálculo del signo del resultado, la multiplicación de mantisas, la suma de exponentes, la normalización y el redondeo del resultado y el tratamiento de casos especiales como números denormalizados o desbordamientos.

Este multiplicador está diseñado para trabajar con los formatos de datos TF32 y BF16 en conjunto con FP32. Este recibe por su entrada un código de formato. Siendo "00" para FP32, "01" para TF32 y "11" para BF16. De esta forma los módulos se adaptan y manejan las características de cada uno.

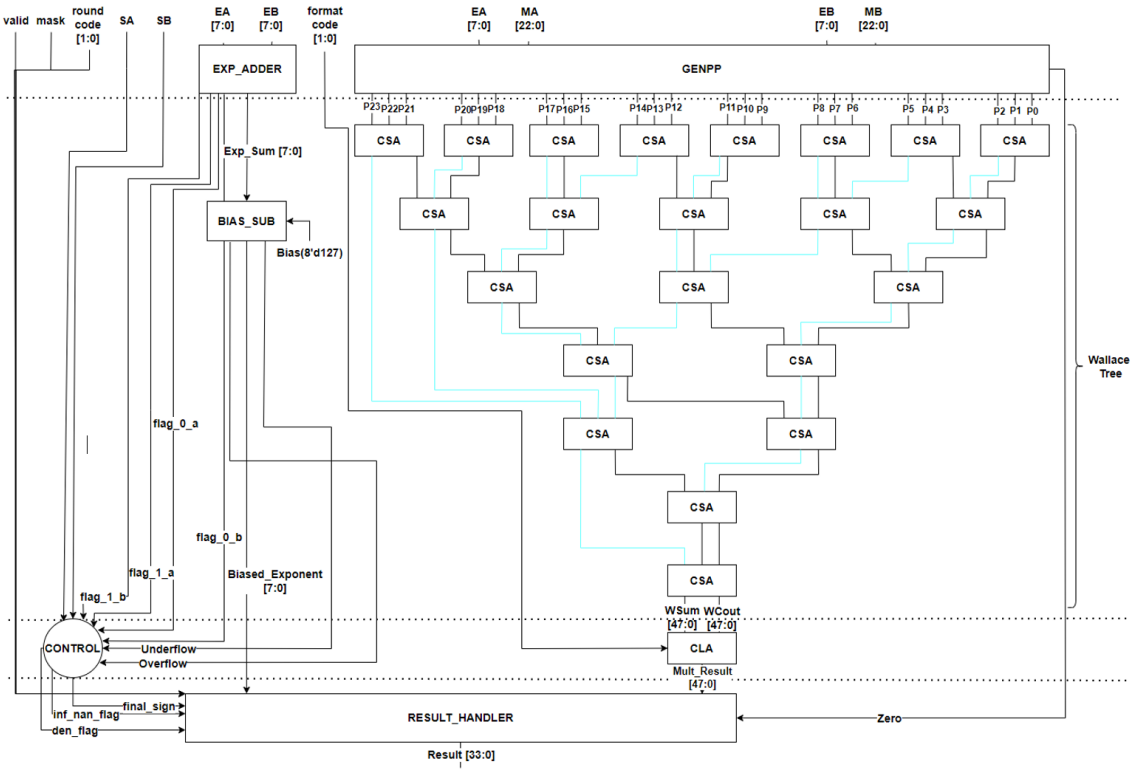

Figura 4.5: Multiplicador de coma flotante de 32 bits.

En la figura 4.5, se muestra el esquema completo de un multiplicador para FP32. A continuación se muestra un análisis de cada componente que forma parte de este multiplicador y se detalla su funcionamiento interno:

- **Exp Adder:** La principal función de este módulo es realizar la suma de los exponentes de los dos operandos. A su vez, este módulo tiene la capacidad de detectar y procesar excepciones en las entradas del operador. Esto incluye la identificación de casos especiales como infinito, NaN y números denormalizados, siguiendo las mismas reglas y procedimientos que se aplican en el operador de suma/resta de coma flotante.

  Las señales que indican estas excepciones son enviadas a la unidad de con-

trol para su posterior manejo junto con otras señales procedentes de otros módulos, asegurando que el sistema responda correctamente a estas condiciones especiales.

- **Bias Sub:** El mayor objetivo de este módulo es ajustar la suma de los exponentes de los dos operandos mediante la resta del exceso. Como se explicó en el capítulo 2, específicamente en la figura 2.1, el exponente en el estándar IEEE 754 se representa en notación científica con un exceso. Al sumar dos exponentes, este exceso se cuenta dos veces, por lo que es necesario realizar esta resta para obtener un resultado correcto.

  Este módulo también es responsable de la detección del *underflow*. La detección de esta excepción se logra analizando patrones específicos en los bits del resultado. Estas señales son enviadas a la unidad de control para su posterior manejo, asegurando que cualquier anomalía en los resultados de la multiplicación sea tratada.

- **Control Unit:** Este módulo se encarga de recoger y gestionar todas las excepciones que provienen de los módulos de suma de exponentes y resta de exceso descritos anteriormente. Las excepciones incluyen situaciones como *underflow*, infinito, NaN y números denormalizados. La unidad analiza estas excepciones, las empaqueta en un formato adecuado y las envía al manejador de excepciones, para su posterior tratamiento.

  A su vez, la unidad de control también es responsable de calcular el signo final del resultado de la multiplicación. Para ello, aplica una función lógica a los signos de los operandos de entrada. Dado que la multiplicación de dos números en coma flotante sigue las reglas de los signos de la aritmética básica (es decir, el producto de dos números con el mismo signo es positivo, mientras que el producto de dos números con signos opuestos es negativo), esta evalúa los signos de los operandos y determina el signo adecuado para el resultado final.

- **Genpp:** Este módulo recibe como entradas las mantisas y los exponentes de los dos operandos que se desean multiplicar. Su primera tarea es añadir el bit implícito a las mantisas, utilizando el mismo método y criterio que se emplea en el sumador/restador de coma flotante, analizando el exponente. Este bit implícito es necesario para garantizar la precisión de los cálculos. En los números normalizados en formato IEEE 754, el bit implícito de la mantisa siempre es "1", mientras que en los números denormalizados es "0".

  Una vez que las mantisas han sido ajustadas con sus respectivos bits implícitos, el generador de productos parciales realiza su función principal. Generar los productos parciales a partir de las mantisas de los operandos para luego enviarselas al árbol de Wallace.

  Los productos parciales son los resultados intermedios obtenidos durante el proceso de multiplicación de los dígitos de las mantisas. En el contexto de la multiplicación binaria, cada bit de la mantisa multiplicador se multiplica por todos los bits de la mantisa multiplicando, produciendo una serie

de productos que, cuando se suman correctamente, dan como resultado el producto final.

- **Wallace Tree:** La función de este módulo es realizar la suma de los productos parciales que provienen de su generador. La estructura interna de este módulo está compuesta por una serie de sumadores con propagación de acarreo (CSA) dispuestos en forma de árbol.

  Un CSA es un tipo de sumador que permite la suma de tres o más números binarios simultáneamente. Cada uno toma tres valores de entrada (A, B y D) y produce dos valores de salida: una suma parcial (S) y un acarreo (C). Estos sumadores parciales y acarreos se envían a los niveles inferiores del árbol de Wallace, reduciendo gradualmente el número de términos hasta que se obtiene el resultado final. La principal ventaja de los CSA es que aceleran el proceso de suma, ya que el tiempo de propagación del acarreo se minimiza considerablemente.

  Si observamos detenidamente el esquema, los cables marcados en azul indican que el acarreo de un CSA se ha desplazado un bit a la izquierda. Este desplazamiento es necesario porque cada bit de acarreo calculado se debe acumular sobre el bit inmediatamente superior. De tal manera que se cumpla que, si las entradas del CSA son A, B y C y las salidas son S y C, entonces $A + B + D = S + 2C$.

- **CLA:** Este módulo tiene como misión realizar la suma del último acarreo y de la última suma parcial provenenientes del árbol de Wallace. Como se mencionó en el caso de estudio del capítulo 3 y como el propio nombre indica, hemos seleccionado un CLA parametrizable que soporta los 3 tipos de formatos soportados y dos niveles de jerarquía de acarreo para esta tarea, ya que nos ofrece entre sus ventajas un retardo reducido y su capacidad para manejar sumas de gran tamaño de manera eficiente.

- **Result Handler:** Este módulo tiene la tarea principal de recopilar todos los resultados parciales, normalizarlos y redondearlos según los requisitos de precisión de cálculo establecidos por la unidad vectorial.

  En operaciones con números denormalizados, teniendo en cuenta que estos números se aproximan al valor 0.0 y las características del resultado de una multiplicación, cuando algún operando se acerca mucho a 0, en nuestro diseño hemos optado por redondear el resultado al valor 0.0.

  Desde el sumador final (CLA), se reciben el doble de bits dado que, al multiplicar números de n bits, el resultado ocupará 2n bits. La mitad de los bits (aquellos situados en la parte baja) son capturados y sometidos a un análisis detallado antes de aplicar el proceso de redondeo de manera similar a la realizada en la suma de coma flotante, utilizando los bits de *guard*, *round* y *sticky*, y aplicándoles funciones lógicas para definir el resultado final.

Al mismo tiempo, se realiza un análisis de las diversas señales que pueden surgir de la unidad de control, incluyendo, entre otras, las señales de *underflow* o NaN. En el caso específico de la excepción de multiplicación por 0, se implementa una medida directa en la que el resultado se establece automáticamente en 0.0, eliminando así cualquier cálculo incorrecto que pueda provenir del algoritmo de multiplicación genérico.

Para las demás excepciones, se sigue un procedimiento similar al utilizado en el sumador de coma flotante. Esto implica ajustar el bit de máscara a 0 para que el resultado no se tenga en cuenta en cálculos posteriores, asegurando que el sistema maneje correctamente estas condiciones excepcionales y mantenga la integridad del cálculo final.

4.4 Multiplicador-Sumador Fusionados

Este operador es el encargado de realizar las operaciones de multiplicación y de suma en la misma instrucción. Al igual que el multiplicador, está diseñado para soportar los formatos de datos TF32 y BF16 además de FP32. Esto lo consigue al igual que el multiplicador, recibiendo el código del formato por la entrada.

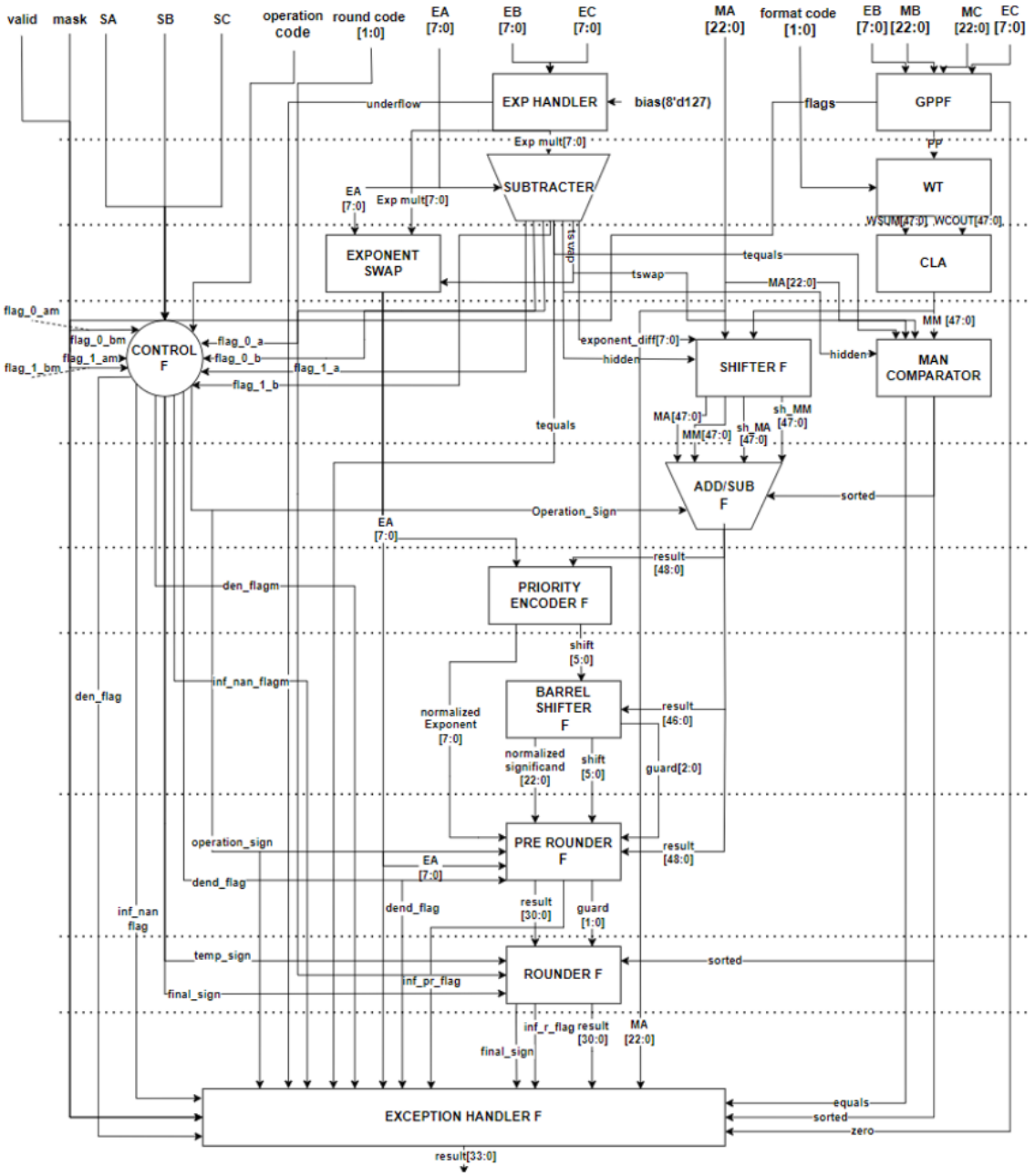

Figura 4.6: FMA de coma flotante de 32 bits.

La diferencia entre este operador y realizar una multiplicación y una suma encadenadas reside en la disposición de los diferentes módulos, ya que como podemos observar en la figura 4.6, las fases de la multiplicación están alineadas con

las primeras fases de la suma/resta. Consiguiendo así reducir el número de ciclos necesarios para producir un resultado.

A pesar de ser un operador diferente, este comparte muchas características con los operadores de suma/resta y multiplicación. A continuación detallaremos las características y el funcionamiento interno de cada módulo:

- **Exp Handler:** La función principal de este módulo consiste en realizar las operaciones necesarias a los exponentes de los números a multiplicar, es decir, sumarlos y luego restarles el exceso al igual que hacíamos en el operador de multiplicación.

  A su vez, este módulo también detecta si existe *underflow* a la hora de restarle el exceso a la suma de exponentes, enviando la señal al manejador para que este la trate más tarde.

- **Gppf:** Este módulo repite la mayoría de las funciones que tiene el generador de productos parciales en el operador de multiplicación. Añade el bit implícito a las mantisas con el mismo criterio para números normalizados y denormalizados, y además calcula los productos parciales para el árbol de Wallace.

  La función añadida, es la de detectar las excepciones de los operandos que realizan la multiplicación, ya sean de infinito/NaN o de número denormalizado mediante el análisis del exponente. Generando señales que se enviarán a la unidad de control para su posterior tratamiento.

- **Subtracter:** En este módulo recibe el exponente del operando A y el exponente de la multiplicación. Con estos valores se realiza la resta de los mismos y se consigue su diferencia. Este valor será usado para desplazar las mantisas y así ajustar los exponentes para realizar correctamente la suma o resta.

  Además, este se encarga de generar una señal de intercambio y una señal de igualdad parcial, de las cuales su uso se explicará en los siguientes módulos.

- **Wallace Tree:** La función de este módulo es exactamente igual al módulo detallado en el multiplicador. Realiza la suma de los productos parciales que provienen de su generador.

- **Exponent Swap:** Este módulo se encarga de intercambiar los exponentes si la señal de intercambio que proviene del Subtracter así lo indica. Con esto nos aseguramos de tener el exponente más grande para el resultado de la operación de suma/resta.

- **CLA:** El módulo tiene una estructura interna y función análoga al CLA detallado en el operador de multiplicación. Realiza la suma del último acarreo y de la suma parcial que provienen del árbol de Wallace mediante una estructura con 2 niveles de jerarquía de acarreo.

- **Control F:** Una de sus funciones es la de recibir todas las excepciones (infinito, NaN...) que se han ido generando en los módulos anteriores, empaquetarlas y mandarlas al manejador para su posterior tratamiento.

  A su vez, es el encargado de calcular el signo de la multiplicación (aplicando la regla de signos), generar el signo de la operación de suma/resta y enviar al sumador de mantisas la operación a realizar, de igual manera que lo realiza la unidad de control del operador de suma/resta.

- **Shifter F:** Este módulo se encarga de ajustar las mantisas a 48 bits y añadir el bit implícito a la mantisa del operador A. Después, realiza los desplazamientos a la derecha de las mantisas el número de veces dado por la diferencia de los exponentes de la suma.

  Como en este diseño no hemos conseguido obtener de momento que operando es más grande que otro, este módulo realiza el desplazamiento en paralelo de las 2 mantisas, consiguiendo así obtener el resultado correcto teniendo en cuenta las limitaciones de nuestra implementación.

- **Man Comparator:** Este módulo es el encargado de asegurar que la suma de mantisas se realice con los valores correctos. Este recibe la mantisa resultado de la multiplicación y la compara con la mantisa del operando A, para así poder afirmar con certeza que operando es más grande.

- **Add Sub F:** La función de este módulo es realizar la suma de mantisas. Escogiendo los operandos adecuados gracias al módulo anterior. Para eso, utiliza un sumador CLA de 48 bits parametrizable con 2 niveles de jerarquía de acarreo.

- **Priority Encoder F:** El módulo realiza la misma función que el Priority Encoder del sumador/restador. Identifica la posición en la que se encuentra el bit más significativo de la mantisa con valor "1".

- **Barrel Shifter F:** Se encarga de realizar desplazamientos a la izquierda a la mantisa resultado el número de veces dado por el Priority Encoder F.

  Además, al recibir 47 bits y solo necesitar para el resultado los 23 de mayor peso, los bits restantes son usados para el redondeo, obteniendo el bit de guarda, de redondeo y los *sticky bits*.

- **Pre Rounder F:** Este módulo se encarga de recopilar los resultados proporcionados por el Priority Encoder F y el Barrel Shifter F, junto con los bits adicionales necesarios para el redondeo, y empaquetar estos resultados para enviarlos a la unidad de redondeo final.

  De igual forma que el Pre Rounder del sumador/restador, tiene en cuenta el número de desplazamientos realizados por el Barrel Shifter F para terminar de determinar los bits del redondeo. Se encarga de realizar comprobaciones en el caso de operaciones con dos números denormalizados y en la operación de suma para asegurar la exactitud del resultado final.

- **Rounder F:** Realiza una función análoga al módulo Rounder del operador de suma/resta. Este módulo recibe un código desde el input principal que especifica el modo de redondeo a usar en la operación.

  Con ese dato y los bits de guarda recibidos por el módulo anterior, aplica el redondeo adecuado y mantiene el resultado normalizado incluso si el redondeo provoca una denormalización del valor. Gestionando la excepción de *overflow* que pudiera surgir.

- **Exception Handler F:** Este módulo es el encargado de manejar todas las excepciones que se han ido generando durante todo el proceso de la operación. Recibiendo la mayoria ya empaquetadas por la unidad de control.

  Dentro del mismo se tienen en cuenta los casos particulares que se pueden generar, como por ejemplo, la resta entre 2 números iguales, operaciones con números denormalizados o excepciones como infinito o NaN. En este caso, y al igual que en todos los operadores, el bit de máscara se ajusta a 0 para que el resultado no se considere en cálculos posteriores.

4.5 Divisor

La división es una operación crítica en el diseño de unidades de cálculo, ya que es la operación básica que más ciclos de reloj consume, convirtiéndose en un potencial cuello de botella para los diseños. Por este motivo, encontrar un diseño óptimo para este operador es especialmente importante. Para lograrlo, es indispensable segmentar correctamente el operador, como explicamos en el capítulo 2.

En la figura 4.7 podemos observar el esquema completo de un divisor para FP32. Para ayudar con la facilidad de representación, las etapas en la que está segmentado el divisor de mantisas se muestra dentro del módulo. En esta sección se realiza un análisis de cada componente y se detalla su funcionamiento interno:

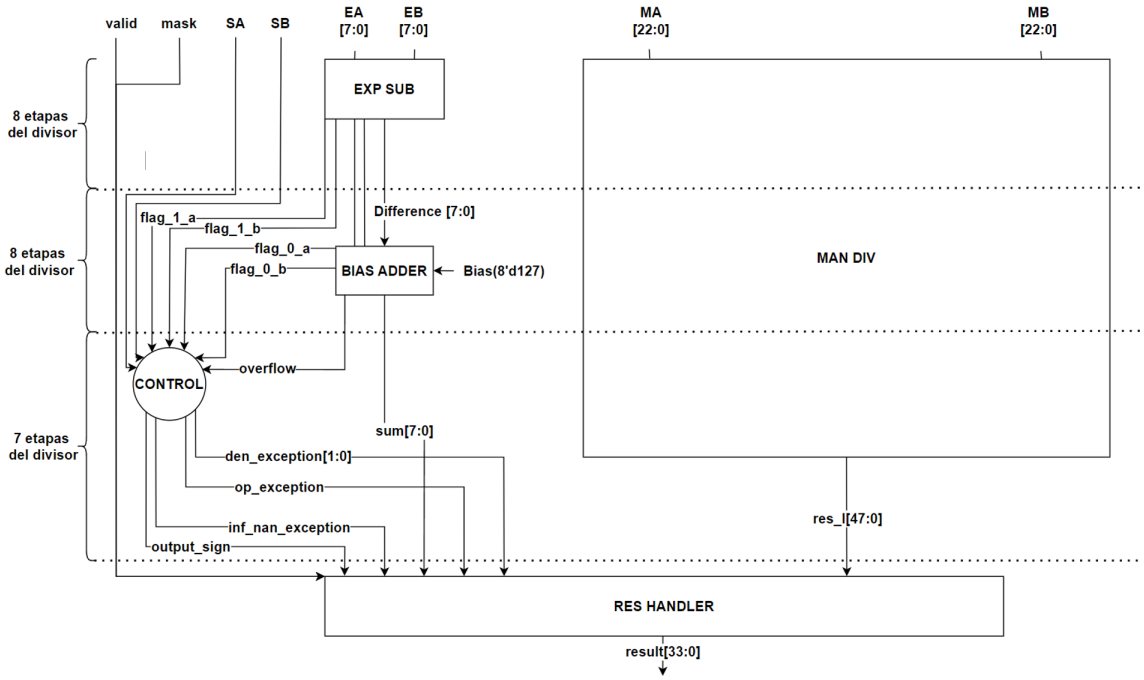

Figura 4.7: Divisor de coma flotante de 32 bits.

- **Exp Sub:** La principal tarea de este módulo es restar los exponentes de los operandos. En paralelo, este módulo puede detectar y manejar excepciones en las entradas del operador. Esto incluye identificar casos especiales como infinito, NaN y números denormalizados, utilizando los mismos métodos y procedimientos que se han ido usando en los anteriores operadores.

  Las señales que indican estas excepciones son enviadas a la unidad de control para su posterior tratamiento, junto con otras señales de otros módulos. Esto garantiza que el sistema responda adecuadamente a estas condiciones especiales, asegurando la precisión y fiabilidad en las operaciones de división en coma flotante.

- **Bias Adder:** Este módulo tiene la función de tomar la resta de los exponentes calculada en el módulo anterior y sumarle el exceso. Esto se hace para

ajustar el exponente al formato de representación IEEE 754, similar a la razón detrás del restador de exceso en el multiplicador de coma flotante. En este estándar, los exponentes se representan con notación de exceso para evitar valores negativos. Al restar dos valores durante la división, el exceso se resta dos veces y es necesario ajustar el resultado para mantener la precisión correcta.

A su vez, el módulo cuenta con mecanismos para detectar excepciones de *overflow* mediante el análisis del resultado generado por el operador y del úlitmo acarreo. Cuando el módulo identifica que se ha producido esa excepción, genera una señal específica que se envía a la unidad de control.

- **Control:** La principal responsabilidad de este módulo es recoger y gestionar todas las excepciones que provienen de los módulos de resta de exponentes y suma de exceso descritos anteriormente. Estas excepciones incluyen situaciones necesarias de controlar como *overflow*, infinito, NaN y números denormalizados.

  El módulo de control analiza cada una de estas excepciones, las empaqueta en un formato adecuado y las envía al manejador de resultado para su tratamiento posterior. Este proceso garantiza que cualquier anomalía detectada durante las operaciones previas sea adecuadamente gestionada, manteniendo así la precisión y la integridad de la operación.

  En paralelo a la gestión de excepciones, la unidad de control también tiene la tarea de calcular el signo final del resultado de la división. Esto se logra aplicando la regla de los signos de la aritmética básica, asegurando que el signo del resultado sea correcto basado en los signos de los operandos de entrada.

- **Man Div:** En este módulo es donde se invierte la mayor cantidad de recursos y de tiempo de cómputo en la operación de división de coma flotante. Su principal función es realizar la división de las mantisas con el bit implícito ya concatenado en la parte alta. Para nuestra unidad, se ha diseñado un divisor de 48/24 bits, debido a que en esta operación sucede lo inverso a la multiplicación, perdemos bits con la operación. Por ese motivo, necesitamos extender el numerador a 48 bits y desplazar el valor a la parte alta para poder realizar correctamente la división.

  La estructura interna de este módulo, como se muestra en la figura 4.8, está compuesta por un gran número de unidades de procesamiento (*processing units*) organizadas en niveles y segmentadas de forma que se forman 2 niveles por cada bloque de segmentación. Estas unidades, mediante sus conexiones, forman un divisor de enteros siguiendo las pautas del algoritmo de división con restauración.

  El algoritmo de división con restauración es un método clásico utilizado en la aritmética binaria para realizar divisiones de enteros. Nuestro diseño combinacional lo aplica de la siguiente forma:

Los bits del dividendo se van desplazando uno a la izquierda en cada momento de la operación, después de cada desplazamiento a la izquierda, el resultado parcial (rp) de cada nivel es comparado con el divisor, en el caso de que este resultado parcial sea mayor que el divisor se coloca un 1 en el cociente vinculado al nivel de *processing units* y se le resta el divisor al resultado parcial y baja al siguiente nivel. En caso contrario, se coloca un 0 en el cociente relacionado con el nivel y el resultado parcial baja al otro nivel sin ser modificado.

Una característica peculiar del diseño[8] es que los primeros 24 niveles de *processing units* están compuestos por 24 unidades, pero a partir del nivel 25, estos niveles pasan a ser formados por 25 unidades, esto se debe a que, para el resto necesitamos como máximo los mismos bits del divisor, ya que el valor máximo teórico que puede tener es $2^M - 2$ donde M es el nº de bits del divisor.

- **Res Handler:** Este módulo es el encargado de recopilar todas las operaciones realizadas en los módulos anteriores y normalizarlos sin llegar a redondearlos, ya que en el caso de la división, al realizar una divisón de 48/24, el resultado ya consta de 24 bits y no se descartan bits significativos en ninguna parte del proceso. Por ende, el divisor de mantisas produce un resultado con la precisión suficiente desde el primer momento.

  Finalmente, este módulo realiza un análisis de las diversas señales que pueden surgir de la unidad de control, incluyendo, entre otras, las señales de overflow y NaN. En el caso específico de la excepción de división por 0, se implementa una medida directa donde el resultado se establece automáticamente en excepción por infinito o NaN, eliminando cualquier cálculo incorrecto.

  Para las demás excepciones, se sigue un procedimiento similar al utilizado en los demás operadores. Esto implica ajustar el bit de máscara a 0 para que el resultado no se considere en cálculos posteriores, a su vez que asegurando así la integridad y precisión de las operaciones.

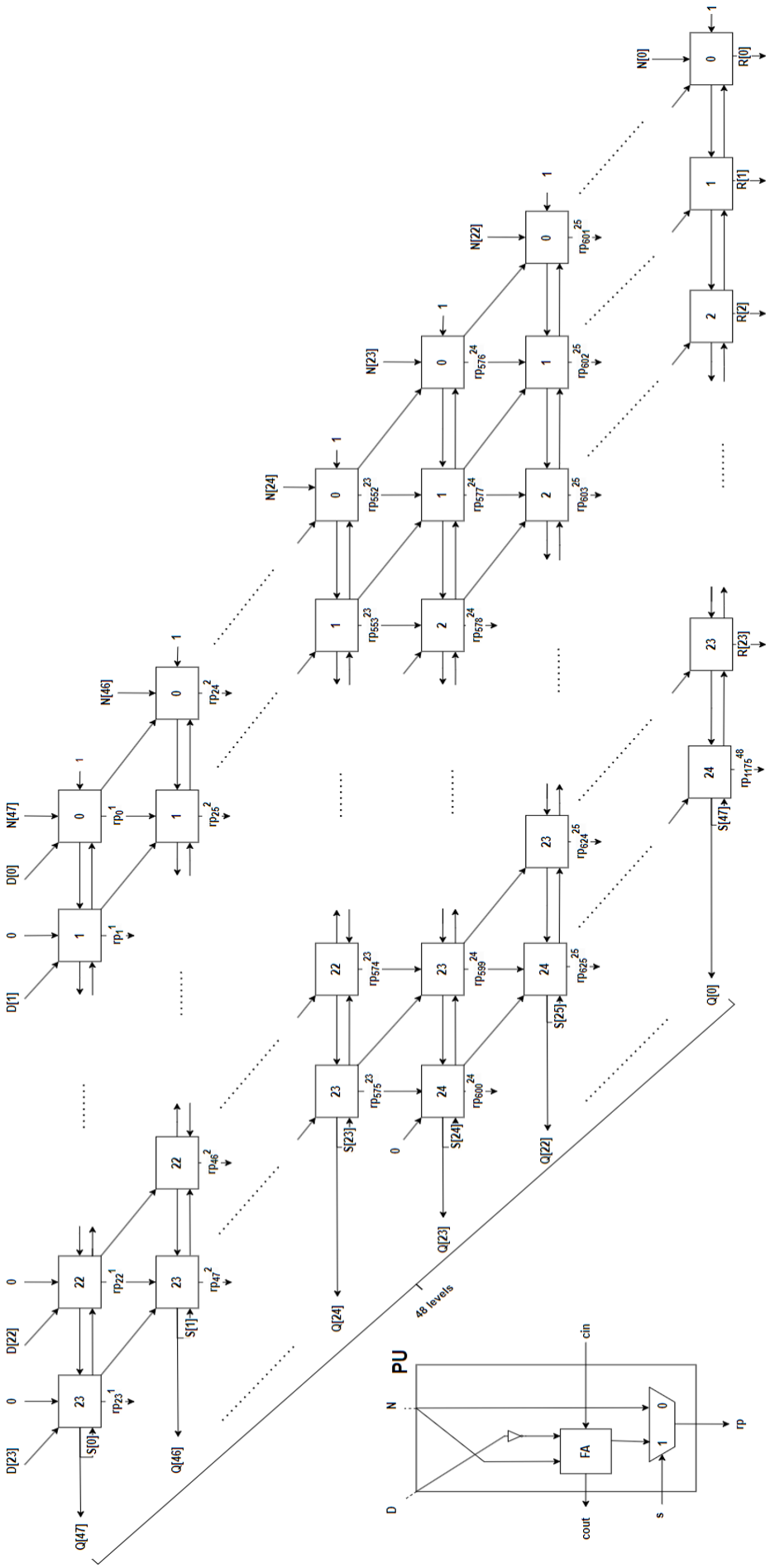

Figura 4.8: Divisor de mantisas de 48/24 bits.

CAPÍTULO 5
Verificación y Resultados

En este capítulo se llevará a cabo una verificación de los diseños de los operadores desarrollados. También se presentará un análisis de su ocupación en LUT (*Look-Up Tables*), el retardo en ciclos de reloj y se explicará el razonamiento detrás de la segmentación de cada operador.

A su vez, se mostrará el funcionamiento de los operadores mediante una serie de ejemplos que abarcan diferentes tipos de valores, ya sea números normalizados o denormalizados. Se compararán los resultados con los resultados esperados calculados externamente y se evaluará cómo cada operador maneja las excepciones.

A través de este capítulo, se pretende proporcionar una visión clara y completa del desempeño de los operadores, validando su diseño y asegurando que cumplen con los requisitos en el procesamiento de números en coma flotante.

5.1 Criterios de Segmentación

Los criterios de segmentación son un aspecto de gran importancia compartido por todos los operadores, ya que como hemos destacado a lo largo de este trabajo, la segmentación es una característica esencial de nuestra unidad. No solo permite el procesamiento de elementos del vector en cada ciclo, sino que también contribuye de manera significativa a la mejora del rendimiento general del sistema.

En particular, nuestros criterios de segmentación se han basado en un análisis detallado del camino crítico del módulo **Man Add** de nuestro sumador de coma flotante. Hemos seleccionado este módulo como referencia debido a su mayor latencia en comparación con otros módulos. Esta elección estratégica nos ha proporcionado una guía clara y coherente durante todo el proyecto, permitiéndonos identificar y optimizar los puntos críticos que podrían afectar el rendimiento general de la unidad.

En los esquemas relacionados con cada operador (4.2 para el sumador, 4.5 para el multiplicador, 4.6 para el FMA y 4.7 para el divisor), se pueden observar las etapas en la que se ha segmentado delimitadas por una línea discontinua de

puntos. Estas etapas tienen una latencia aproximada de 7 ns, lo que equivale a una frecuencia de operación de aproximadamente 142 MHz.

5.2 Sumador/Restador

El esquema completo del sumador/restador es el de la figura 4.2. En este podemos ver claramente como está segmentado en 9 etapas cumpliendo con los criterios de segmentación.

5.2.1. Verificación del Funcionamiento

En primer lugar, vamos a comprobar que el sumador ofrezca resultados correctos enfrente a las diferentes situaciones a las que se puede enfrentar este tipo de operador, diferenciando claramente entre operaciones con números normalizados y denormalizados o excepciones.

Operador A	Operador B	Operación	Resultado Esperado	Resultado
43.3856	54.8523	+	98.2379	98.237885
43.3856	54.8523	-	-11.466698	-11.466698
236.54	-47.58	+	188.959	188.95879
236.54	-47.58	-	284.12	284.1245
58.0	58.0	-	0.0	0.0
-5.65	-8.65	-	2.9999995	2.9999995

Tabla 5.1: Sumas/restas con números normalizados.

En la tabla 5.1 podemos observar una variedad de ejemplos de operaciones con números normalizados. Viendo como resuelve correctamente las operaciones con diferentes signos, o restas de dos números iguales.

Operador A	Operador B	Operación	Resultado Esperado	Resultado
43.3856	1.41E-39	+	43.3856	43.3856
43.3856	1.41E-39	-	43.3856	43.3856
4.18E-39	3.43E-39	+	7.61E-39	7.61E-39
4.18E-39	3.43E-39	-	7.51E-40	7.51E-40
2.306E38	2.19E38	+	NAN.0	NAN.0
INF	-8.65	-	NAN.0	NAN.0

Tabla 5.2: Sumas/restas con números denormalizados y excepciones.

En la tabla 5.2 podemos ver casos con números denormalizados y excepciones. Si la observamos detenidamente podemos observar como el operador soporta tanto las sumas y restas entre números denormalizados como los casos en el que se genera alguna excepción.

5.2.2. Verificación de la Segmentación

A continuación, comprobaremos que el operador cumpla con los criterios de segmentación y sea capaz de realizar operaciones encadenadas con cada elemento del vector. En la figura 5.5 podemos observar como se ha realizado la operación $[1,1,1,1,1,1] + [1,2,3,4,5,6]$ y los resultados han empezado a ser mostrados a partir del noveno ciclo,uno en cada posterior ciclo de reloj. Con estos resultados podemos afirmar que el operador está segmentado correctamente y realiza operaciones de forma encadenada.

5.2.3. Análisis de Retardos y Ocupación

Por último, vamos a analizar los retardos de los módulos y su ocupación en LUT dentro de la FPGA.

Módulo	Tamaño en LUT	Retardo en ns
Exp check	6	4.646
Exp Sub	66	6.651
Sign Swap	1	4.253
Exponent Swap	4	4.253
Significand Swap	1	4.253
Control	3	4.251
Right Shifter	83	5.972
Man Adder	113	6.876
Priority Encoder	33	5.955
Barrel Shifter	53	5.430
Pre Rounder	40	4.667
Rounder	80	6.531
Exception Handler	36	4.703

Tabla 5.3: Tabla de LUT y retardos del sumador.

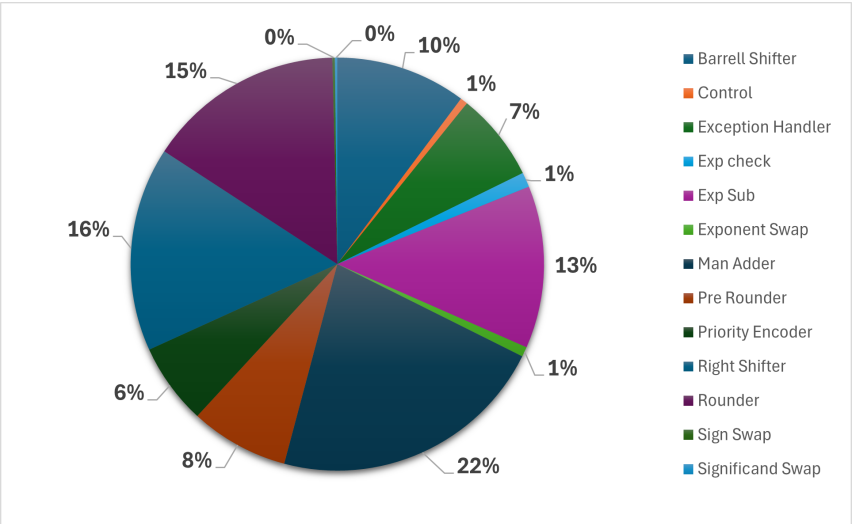

Figura 5.1: Porcentaje de ocupación de cada módulo del sumador/restador.

En la tabla 5.3 y en la figura 5.1 podemos observar la ocupación en LUT y el retardo que ofrece el camino crítico de cada módulo que compone el operador. Como ibamos avanzando, el sumador de mantisas es el módulo que más recursos utiliza y más retardo tiene respecto a todos los otros.

5.3 Multiplicador

El esquema completo del multiplicador se puede observar en la figura 4.5, en este podemos ver como el operador está dividido en 4 etapas.

5.3.1. Verificación del Funcionamiento

Lo primero que debemos hacer es comprobar que el operador realice operaciones correctamente en la variedad de situaciones a las que se puede enfrentar en su uso, diferenciando entre operaciones con números denormalizados y denormalizados o excepciones.

Operador A	Operador B	Resultado Esperado	Resultado
6.35	4.58	29.082998	29.082998
6.35	-4.58	-29.082998	-29.082998
-6.35	-4.58	29.082998	29.082998
89.54	0.0	0.0	0.0
30.25	0.23	6.9575	6.957452

Tabla 5.4: Multiplicaciones con números normalizados.

En la tabla 5.4 podemos observar multiplicaciones de números normalizados, y como los resultados coinciden con los resultados esperados para cada operación. Soportando las multiplicaciones por 0.0.

Operador A	Operador B	Resultado Esperado	Resultado
1.04E-38	9.87E-39	0.0	0.0
58.59	9.87E-39	5.78E-37	0.0
-6.84E-39	8.56	-5.86E-38	0.0
INF	5.0	NAN.0	NAN.0
3.40E38	1.47E11	NAN.0	NAN.0

Tabla 5.5: Multiplicaciones con números denormalizados y excepciones.

Si observamos la tabla 5.5, podemos ver multiplicaciones con números denormalizados, mixtos y con excepciones. Hay que destacar el caso de la segunda y tercera fila de la tabla, en el que, no cumple con el resultado esperado ya que como mencionamos en el apartado 4.3 del capítulo 4, en el caso de multiplicaciones con números mixtos y denormalizados, nuestro operador redondearía el resultado a 0.0 por la proximidad al mismo.

5.3.2. Verificación de la Segmentación

A continuación, comprobaremos que el multiplicador cumpla con los criterios de segmentación y sea capaz de realizar operaciones encadenadas con cada elemento del vector. En la figura 5.6 podemos observar como se ha realizado la operación $[2,2,2,2,2,2] \times [5,6,7,8,9,10]$ y los resultados han empezado a ser mostrados a partir del cuarto ciclo, uno en cada posterior ciclo de reloj. Concluyendo que la segmentación ha sido exitosa y el operador produce resultados de forma encadenada.

5.3.3. Análisis de Retardos y Ocupación

Para terminar con la verificación, vamos a analizar los retardos de los módulos y su ocupación en LUT.

Módulo	Tamaño en LUT	Retardo en ns
genpp	362	4.457
Exp Adder	17	4.887
Bias Sub	28	5.423
Wallace Tree FP32	1261	6.565
Wallace Tree TF32	245	5.249
Wallace Tree BF16	115	4.689
CLA	135	5.972
Control	2	3.754
Result Handler	211	6.528

Tabla 5.6: Tabla de LUT y retardos del multiplicador.

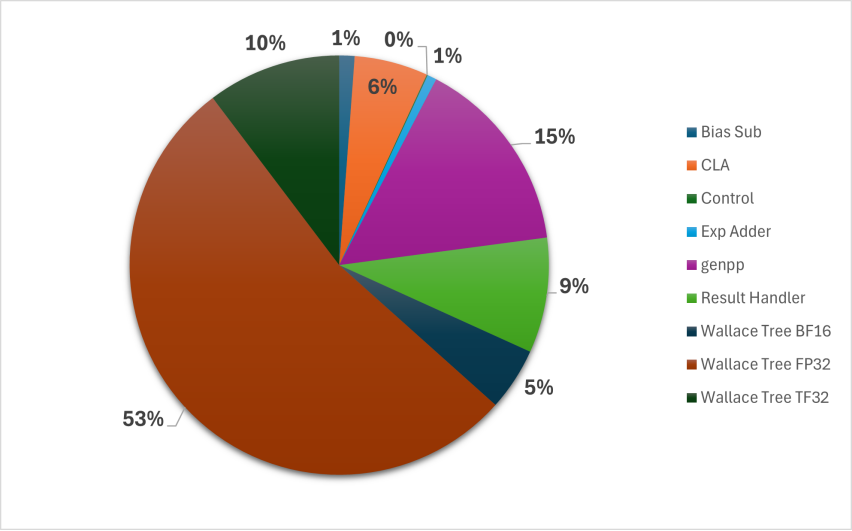

Figura 5.2: Porcentaje de ocupación de cada módulo del multiplicador.

En la tabla 5.6 se detallan la ocupación y los retardos de los caminos críticos de cada módulo que compone el operador de multiplicación. Como anticipamos en la sección 4.3, y se puede ver claramente en la figura 5.2, el árbol de Wallace para FP32 es el módulo con mayor ocupación y retardo.

Además, como se mencionó en el capítulo 2, se evidencia la utilidad de los formatos FP32 y BF16. Los árboles de Wallace para estos formatos presentan una ocupación significativamente menor en LUT y menores retardos.

5.4 Multiplicador-Sumador Fusionados

El esquema de este operador se puede observar en la figura 4.6, en esta podemos ver que está segmentado en 10 etapas cumpliendo con los criterios de segmentación.

5.4.1. Verificación del Funcionamiento

En esta sección comprobaremos que el operador realice las operaciones de forma correcta. Para ello realizaremos distintas operaciones diferenciando claramente entre operaciones con números normalizados y denormalizados o con excepciones.

Operador A	Operador B	Operador C	Operación	Resultado Esperado	Resultado
5.654	4.567	76.564	+	355.32178	355.321776
5.654	4.567	76.564	-	-344.0138	-344.01378
-8.5	150.6	0.1	-	-23.560001	-23.56
50.0	5.0	10.0	-	0.0	0.0
-65.45	-78.6	-34.67	+	2659.6118	2659.6118

Tabla 5.7: FMA con números normalizados.

Como podemos ver en la tabla 5.7, el operador produce los resultados esperados con números normalizados, consigue calcular correctamente los signos y el caso de resta con números iguales.

Operador A	Operador B	Operador C	Operación	Resultado Esperado	Resultado
5.654	4.567	0.0	+	5.654	5.654
5.654	7E25	8E25	-	NAN.0	NAN.0
-8.5	150.6	6E-25	-	-8.5	-8.5
INF	5.0	10.0	-	NAN.0	NAN.0

Tabla 5.8: FMA con números denormalizados o excepciones.

En la tabla 5.8 podemos observar diferentes casos de excepciones o operaciones con números denormalizados soportados por el operador. Si las observamos,

podemos ver como produce un resultado correcto cuando se multiplica por 0 o por un número denormalizado, ya que este multiplicador trata las excepciones de la misma forma que el operador de multiplicación (4.3).

Por otro lado, podemos ver que cuando operamos con infinito, el operador saca como resultado la excepción NAN.0, siendo el resultado correcto.

5.4.2. Verificación de la Segmentación

En la figura 5.7, podemos observar 3 formas de onda distinas. Estas hacen referencia al periodo de reloj, al resultado de la operación y a un contador de ciclos situado para mejorar la interpretación. Para su creación hemos realizado la FMA de 3 vectores, $A = [5,6,7,8,9,10]$ + $B = [5,5,5,5,5,5]$ $\times C = [5,6,7,8,9,10]$. Si analizamos la forma de onda del resultado, podemos ver que a partir del décimo ciclo, los resultados empiezan a mostrarse de forma encadenada, generando un resultado por ciclo de reloj.

Esta figura demuestra que el operador está correctamente segmentado y es capaz de realizar operaciones FMA vectoriales.

5.4.3. Análisis de Retardos y Ocupación

En esta última sección, vamos a analizar los retardos de los módulos y su ocupación en LUT dentro de una FPGA.

Módulo	Tamaño en LUT	Retardo en ns
Exp Handler	23	4.900
GppF	318	4.449
Subtracter	26	5.105
Wt FP32	1261	6.565
Wt TF32	245	5.249
Wt BF16	115	4.689
CLA	135	5.972
Exponent Swap	4	3.762
Control F	5	3.762
Shifter F	212	5.008
Man Compare	41	4.780
Add Sub F	321	6.986
Priority Encoder F	64	5.465
Barrel Shifter F	132	5.595
Pre Rounder F	39	4.212
Rounder F	82	5.980
Exception Handler F	38	4.274

Tabla 5.9: Tabla de LUT y retardos del FMA.

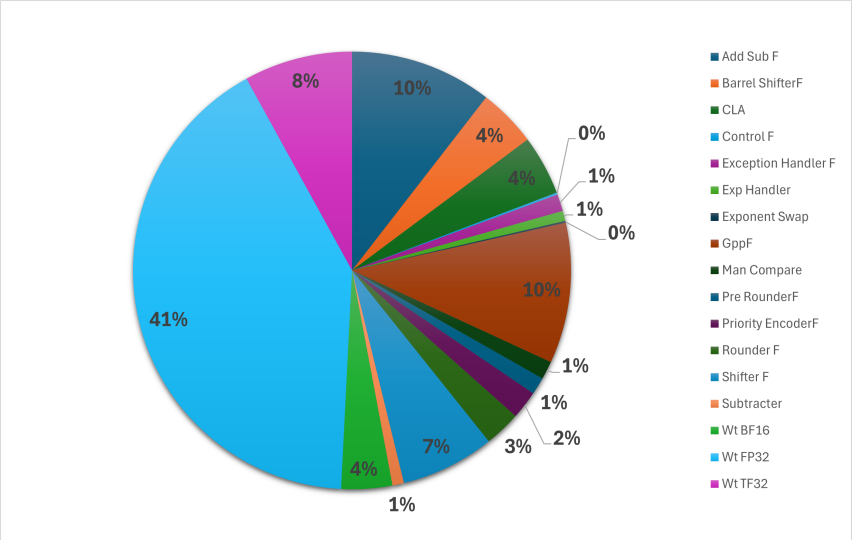

Figura 5.3: Porcentaje de ocupación de cada módulo del FMA.

En la tabla 5.9 y en la figura 5.3 podemos observar la ocupación de cada módulo y su retardo. Podemos destacar que el árbol de Wallace para FP32 representa en LUT más del 40 % del operador, y que el módulo que más retardo tiene es el sumador de mantisas.

5.5 Divisor

El esquema completo del divisor se puede observar en la figura 4.7. Donde podemos ver que en total, el divisor está segmentado en 24 etapas distribuidas para cumplir con los criterios de segmentación mencionados en la sección 5.1.

5.5.1. Verificación del Funcionamiento

Lo primero que debemos comprobar es que el divisor ofrezca resultados correctos enfrente a las diferentes situaciones a las que se puede enfrentar este tipo de operador, diferenciando claramente entre operaciones con números normalizados y denormalizados o excepciones.

Operador A	Operador B	Resultado Esperado	Resultado
57.65	4.57	12.61488	12.614879
57.65	-4.57	-12.61488	-12.614879
57.65	0.58	99.39656	99.396492
8.5	0.0	NAN.0	NAN.0
0.53	4.56	0.11622	0.116218
0.0	10.25	0.0	0.0

Tabla 5.10: Divisiones con números normalizados.

En la tabla 5.10 podemos observar diferentes ejemplos de operaciones con números normalizados. A su vez, se puede observar como el divisor realiza correctamente las divisiones con esta clase de valores.

Operador A	Operador B	Resultado Esperado	Resultado
7E-45	4.203E-45	1.666666	1.666666
101.654	4.203E-45	NAN.0	NAN.0
2.10E-44	67564.0	0.0	0.0
8.5	0.0	NAN.0	NAN.0
INF	25.65	NAN.0	NAN.0
5.63	INF	NAN.0	NAN.0

Tabla 5.11: Divisiones con números denormalizados y excepciones.

En la tabla 5.11 podemos ver diferentes ejemplos de operaciones con números denormalizados, mixtos o excepciones viendo claramente como son tratadas cada tipo de excepción de forma correcta por parte del operador.

5.5.2. Verificación de la Segmentación

En la figura 5.8, podemos apreciar tres formas de onda distintas relacionadas con un caso de prueba del operador. La primera es una muestra del reloj del sistema, la segunda muestra el resultado en coma flotante de 32 bits y la tercera muestra un contador de los ciclos de reloj.

Para su creación hemos realizado la división del vector $A = [10, 12, 14, 16, 18, 20]$ por el vector $B = [2, 2, 2, 2, 2, 2]$. Al analizar esta figura, observamos que el operador produce el primer resultado después de una latencia de 24 ciclos. Posteriormente, el operador es capaz de generar un resultado por ciclo de reloj, encadenando todos los elementos del vector sin interrupción. Esta figura demuestra claramente que la segmentación del operador se ha completado de forma exitosa.

5.5.3. Análisis de Retardos y Ocupación

Por último vamos a analizar los retardos de los módulos y su ocupación en LUT dentro de la FPGA.

Módulo	Tamaño en LUTs	Retardo en ns
Exp Sub	28	4.743
Bias Add	10	4.430
Control	3	3.754
Man Div	1177	84.426
Res Handler	37	4.22

Tabla 5.12: Tabla de LUT y retardos del divisor.

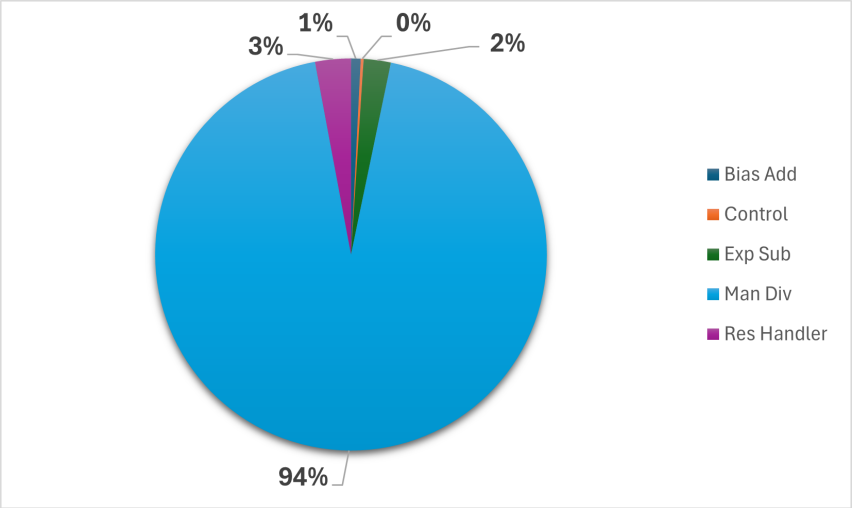

Figura 5.4: Porcentaje de ocupación de cada módulo del divisor.

Como podemos ver en la tabla 5.12 y en la figura 5.4, el divisor de mantisas es el módulo que, por diferencia, nos provoca un mayor retardo y una mayor ocupación como habíamos comentado en el capítulo 4. Por ese motivo, este módulo internamente está segmentado en 24 etapas consiguiendo llegar a la frecuencia objetivo.

5.6 Bucle SAXPY

En esta sección, llevaremos a cabo la prueba de concepto final en la que la unidad vectorial ejecutará un bucle SAXPY (*Single-Precision A·X + Y*). Es una operación fundamental en el cálculo numérico y en la computación científica. Se utiliza para calcular una combinación lineal de vectores, multiplicando un vector X por un escalar a a y sumando el resultado a otro vector Y.

El uso de SAXPY para esta prueba de concepto es ideal porque involucra tanto operaciones de multiplicación como de suma, permitiendo así verificar la funcionalidad completa de la unidad vectorial. Esto nos permitirá confirmar que los operadores y la unidad de control funcionan en perfecta sintonía y producen los resultados esperados.

```
1 vsetvl (rd), aux12, (rs2) # Establecer VLR
2 vld v1, aux9              # Cargar vector X con stride
3 vmul v3, v1, aux1         # Multiplicar a y X
4 vld v2, aux8              # Cargar vector Y
5 vadd v4,v3,v2             # Sumar vector Y
6 vst v4, aux8              # Guardar resultado
```

Como podemos ver en las figuras 5.9 y 5.10, la prueba ha sido todo un éxito, ya que la unidad vectorial realiza las multiplicaciones encadenadas en todos los elementos del vector (multiplicando a por X) en la figura 5.9. El resultado se carga en otro vector Y, como se puede apreciar en la figura 5.10, se le suma el vector Y de forma encadenada.

Con estos resultados, podemos afirmar que la unidad vectorial está operativa y cumple con las expectativas de funcionamiento. La prueba del bucle SAXPY ha demostrado que los operadores y la unidad de control trabajan en perfecta sincronía, ejecutando las operaciones de multiplicación y suma de manera eficiente y precisa.

Figura 5.5: Forma de onda de suma vectorial.

Figura 5.6: Forma de onda de multiplicación vectorial.

Figura 5.7: Forma de onda de FMA vectorial.

Figura 5.8: Forma de onda de división vectorial.

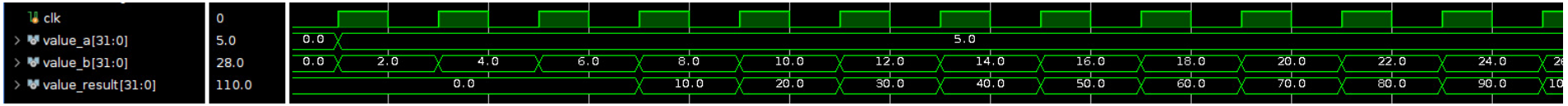

Figura 5.9: Forma de onda para la multiplicación de $a \times \vec{X}$.

Figura 5.10: Forma de onda de $a\vec{X} + \vec{Y}$.

CAPÍTULO 6
Conclusiones

En este capítulo final se presentan las conclusiones de este trabajo, a la vez que la relación del mismo con los estudios cursados y trabajos futuros.

6.1 Conclusiones Finales

6.1.1. Objetivos Iniciales

Respecto a los objetivos iniciales, se puede afirmar que se han cumplido en su totalidad. El diseño de la unidad de coma flotante segmentada se ha llevado a cabo con éxito, implementando la segmentación de manera correcta y asegurando ejecuciones eficientes para operaciones aritméticas como suma/resta, multiplicación, división y FMA.

La precisión y el redondeo adecuado también se han garantizado mediante una investigación de los formatos de redondeo conforme al estándar IEEE 754, proporcionando cuatro modos de redondeo: al par más cercano, truncado, a $+\infty$ y a $-\infty$. Además, se han implementado correctamente las operaciones con máscaras según RISC-V, dotando a la unidad de la capacidad de ejecutar operaciones condicionales.

En cuanto al rendimiento y la eficiencia, las pruebas realizadas han demostrado que la unidad ofrece un rendimiento adecuado, siendo capaz de ejecutar las operaciones de manera eficiente. La flexibilidad de implementación también se ha logrado, ya que el diseño permite que la unidad sea implementada tanto en dispositivos FPGA como en ASIC, lo que asegura su adaptabilidad a diferentes entornos de aplicación.

Asimismo, se ha ampliado el soporte a los formatos de datos utilizados en redes neuronales, integrando con éxito los formatos TF32 y BFloat16 en el sumador/-restador, el multiplicador y el operador FMA, lo cual se alinea con las necesidades para el entrenamiento de redes neuronales y la inferencia de datos. Finalmente, la integración exitosa de la unidad en una unidad vectorial, como se demostró en la verificación de la sección 5.6, confirma su funcionamiento armonioso con la parte de control desarrollada por mi compañero D.Chinesta.

6.1.2. Dificultades sobre el Trabajo Realizado

Durante el desarrollo de este trabajo, uno de los mayores desafíos fue encontrar información de calidad y detallada para implementar correctamente los operadores de la unidad de coma flotante. La mayoría de las fuentes disponibles proporcionaban solo descripciones superficiales o implementaciones genéricas, lo que complicó el desarrollo de aspectos avanzados como el redondeo preciso y el soporte para números denormalizados.

Este desafío subrayó la importancia de saber identificar y recopilar información útil de manera eficiente, una habilidad que se ha mejorado significativamente a lo largo del proyecto. Sin embargo, la dificultad en encontrar recursos adecuados sigue siendo una de las grandes limitaciones enfrentadas.

6.1.3. Aprendizajes y Desarrollo Profesional

A lo largo de este proyecto, se han desarrollado diversas competencias importantes. Se ha mejorado la capacidad para comunicarme de manera efectiva y técnica. Siendo esto esencial para las reuniones con los tutores y con el compañero. También se ha aprendido a recopilar y evaluar información de calidad en menos tiempo, un aspecto necesario en la investigación y desarrollo de proyectos complejos. Además de adquirir un dominio más profundo del entorno de desarrollo Xilinx Vivado y del lenguaje de descripción de hardware Verilog, habilidades fundamentales para crear un buen futuro profesional en el campo del diseño de sistemas digitales.

6.2 Relación con los Estudios Cursados

En esta sección expondremos los conocimientos adquiridos a lo largo de los estudios en el grado en Ingeniería Informática que han sido de ayuda para el correcto desarrollo de este trabajo. Tanto en asignaturas cursadas como en competencias transversales obtenidas a lo largo del grado.

6.2.1. Asignaturas

En este trabajo de final de grado, han sido de gran utilidad los conocimientos adquiridos en diversas asignaturas del plan de estudios. Entre las más relevantes se encuentran:

- **Diseño de Sistemas Digitales (DSD)**: Esta asignatura resultó de gran ayuda para el trabajo, ya que proporcionó un sólido conocimiento previo del entorno de desarrollo Xilinx Vivado, incluyendo sus funcionalidades para simulación y síntesis de componentes. Además, la familiaridad con VHDL facilitó la transición hacia Verilog, que ha sido el lenguaje de descripción de hardware usado en este proyecto.

- **Arquitecturas Avanzadas (AAV)**: A través de esta asignatura, se adquirieron conocimientos sobre las etapas de segmentación, que fueron aplicados en el proceso utilizado en todos los operadores del proyecto. También se obtuvo una buena comprensión sobre el funcionamiento de las tablas LUT, fundamentales en las FPGA para entender los retardos y la utilización de los recursos del operador sintetizado por la herramienta de desarrollo.
- **Estructura de Computadores (ETC)**:En esta asignatura fue de gran ayuda para entender desde las bases la arquitectura de computadores. Entre todos los conceptos dados en esta asignatura, aquel que más ha ayudado en este trabajo ha sido la unidad didáctica sobre aritmética de coma flotante. Ya que en esta, se comentaron muchos conceptos imprescindibles para trabajar con datos de coma flotante, como por ejemplo como está estructurado el valor según el estándar IEEE 754.

6.2.2. Competencias Transversales

Es imprescindible destacar las siguientes competencias transversales que se han requerido y se han puesto en práctica para el correcto desarrollo de este proyecto:

- **CT-2. Innovación y Creatividad**: El proyecto se ha centrado en la implementación de una unidad vectorial de coma flotante segmentada dentro del contexto de la arquitectura RISC-V y las unidades vectoriales, ambas innovaciones significativas en el campo de la arquitectura de computadores. Partiendo de conocimientos muy básicos, se han desarrollado soluciones creativas para diseñar y segmentar la unidad de coma flotante, contribuyendo al avance en el ámbito del diseño de sistemas digitales y la arquitectura de computadores.
- **CT-3. Trabajo en equipo y liderazgo**: Este proyecto forma parte de un esfuerzo mayor para crear una unidad vectorial basada en RISC-V. Este trabajo se llevó a cabo en colaboración con otro compañero, lo que requirió habilidades de trabajo en equipo. Intercambiamos ideas y trabajamos en armonía para asegurar el correcto funcionamiento de la unidad vectorial. Asumimos responsabilidades específicas y contribuimos colectivamente a la mejora y desarrollo del proyecto, demostrando nuestra capacidad para colaborar eficazmente en un entorno de equipo y liderar cuando fue necesario.
- **CT-4. Comunicación efectiva**: En este proyecto, la comunicación efectiva ha sido crucial tanto en las reuniones con los tutores como con el compañero. Mantener una comunicación clara y precisa, compartir ideas y solicitar retroalimentación ha sido indispensable para el correcto desarrollo del TFG. Además, la comunicación constante nos permitió identificar y resolver problemas rápidamente, asegurando una colaboración fluida y eficiente. Participar en discusiones técnicas y presentaciones también fortaleció nuestras habilidades para transmitir conceptos complejos de manera comprensible, facilitando la toma de decisiones informadas y consensuadas en todo momento.

- **CT-5. Responsabilidad y toma de decisiones**: En el desarrollo de este TFG, esta competencia ha sido esencial. Gran parte de la información necesaria para el desarrollo del proyecto ha sido aprendida de forma autónoma, mediante la búsqueda en diversas fuentes como internet y libros especializados en el tema. Cada decisión se tomó después de contrastar múltiples fuentes y opciones, y se justificó claramente el motivo de cada elección. Durante el proyecto, se realizaron numerosas pruebas y experimentos, eligiendo siempre la mejor opción basada en justificaciones empíricas y análisis detallados, asegurando así la solidez del trabajo.

6.3 Trabajos Futuros

A continuación, se presentan algunas propuestas para la continuación y mejora del trabajo desarrollado en este proyecto:

- **Añadir soporte para más operaciones**: Ampliar la funcionalidad de la unidad de coma flotante para incluir operaciones como la raíz cuadrada, la cual podría desarrollarse utilizando aproximaciones de Newton-Raphson, así como la operación de comparación y la operación de recíproco.
- **Ampliar el soporte de más formatos de datos**: Actualmente, el operador de *fused multiply-add*, el multiplicador y el divisor no han sido completamente parametrizados debido a limitaciones de tiempo y conocimiento. En el futuro, sería beneficioso parametrizar estos operadores para soportar una mayor variedad de formatos de datos. Dado que estas estructuras son más complejas que el sumador/restador, este esfuerzo requeriría una mayor investigación y desarrollo.
- **Soporte exacto y preciso a números denormalizados y excepciones**: Los números denormalizados presentan un desafío significativo en el desarrollo de unidades de coma flotante. Se ha realizado un esfuerzo notable para considerar todos los posibles casos, pero, dadas las limitaciones de tiempo y conocimiento a las que se han enfrentado en este trabajo de final de grado. Soportar el gran número de casos particulares que pueden surgir demostró ser particularmente desafiante.

  Ofrecer un soporte exacto y preciso para estos valores mejoraría significativamente la fiabilidad y precisión de la unidad. Esto requerirá un enfoque detallado y riguroso para gestionar adecuadamente los casos excepcionales que estos números presentan, incluyendo la implementación de algoritmos y técnicas avanzadas que aseguren una correcta manipulación y procesamiento de los mismos.

Bibliografía

[1] *BFloat16: The secret to high performance on Cloud TPUs*. Google Cloud Blog. URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus (visitado 26-06-2024).

[2] Michael Flynn. "Flynn's Taxonomy". En: *Encyclopedia of Parallel Computing*. Ed. por David Padua. Boston, MA: Springer US, 2011, págs. 689-697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2. URL: https://doi.org/10.1007/978-0-387-09766-4_2.

[3] John L. Hennessy y David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5.ª ed. Amsterdam: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[4] "IEEE Standard for Floating-Point Arithmetic". En: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (jul. de 2019), págs. 1-84. DOI: 10.1109/IEEESTD.2019.8766229.

[5] "IEEE Standard for Verilog Hardware Description Language". En: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), págs. 1-590. DOI: 10.1109/IEEESTD.2006.99495.

[6] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". En: 2017. URL: https://arxiv.org/pdf/1704.04760.pdf.

[7] Baljinder Kaur y Vipasha Thakur. "Review of Booth Algorithm for Design of Multiplier". En: 2014. URL: https://api.semanticscholar.org/CorpusID:17135558.

[8] Swaroop Kumar. "FPGA Implementation of Fixed point Integer Divider Using Iterative Array Structure". En: 2015. URL: https://www.semanticscholar.org/paper/FPGA-Implementation-of-Fixed-point-Integer-Divider-Kumar/b384aa873eb84a8da74870ae313b765061774522.

[9] Haichengw Says. *Accelerating AI Training with NVIDIA TF32 Tensor Cores*. NVIDIA Technical Blog. 27 de ene. de 2021. URL: https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/ (visitado 26-06-2024).

[10] Andrew Waterman et al. "The RISC-V Instruction Set Manual". En: 2014. URL: https://api.semanticscholar.org/CorpusID:61619035.

[11] Kun-Yi Wu et al. "Multiple-mode floating-point multiply-add fused unit for trading accuracy with power consumption". En: *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. 2013, págs. 429-435. DOI: 10.1109/ICIS.2013.6607878.

APÉNDICE A
Objetivos de Desarrollo Sostenible

En la siguiente tabla (A.1) se mostrará el grado de relación de nuestro trabajo con los Objetivos de Desarrollo Sostenible(ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**	X			
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**	X			
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Figura A.1: Tabla de ODS

Este trabajo de final de grado (TFG) presentado tiene una relación directa con varios Objetivos de Desarrollo Sostenible (ODS), alineándose con las metas globales establecidas por las Naciones Unidas para fomentar un futuro más sostenible. A continuación, se expone cómo este proyecto contribuye a los objetivos específicos de Educación de calidad, Industria, innovación e infraestructuras, Producción y consumo responsables, y Acción por el clima.

- **Educación de calidad**: El ODS 4, Educación de calidad, tiene como meta garantizar una educación inclusiva, equitativa y de calidad, promoviendo oportunidades de aprendizaje durante toda la vida para todos. Este TFG, al centrarse en el desarrollo y explicación detallada de la arquitectura y funcionamiento de diferentes operadores de coma flotante, ofrece un recurso valioso para la enseñanza en campos de la informática y la ingeniería.

  La documentación y el análisis de los mismos, proporcionan un material didáctico que puede ser utilizado tanto en el ámbito académico como en la formación profesional. Al desglosar conceptos complejos en partes comprensibles y demostrables, se facilita el aprendizaje y comprensión de tecnologías avanzadas, beneficiando a estudiantes y profesionales interesados en el diseño de unidades de procesamiento.

- **Industria, innovación e infraestructuras**: El ODS 9, Industria, innovación e infraestructuras, promueve la construcción de infraestructuras resilientes, la promoción de una industrialización inclusiva y sostenible, y el fomento de la innovación. Este TFG se alinea con este objetivo mediante el desarrollo de una unidad de coma flotante que es parte de una unidad vectorial, destacando su relevancia e innovación respecto a las unidades básicas tradicionales.

  Las unidades vectoriales representan un avance significativo en el procesamiento de datos, ofreciendo mejoras en rendimiento y eficiencia que son cruciales para aplicaciones de alta demanda computacional. La investigación y desarrollo en este campo no solo impulsa la innovación tecnológica sino que también potencia la capacidad industrial para producir soluciones avanzadas, apoyando así el crecimiento económico y la modernización de infraestructuras tecnológicas.

- **Producción y consumo Responsables**: El ODS 12, Producción y consumo responsables, busca garantizar modalidades de consumo y producción sostenibles. La elección de realizar pruebas y desarrollos sobre plataformas FPGA se alinea con este objetivo, ya que las FPGA son dispositivos reprogramables que permiten un uso eficiente y sostenible de los recursos.

  A diferencia de los circuitos integrados específicos que, una vez fabricados, no pueden ser modificados, las FPGA pueden ser reconfiguradas múltiples veces para distintos propósitos. Esta característica reduce la necesidad de producir múltiples chips para diferentes aplicaciones, minimizando el desperdicio y la huella ambiental asociada con la fabricación de hardware. Además, el entorno de desarrollo Vivado facilita la simulación y prueba de los

circuitos antes de su implementación final, permitiendo ajustes y optimizaciones sin necesidad de recurrir a múltiples iteraciones de fabricación.

- **Acción por el clima**: El ODS 13, Acción por el clima, tiene como objetivo adoptar medidas urgentes para combatir el cambio climático y sus efectos. El enfoque en el uso de FPGA y herramientas de simulación como Vivado, en lugar de la fabricación de chips específicos y de un solo uso, contribuye significativamente a la sostenibilidad ambiental.

  Al utilizar FPGA reprogramables, se disminuye la demanda de recursos y energía necesarios para la producción de nuevos componentes, lo que a su vez reduce la emisión de gases de efecto invernadero y otros contaminantes asociados con la fabricación de hardware. La capacidad de realizar múltiples pruebas y simulaciones en un entorno controlado antes de la implementación física también optimiza el uso de materiales y reduce el impacto ambiental, apoyando así las iniciativas para mitigar el cambio climático.