UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería de Computadores

Añadiendo funcionalidades a un procesador RISC-V basado en el proyecto RVFPGA

Adding functionalities to a RISC-V CPU based on the RVFPGA project

Realizado por
Alfonso Martínez Conejo

Tutorizado por
Julio Villalba Moreno

Departamento
Arquitectura de Computadores

MÁLAGA, JUNIO DE 2024

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DE COMPUTADORES

AÑADIENDO FUNCIONALIDADES A UN PROCESADOR RISC-V BASADO EN EL PROYECTO RVFPGA

ADDING FUNCTIONALITIES TO A RISC-V CPU BASED ON THE RVFPGA PROJECT

Realizado por
ALFONSO MARTÍNEZ CONEJO

Tutorizado por
JULIO VILLALBA MORENO

Departamento
ARQUITECTURA DE COMPUTADORES

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2024

Fecha defensa: julio de 2024

Abstract

Due to the recent microchip shortage produced after the pandemic, in Europe, the need for having its own technology has emerged, caused by the huge importance of these types of components and by the fact that nowadays, both manufacturing and design of microchips is mainly performed in the United States and China. That's where RISC-V comes into relevance: an open source hardware architecture, but whose development and implementation are still on early stages. The aim of this thesis is to study the functioning of the RISC-V architecture, in order to subsequently implement new functionalities to a CPU of such architecture, in the form of support for bit manipulation instructions, a new hardware counter and, especially, enabling floating point operations, which will be done using a new representation format called HUB (Half Unit Biased). To be able to access a CPU and modify it, the RVfpga project, developed by the Imagination University Programme, has been utilized.

Keywords: RISC-V, Floating Point, FPGA, HUB format, ISA

Resumen

Con motivo de la reciente escasez de microchips producida tras la pandemia, en Europa ha surgido la necesidad de contar con una tecnología propia, debido a la suma importancia de este tipo de componentes y a que actualmente tanto el diseño como la producción de microchips se encuentra focalizada en Estados Unidos y China. Es ahí donde cobra importancia RISC-V, una arquitectura de hardware libre, pero cuyo desarrollo e implementación se encuentran aún en una fase algo temprana. El objetivo de este trabajo de fin de grado es estudiar el funcionamiento de la arquitectura RISC-V, para posteriormente añadir nuevas funcionalidades a un procesador de dicha arquitectura, en la forma de instrucciones de manipulación de bits, un contador de eventos hardware y, especialmente, añadiendo soporte para realizar operaciones en punto flotante, lo cual se hará empleando un nuevo formato de representación llamado HUB (Half Unit Biased). Para poder acceder a un procesador y modificarlo, se ha accedido al proyecto RVfpga, desarrollado por Imagination University Programme.

Palabras clave: **RISC-V, Punto Flotante, FPGA, formato HUB, ISA**

Índice

1 Introducción

1.1. Motivación

Como resultado de la reciente escasez de microchips, en Europa ha comenzado a surgir la necesidad de contar con una tecnología propia, ya que, a día de hoy, la gran mayoría de los microchips se diseñan y fabrican en empresas privadas en el extranjero. Concretamente, en lugares como Estados Unidos, China y Taiwán. En esta última región se encuentra la mayor fundición de semiconductores del mundo [1].

Al ser lugares que se encuentran alejados de Europa y que pueden ser fuente de futuros conflictos, se están comenzando a barajar distintas alternativas para lograr una mayor independencia en este sector, y es que los microchips son componentes de vital importancia para el desarrollo de muchas otras industrias, ya que de ellos dependen tecnologías como el 5G, smartphones, automoción, productos IoT, centros de cómputo, tarjetas gráficas para entrenar IAs, y un largo etcétera.

Es por ello que resulta necesario disponer de una diversa gama de microchips: desde microcontroladores orientados al bajo consumo y a la eficiencia para su integración en sistemas empotrados, hasta procesadores de alto rendimiento utilizados en centros de procesamiento masivo de datos.

Con todo esto en mente, una de las arquitecturas por las que se está apostando con mayor fuerza es RISC-V: una arquitectura de conjunto de instrucciones de hardware libre que tiene el potencial de cumplir con los requisitos mencionados, y cuyos detalles se explicarán más adelante en este documento.

Para poder acceder a un procesador basado en RISC-V y modificarlo, se ha hecho uso del proyecto RVfpga [2], desarrollado por Imagination University Programme. Este contiene los

archivos necesarios para sintetizar el núcleo SweRV EH1 en la FPGA[1] Nexys A7, así como diversas guías y prácticas para familiarizarse con el entorno.

Dentro del campo de la computación, las operaciones en punto flotante, es decir, las operaciones llevadas a cabo con números reales, resultan de vital importancia en aplicaciones científicas, simulaciones, etc. Sin embargo, desde el punto de vista del hardware, representan una de las operaciones más costosas de implementar. Así pues, existe una búsqueda constante por representar este tipo de valores de una forma más eficiente. Una de esas representaciones es el formato HUB[2], que promete una reducción en el área, energía, y en el tiempo de ejecución [3].

1.2. Objetivos

El principal objetivo de este trabajo de fin de grado es incrementar la funcionalidad del procesador del que se parte, principalmente añadiendo soporte para operaciones en punto flotante en formato HUB, así como otro tipo de instrucciones aritmético-lógicas y contadores de eventos hardware, aprendiendo en el proceso las características de la arquitectura RISC-V y cómo se lleva a cabo el desarrollo de la misma. Con el fin de alcanzar dichos objetivos, se han planteado una serie de metas con dificultad incremental:

- Estudio del funcionamiento y características de la arquitectura RISC-V
- Lectura en profundidad de la documentación proporcionada en el proyecto RVfpga
- Implementación de las instrucciones de manipulación de bits Min/Minu/Max/Maxu
- Implementación de un contador hardware de instrucciones tipo I
- Adaptación del pipe del divisor para soportar instrucciones de punto flotante
- Modificación de la unidad de FP para soportar el formato HUB
- Integración de la unidad FP HUB en la FPGA

[1] Field-Programmable Gate Array
[2] Half-Unit Biased

1.3. Estructura del documento

A continuación se presentan las distintas secciones en las que se encuentra dividida esta memoria.

- **Sección 1: Introducción.** En esta sección se ha expuesto el contexto y motivación que han llevado al desarrollo de este trabajo de fin de grado, así como los objetivos que se plantean en el mismo.
- **Sección 2: Estado del arte.** Aquí se explican tanto las distintas tecnologías utilizadas, como las herramientas software y hardware que han sido necesarias durante el transcurso del proyecto, con un especial énfasis en la arquitectura RISC-V y el funcionamiento del procesador SweRV-EH1.
- **Sección 3: Implementación.** En este apartado se lleva a cabo la implementación de nuevas funcionalidades en el SweRV-EH1, empezando con instrucciones de manipulación de bits, para posteriormente añadir un contador de eventos hardware y, finalmente, añadir soporte para instrucciones de punto flotante con formato HUB.
- **Sección 4: Testeo y verificación.** Tras finalizar la implementación, se elaboran distintas pruebas para comprobar el correcto funcionamiento del sistema, de manera que las nuevas funciones se ejecuten de forma adecuada y sin alterar el comportamiento del resto del dispositivo.
- **Sección 5: Conclusiones y líneas futuras.** Finalmente, en esta sección se realiza una reflexión sobre el resultado final obtenido y el desarrollo general de este trabajo, además de sugerir una serie de mejoras que podrían llevarse a cabo en próximas iteraciones.
- **Anexo A: Manual de usuario.** En este apartado se incluye una guía sobre la instalación de los programas necesarios, junto con instrucciones para ejecutar el sistema, ya sea mediante simulación o mediante síntesis en la placa FPGA.

2

Estado del arte

2.1. RISC-V

2.1.1. Antecedentes: CISC y RISC

La arquitectura RISC-V es una arquitectura de conjunto de instrucciones (ISA) de hardware libre basada en un diseño de tipo RISC[3]. Tradicionalmente se habla de dos tipos de arquitecturas: las llamadas CISC[4] y las RISC.

Las arquitecturas CISC se caracterizan por contar con un conjunto de instrucciones complejas y avanzadas, permitiendo así realizar un elevado número de operaciones empleando pocas líneas de código ensamblador, asemejándose en cierta manera a un lenguaje de alto nivel. La principal ventaja de CISC es que, debido a esto, el compilador tiene que realizar poco trabajo para traducir un código de alto nivel a instrucciones máquina. Además, como la longitud del código es relativamente pequeña, este ocupa poco espacio en la memoria RAM. A la hora de diseñar la arquitectura, se hace énfasis en implementar instrucciones complejas directamente en el hardware [4].

Sin embargo, esta complejidad es también su principal desventaja, pues los procesadores de esta arquitectura resultan costosos de fabricar y, al constar de más componentes, tienden a consumir más energía. Concretamente, la unidad de control suele ser especialmente intrincada. Pese a ello, esta arquitectura fue la única utilizada hasta finales de la década de 1970, cuando surgió por primera vez el concepto de RISC [5].

Esta surgió como una alternativa a CISC, y se basa en la utilización de instrucciones más sencillas y que necesitan una menor cantidad de tiempo y hardware para ser ejecutadas. Aunque para ello se requiera de más líneas de código y de compiladores más sofisticados, en la

[3] Reduced Instruction Set Computing
[4] Complex Instruction Set Computing

práctica se ha demostrado que los procesadores de esta arquitectura pueden obtener un rendimiento igual o superior a los basados en CISC [4], especialmente los casos en los que el consumo de energía es crítico.

Aunque todavía existen procesadores CISC, estos prácticamente se restringen a arquitecturas basadas en la familia Intel x86 por su amplia implantación a lo largo de décadas. Hoy en día, los procesadores de nuevo diseño se realizan casi exclusivamente con tecnología RISC.

No obstante, algo que comparten la mayoría de arquitecturas modernas es que están protegidas por patentes, limitando así el desarrollo y la innovación que pueden ser llevados a cabo por terceros, de modo que en 2010 la Universidad de California en Berkeley comenzó el proyecto RISC-V [6].

2.1.2. Características de RISC-V

RISC-V es un conjunto de instrucciones libre y abierto, el cual se puede usar sin regalías para cualquier propósito, permitiendo así que cualquiera diseñe, fabrique y venda chips de esta microarquitectura.

Cuando se planteó RISC-V, el objetivo principal era que se convirtiese en un conjunto de instrucciones universal, para ser utilizado por procesadores de todos los ámbitos: desde dispositivos IoT de bajo consumo hasta superordenadores. Con el fin de lograr tal objetivo, se establecieron cinco principios que debía cumplir la arquitectura [2]:

- Debe ser compatible con una amplia gama de paquetes de software y lenguajes de programación.
- Su implementación debe ser plausible mediante todo tipo de tecnologías: desde FPGAs hasta ASICs[5], así como otras tecnologías emergentes.
- Debe ser eficiente en distintos escenarios a nivel de microarquitectura, incluyendo aquellos que implementan microcódigo, pipelines en orden y fuera de orden, distintos tipos de paralelismo, etc.
- Debe ser capaz de adaptarse a tareas específicas para conseguir el rendimiento máximo requerido sin limitaciones impuestas por el propio ISA.
- Su conjunto de instrucciones base debe ser estable y duradero, ofreciendo un framework común y sólido para los desarrolladores.

[5] Application Specific Integrated Circuit

Nombre	Descripción	Estado
RV32I	**Conjunto de instrucciones base con enteros de 32 bits**	**Ratificado**
RV32E	Conjunto de instrucciones base con enteros de 32 bits para sistemas empotrados	Borrador
RV64I	**Conjunto de instrucciones base con enteros de 64 bits**	**Ratificado**
RV128I	Conjunto de instrucciones base con enteros de 128 bits	Borrador

Cuadro 1: ISAs base de RISC-V [7]

Una de las principales características de RISC-V es que cuenta con un ISA modular, en lugar de incremental, dando como resultado una arquitectura flexible y pulcra. Los procesadores implementan el ISA base y sólo las extensiones que son utilizadas. Esta modularidad difiere con las arquitecturas tradicionales, como x86 o ARM, las cuales cuentan con conjuntos de instrucciones incrementales, en los que los ISAs anteriores son expandidos y cada nuevo procesador ha de implementar todas las instrucciones, incluyendo aquellas marcadas como ”obsoletas”, para así asegurar la compatibilidad con programas antiguos. Por ejemplo, x86 comenzó con 80 instrucciones y a día de hoy cuenta con más de 1300 [2]. Este elevado número de instrucciones y la necesidad de retrocompatibilidad desembocan en procesadores que requieren de una gran cantidad de hardware y energía.

RISC-V dispone de cuatro opciones para el ISA base: dos versiones de 32 bits (para números enteros y para sistemas empotrados, RV32I y RV32E), y versiones de 64 y 128 bits (RV64I y RV128I), como se muestra en el Cuadro 1. Los módulos marcados como ”Congelado” no deberían sufrir grandes cambios antes de ser ratificados. En contraste, aquellos marcados como ”Borrador” es probable que sean modificados antes de su ratificación. A estos ISAs base se les puede añadir extensiones para habilitar funcionalidades específicas, como operaciones en punto flotante, multiplicación y división, operaciones vectoriales, etc. Cada extensión viene identificada con una letra que debe añadirse al ISA base para representar las capacidades hardware de un procesador, como se muestra en el Cuadro 2. Por simplicidad, si un procesador soporta las extensiones M, A, F y D, las cuales son bastante comunes, se emplea la letra G (General).

RISC-V cuenta con cuatro tipos principales de instrucciones (R/I/S/U) y dos adicionales

Extensión	Descripción	Estado
M	**Multiplicación y división de enteros**	**Ratificado**
A	Instrucciones atómicas	Congelado
F	**Punto flotante de precisión simple**	**Ratificado**
D	**Punto flotante de precisión doble**	**Ratificado**
Q	**Punto flotante de precisión cuádruple**	**Ratificado**
L	Punto flotante decimal	Borrador
C	**Instrucciones comprimidas**	**Ratificado**
B	Manipulación de bits	Borrador
J	Lenguajes traducidos dinámicamente	Borrador
T	Memoria transaccional	Borrador
P	Instrucciones Packed-SIMD	Borrador
V	Operaciones vectoriales	Borrador
N	Interrupciones a nivel de usuario	Borrador

Cuadro 2: Extensiones de RISC-V [7]

(B/J) [7]:

- R: instrucciones que usan 3 registros con los que se realizan operaciones aritmético-lógicas (add, xor, mul, etc.)
- I: instrucciones con inmediatos y loads (addi, lw, etc.)
- S: instrucciones de almacenamiento (sw, sb, etc.)
- U: instrucciones con inmediatos superiores (lui, auipc, etc.)
- B y J: instrucciones de salto (beq, jal, etc.)

Todos los tipos principales de instrucciones son de una longitud fija de 32 bits, y deben estar alineadas en un límite de cuatro bytes en memoria. Los registros de entrada (*rs1* y *rs2*) y salida (*rd*) se mantienen en la misma posición para así facilitar la decodificación. En la figura 1 se muestran los formatos de cada tipo de instrucicón.

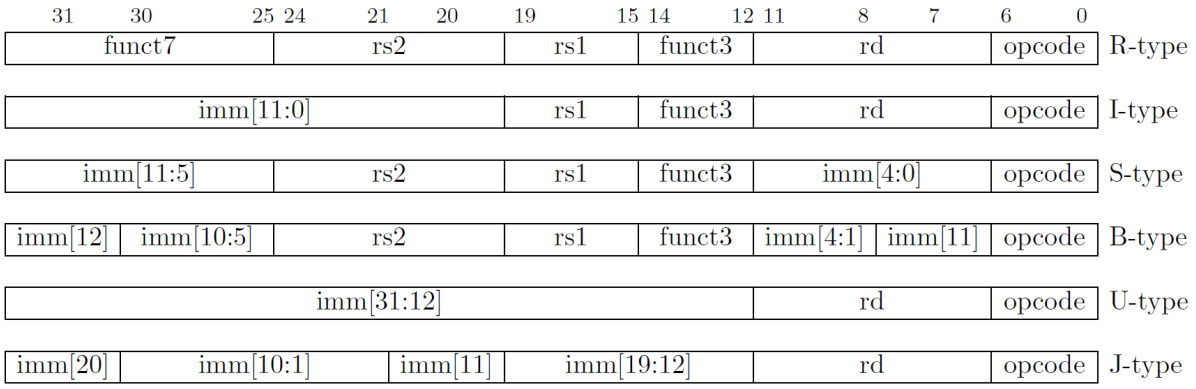

Figura 1: Tipos de instrucciones en RISC-V [7]

Core Name	RISC-V Type	Pipeline Stages	Threads	Size @ TSMC	CoreMarks/Mhz
SweRV Core EH1	RV32IMC	9- dual issue	Single	.11mm @ 28nm	4.9
SweRV Core EH2	RV32IMC	9- dual issue	Dual	.067 @ 16nm	6.3
SweRV Core EL2	RV32IMC	4- single issue	Single	.023 @ 16nm	3.6

Figura 2: Características de los procesadores SweRV de Western Digital [7]

2.2. SweRV EH1

El SweRV EH1 es un procesador RISC-V desarrollado por Western Digital, y el empleado en el proyecto RVfpga. Cuenta con el ISA base RV32I (enteros de 32 bits), junto con las extensiones M (multiplicación y división de enteros) y C (instrucciones comprimidas). Ha sido diseñado junto con los procesadores EH2 y EL2, cada uno de los cuales cuenta con distintas características, que se ven reflejadas en la figura 2. De entre esos tres, ha sido el elegido debido a su alta potencia por megahercio y a su simplicidad al ser monohilo.

Es un procesador superescalar de 32 bits, con un *pipeline* de 9 etapas y 2 vías de ejecución. Ambas vías cuentan con un pipe para realizar operaciones ALU (I0 e I1) y, además, en una de las vías se llevan a cabo las operaciones load/store, mientras que en la otra se realizan las operaciones de multiplicación. Para el cálculo de las divisiones, existe un divisor de 34 ciclos fuera del pipe. La etapa de *fetch* está dividida en dos etapas, e incluye un predictor de salto de

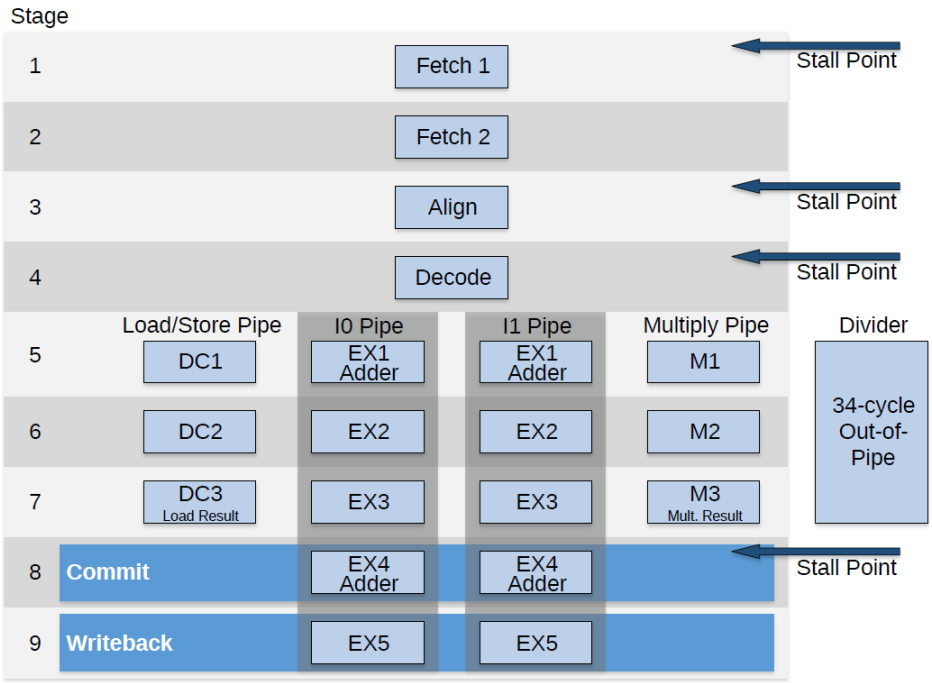

Figura 3: Pipeline del procesador SweRV EH1 [9]

tipo Gshare [8]. Durante la etapa de decodificación, en cada ciclo se decodifican hasta dos instrucciones provenientes de cuatro buffers. En la etapa de *commit* se consolidan también hasta dos instrucciones y, finalmente, en la etapa de *writeback* se actualizan los registros necesarios. A continuación se explican con mayor detalle las distintas etapas del pipe.

- **Etapa de Fetch**

  Esta etapa se encuentra a su vez dividida en las etapas FC1 y FC2, y es la responsable de leer instrucciones de la memoria. La dirección de la instrucción se calcula en la etapa FC1, y se le proporciona al controlador de la memoria de instrucciones. Durante la etapa FC2, se lee la instrucción, bien de la caché de instrucciones o bien de la memoria principal. Si está presente en la caché, la lectura se realiza de forma inmediata. En caso contrario, el pipeline ha de detenerse hasta que la memoria principal proporcione la instrucción, proceso que puede tomar varios ciclos. En el proyecto RVfpga, el sistema cuenta con una caché de instrucciones de 16 KiB y una memoria externa de 128MiB. Si no hay paradas, se pueden leer hasta dos instrucciones por ciclo. En caso contrario, se almacenan en un búfer de 128 bits, permitiendo almacenar hasta cuatro instrucciones.

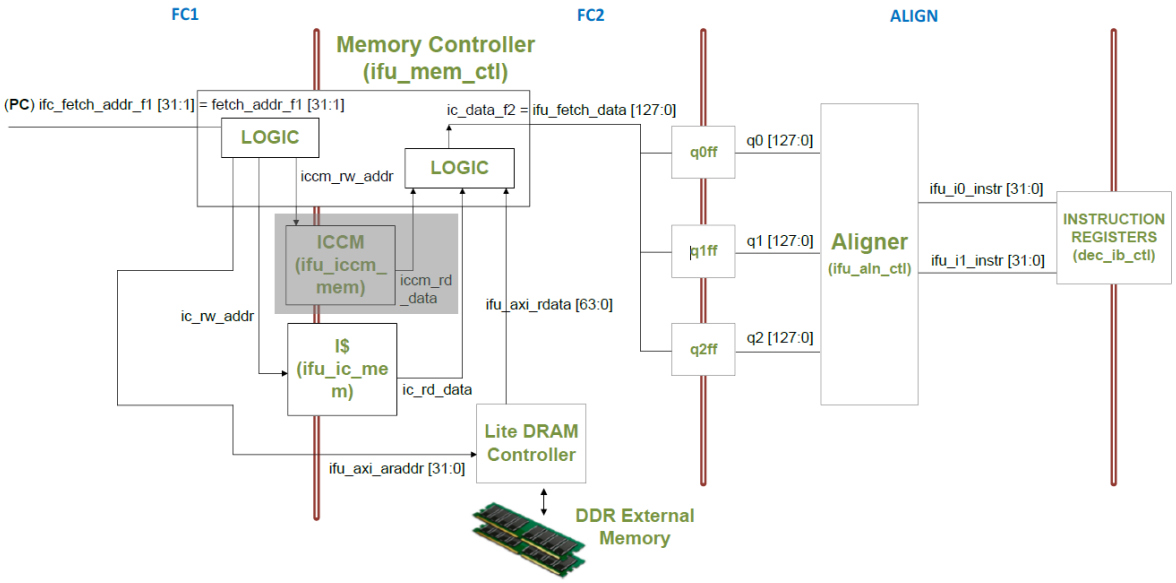

Figura 4: Lógica de las etapas Fetch y Align [2]

- **Etapa de Align**

  Esta etapa cumple dos funciones principales. La primera de ellas es proporcionar dos instrucciones de 32 bits por ciclo a la etapa de decodificación. Para ello, las extrae del búfer de 128 bits y asigna cada una a una vía distinta del pipeline, almacenándolas en dos registros de instrucciones. La otra función es descomprimir las instrucciones: la extensión de RISC-V para las instrucciones comprimidas disminuye el tamaño de las instrucciones de enteros y punto flotante a 16 bits, reduciendo el tamaño de los campos de control, inmediatos y registros. Emplear un tamaño limitado de instrucciones atenúa el coste, el consumo y la memoria necesaria. En la etapa de Align se descomprimen las instrucciones, ya que el decodificador necesita instrucciones de 32 bits.

- **Etapa de Decodificación**

  En cada ciclo, la unidad de control recibe hasta dos instrucciones descomprimidas. A continuación, se encarga de decodificar las instrucciones para generar las señales de control, las cuales son propagadas a las etapas posteriores del pipeline mediante el uso de los registros del control (ver figura 5). Una vez decodificadas, cada instrucción es enviada a un pipe distinto en función del tipo de operación que se realice. Como se ha explicado anteriormente, el SweRV EH1 cuenta con dos pipes para operaciones ALU de enteros,

un pipe para operaciones de multiplicación y un pipe para instrucciones Load/Store, además de un divisor de 34 ciclos fuera del pipe. El planificador de tareas tiene la misión de programar la ejecución de dos instrucciones simultáneas siempre que pueda, pero hay casos en los que no es posible. Por ejemplo, si se decodifican dos instrucciones de multiplicación en el mismo ciclo, o si hay instrucciones ALU con dependencias entre ellas.

- **Etapa de Ejecución**

  Dependiendo del pipe al que pertenezca, cada instrucción se ejecutará de una manera distinta. En el caso de las operaciones de enteros, los pipes I0 e I1 cuentan cada uno con tres etapas EX1, EX2 y EX3. La unidad ALU realiza las operaciones en un ciclo en la etapa EX1, mientras que en EX2 y en EX3 no se llevan a cabo grandes tareas, pero son fases necesarias para sincronizar este tipo de instrucciones con el resto.

  El pipe de multiplicación está dividido en las etapas M1, M2 y M3, ya que cuenta con un multiplicador de enteros en el que dicha operación necesita de tres ciclos para completarse.

  El pipe de Load y Store también está formado por tres etapas. Para las instrucciones del primer tipo, en la etapa DC1, el sumador calcula la dirección a la que hay que acceder sumando la dirección base del registro y el offset del inmediato. En la etapa DC2, las instrucciones Load leen la memoria usando la dirección calculada. Si esta dirección lleva a la caché de datos, las lecturas se realizan en un sólo ciclo. Sin embargo, si hay que ir hasta la memoria principal, el pipeline puede llegar a detenerse durante varios ciclos, dependiendo de si se han usado Loads bloqueantes o no, o de si existen dependencias de datos. En la etapa DC3, las instrucciones Store comienzan a escribir en memoria, proceso que puede llevar varios ciclos. Si la escritura tiene como destino la caché de datos, primero se escribe en el búfer de almacenamiento. En caso de que el destino sea la memoria principal, la escritura se realiza directamente allí.

  Las divisiones se llevan a cabo en un divisor no segmentado de 34 ciclos, que se encuentra fuera del pipe.

- **Etapa de Commit**

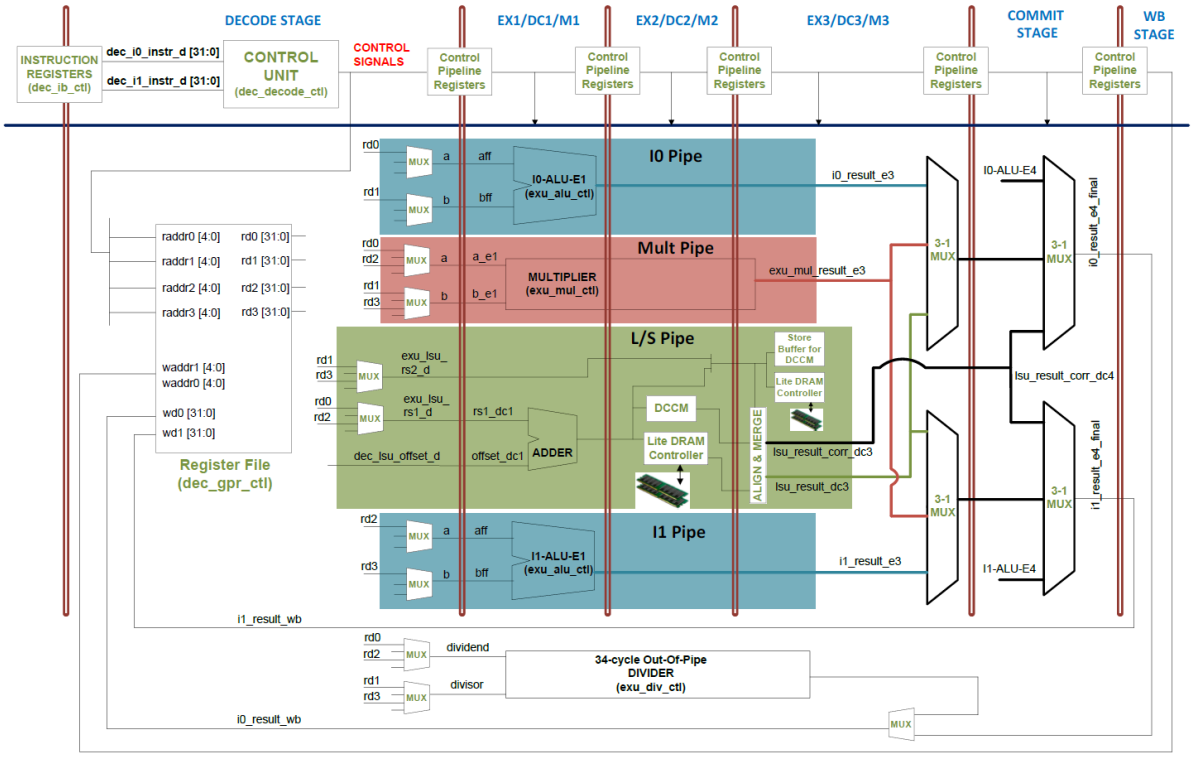

Figura 5: Lógica de las etapas de Decodificación, Ejecución, Commit y Writeback [2]

En esta etapa, dos multiplexores (uno por cada vía) seleccionan el resultado a escribir en en los registros de destino.

- **Etapa de Writeback**

  En la etapa final se escriben los resultados en los registros correspondientes. Se pueden escribir hasta dos registros por ciclo, aunque esto no siempre será necesario, pues hay instrucciones (como las instrucciones de salto o de Store) que no guardan datos en ningún registro.

2.2.1. SweRV EH1 Core Complex

Western Digital ha desarrollado también una versión extendida de este procesador, denominado *SweRV EH1 Core Complex*. Esta versión modificada añade los siguientes elementos:

- Dos memorias estrechamente acopladas al procesador: una para instrucciones (ICCM) y otra para datos (DCCM). Estas memorias proporcionan un acceso de baja latencia, y

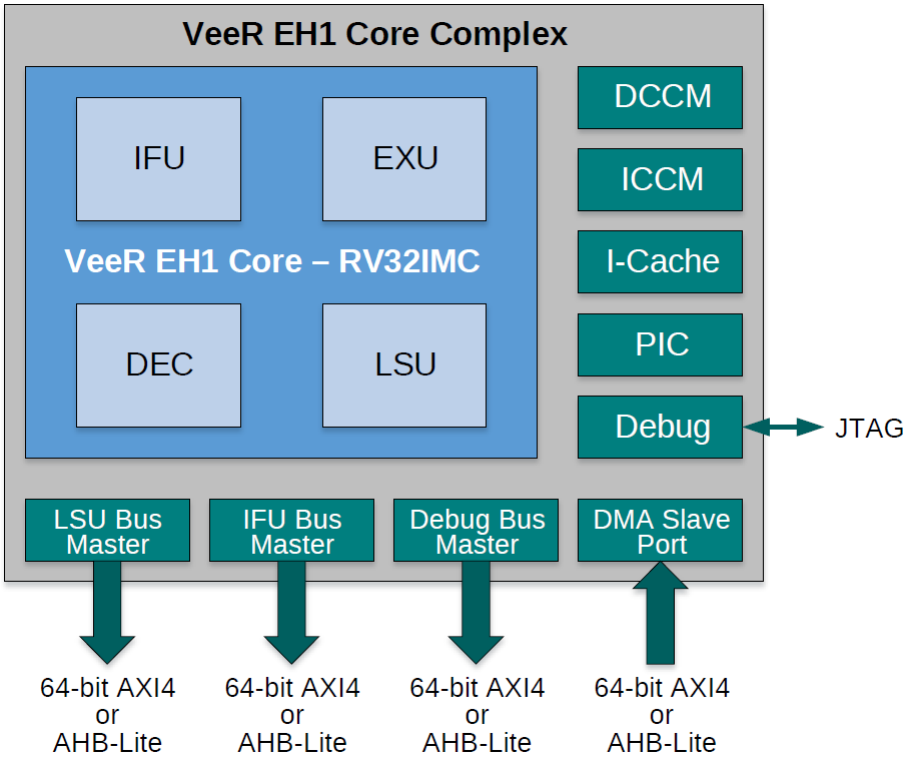

Figura 6: SweRV Core Complex [9]

cuentan con un código corrector de errores (ECC). Pueden configurarse para tener una capacidad de entre 4 y 512 KB.

- Una caché opcional de 4 vías y asociativa, con paridad de bits o ECC.
- Un Controlador Programable de Interrupciones (PIC) opcional, con soporte para hasta 255 interrupciones externas.
- Cuatro interfaces de buses del sistema, para el fetch de instrucciones (IFU), acceso a datos (LSU), acceso al debug y acceso la DMA[6] externa.
- Una unidad central de debug.

2.2.2. SweRVolfX SoC

El SoC SweRVolf parte del SweRV EH1 Core Complex, al que se le añade una ROM de arranque, un UART, un controlador del sistema y un controlador SPI[7]. Debido a que el EH1

[6]Direct Memory Access
[7]Serial Peripheral Interface

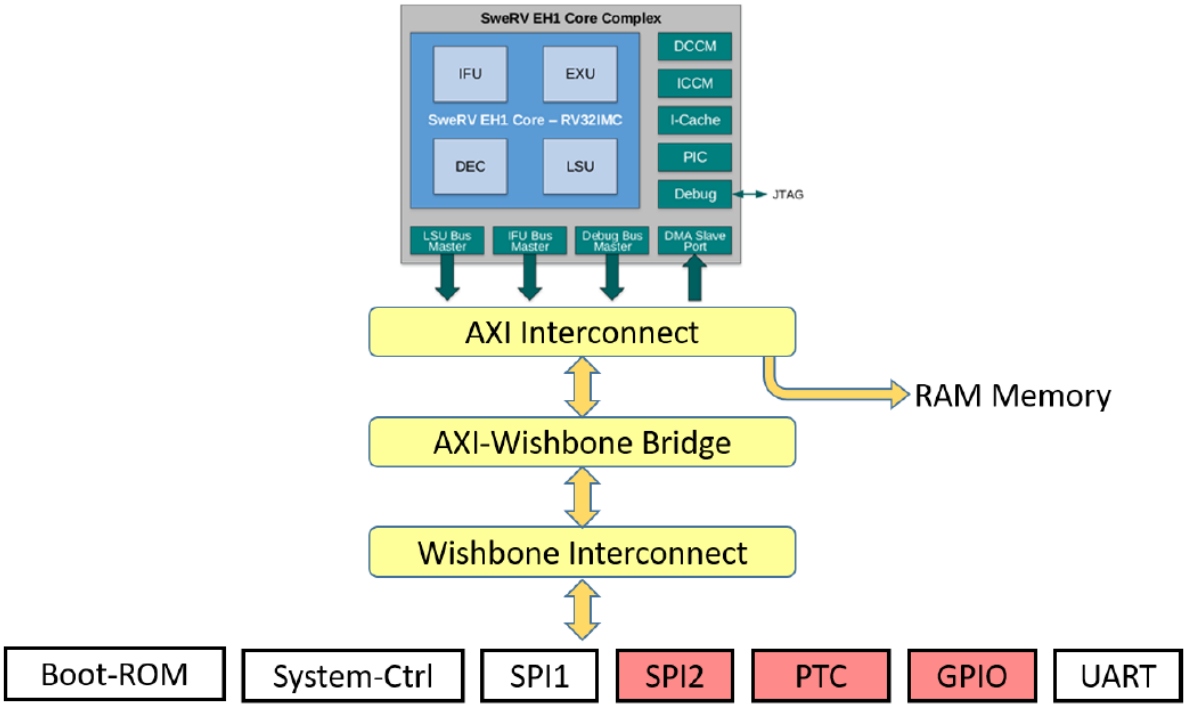

Figura 7: SweRVolfX SoC [2]

utiliza un bus AXI[8] y los periféricos uno de tipo *Wishbone*, en el SoC está presente un puente AXI-Wishbone.

En el proyecto RVfpga se le añade algo más de funcionalidad al SoC, en la forma de un controlador SPI adicional, un controlador GPIO, un módulo PTC[9] y un controlador de displays de 7 segmentos, dando como resultado el SweRVolfX SoC, el cual se muestra en la figura 7.

2.3. Proyecto RVfpga

El proyecto RVfpga, desarrollado por Imagination University Pogramme, tiene como objetivo proporcionar al usuario el conocimiento necesario para entender el funcionamiento de un procesador RISC-V comercial, de un SoC RISC-V y del ecosistema RISC-V. Para ello se explican los elementos del sistema: desde el diseño digital y señales hasta el conjunto de instrucciones de la arquitectura, junto con el entorno de programación del procesador y el compilador.

Con el objetivo de tener acceso al procesador SweRV EH1 y modificarlo, se ha empleado una FPGA. Las FPGAs son un tipo de circuito integrado que puede ser reprogramado tras ser

[8] Advanced eXtensible Interface
[9] PWM/Timer/Counter

producido. Están formadas por una matriz de bloques lógicos y conexiones que se pueden configurar para realizar distintas funciones [10]. Debido a la flexibilidad que proporcionan y a su coste contenido, son muy utilizadas en entornos de desarrollo y prototipado. Para configurarlas, normalmente se utiliza un Lenguaje de Descripción de Hardware (HDL).

El SoC SweRVolfX puede ser funcionar directamente en la FPGA Nexys A7 o bien mediante simuladores de HDL, y entre los archivos incluidos en el proyecto se encuentran los ficheros necesarios para ambas opciones.

La Nexys A7 es una placa de desarrollo de Digilent que usa la FPGA Xilinx Artix-7. Esta cuenta con más de 15.000 unidades lógicas, 170 pines de entrada/salida y un reloj interno de 450MHz. Además, en la placa se incluyen:

- 128 MiB de memoria RAM DDR
- 128 Mibit de memoria Flash SPI
- 8 displays de 7 segmentos
- 16 switches
- 16 LEDs
- Distintos sensores y conectores, incluyendo micrófono, jack de audio, puerto VGA, sensor de temperatura, y acelerómetro, entre otros.

2.4. Punto flotante. IEEE 754

El punto flotante es un formato utilizado en computación para representar números reales usando un número entero c con una precisión fija , llamado mantisa, escalado por un exponente entero q de una base b, junto con su signo s. De esta forma, el valor numérico v de un número finito es:

$$v = (-1)^S \times c \times b^q$$

Por ejemplo, si empleamos la base 10, con signo positivo, mantisa 12345 y exponente -3, el valor obtenido es $(-1)^0 \times 12345 \times 10^{-3} = 12{,}345$

Debido a los problemas derivados de que distintos fabricantes desarrollasen diferentes implementaciones de este tipo de formato, en 1984 se creó el primer estándar para la aritmética de punto flotante, el IEEE 754. El estándar vigente [12], actualizado en 2008, define:

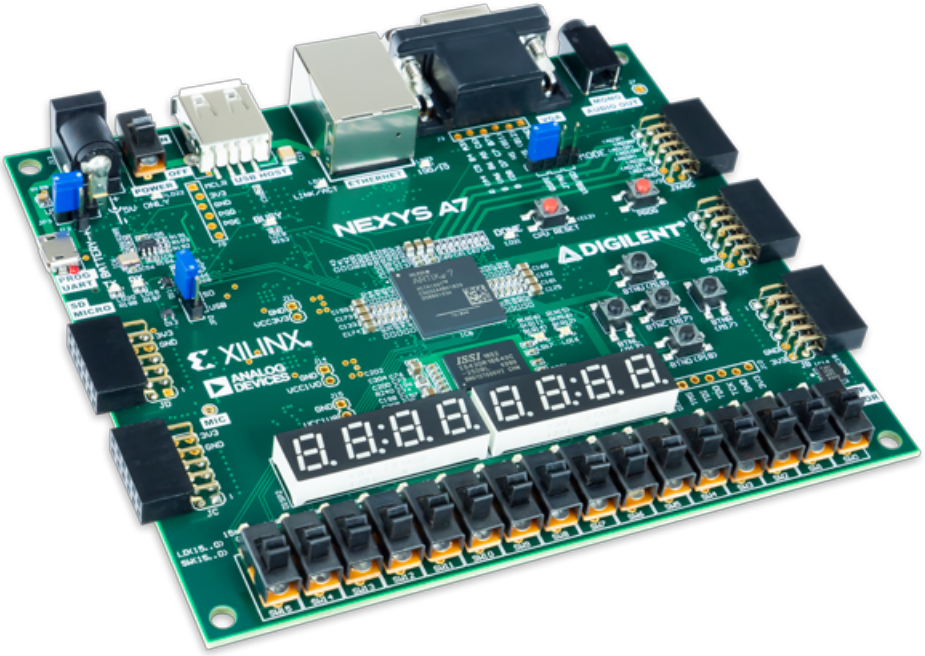

Figura 8: Placa FPGA Digilent Nexys A7 [11]

- Formatos para representar datos en punto flotante en binario y decimal, para la computación y el intercambio de datos.

- Operaciones como la suma, resta, multiplicación, comparación y raíz cuadrada, entre otras.

- Conversiones entre números enteros y en punto flotante, así como entre distintos formatos de punto flotante.

- Manejo de excepciones y errores, incluyendo valores no numéricos (NaN).

Los formatos más comúnmente usados son los binarios de precisión simple (32 bits) y doble (64 bits), pero también existen otros, como la precisión cuádruple, media precisión (usado en Deep Learning) o formatos decimales. En la figura 9 se muestra la disposición de los bits para representar un número en precisión simple. En precisión doble se sigue un patrón similar, pero empleando más bits para el exponente y la mantisa.

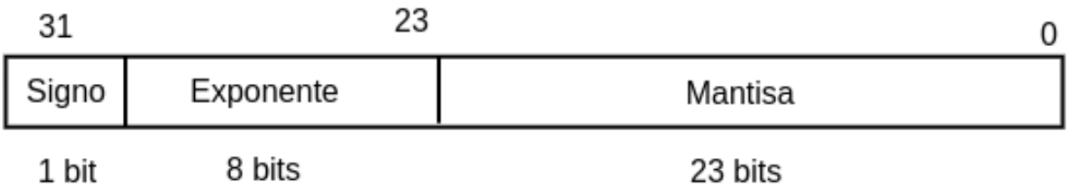

Figura 9: Representación de punto flotante con precisión simple [12]

2.5. Formato HUB

Dos componentes básicos que se utilizan en muchas operaciones aritméticas son los circuitos de redondeo y los circuitos de complemento de la base. Los circuitos de redondeo se emplean cuando es necesario reducir el número de dígitos significativos. Si bien este redondeo puede llevarse a cabo de diversas modos, el más común, y el que se usa en el estándar IEEE-754 es el redondeo al número más cercano [12]. El circuito de complemento de la base es usado para cambiar el signo del número, lo cual puede ser necesario para realizar las restas o para obtener el valor absoluto. Cualquier mejora en la eficiencia que se realice en estos circuitos puede afectar directamente al rendimiento de las unidades funcionales que los incluyan, debido especialmente a que tales circuitos suelen formar parte del camino crítico de estas unidades [3].

El formato HUB (Half-Unit Biased) es una forma alternativa de representar números reales, cuyo principio se basa en desplazar en media unidad los números exactamente representables (ERNs). En punto flotante, dado un número determinado de bits, los números exactamente representables son aquellos cuyos valores pueden ser representados sin ningún error. Por ejemplo, para un sistema FP convencional en el que los ERN se corresponden con los números {0, 1, 2, 3, 4...} en el formato HUB equivalente, estos serían {0.5, 1.5, 2.5, 3.5, 4.5...}. Esto conlleva una serie de implicaciones importantes:

- Los sets de ERNs convencionales y de HUB tienen el mismo tamaño, pero son disjuntos.
- La distancia entre dos ERN consecutivos es la misma en los dos casos. Es decir, ambos formatos tienen la **misma precisión**.
- La conversión de un número convencional a HUB tiene un error de media unidad.
- Para representar un número en HUB hace falta un bit más.

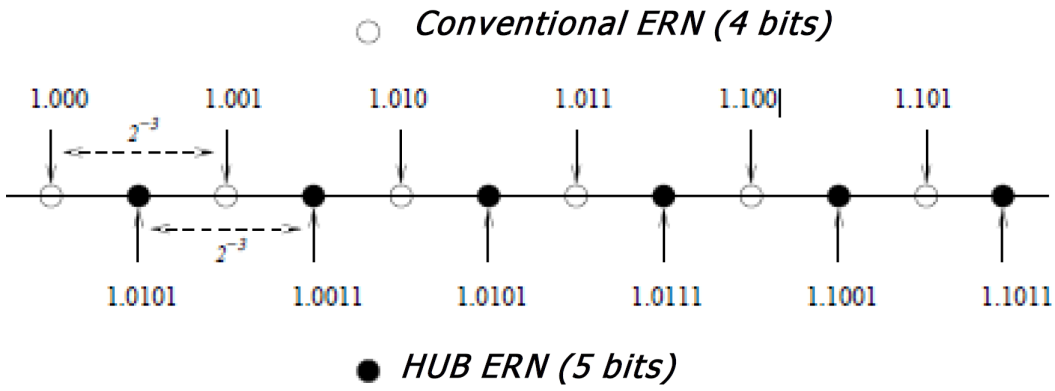

Figura 10: Números ERN en formato convencional y en HUB [13]

Al analizar la figura 10, se puede observar que el bit menos significativo de los números HUB siempre es 1, dando lugar al Bit Menos Significativo Implícito (ILSB), y es que, al saber que este siempre toma el mismo valor, no hace falta tenerlo en cuenta a la hora de almacenar, transmitir o representar un número. Tan sólo sería necesario para realizar las operaciones. Esto podría provocar que en el camino de datos de las operaciones hubiese que añadir un bit extra. Sin embargo, esto se compensa con el hecho de que en HUB no se requiere de bit de redondeo.

En la representación estándar, el método para redondear un número consiste en añadirle media unidad y posteriormente truncarlo. Sin embargo, esa suma puede tener acarreo, haciendo que pueda llegar a ser bastante costosa. En cambio, en el formato HUB, el redondeo se hace simplemente truncando y concatenando un 1 a la derecha del número (o lo que es lo mismo, media unidad), de forma que quede como un valor perteneciente al set de ERNs de HUB. En la figura 11 se muestra este proceso de forma simplificada.

Otra ventaja que presenta HUB es a la hora de realizar el complemento a dos de un número. El complemento a dos sirve sirve para representar números negativos: dado un número positivo, su complemento a dos se realiza invirtiendo todos los bits del mismo y sumándole un 1. Al igual que ocurría con el redondeo, esta suma puede llevar acarreo, concretamente cuando el bit menos significativo del número original es 0. En HUB, como el ILSB siempre vale 1, nunca va a existir acarreo al realizar esta operación.

Con todo esto, en el estudio realizado en [13], se obtiene una mejora del rendimiento en operaciones como la suma de hasta un 14 %, acompañado de una reducción del área y del consumo de un 38 % y 25 %, respectivamente.

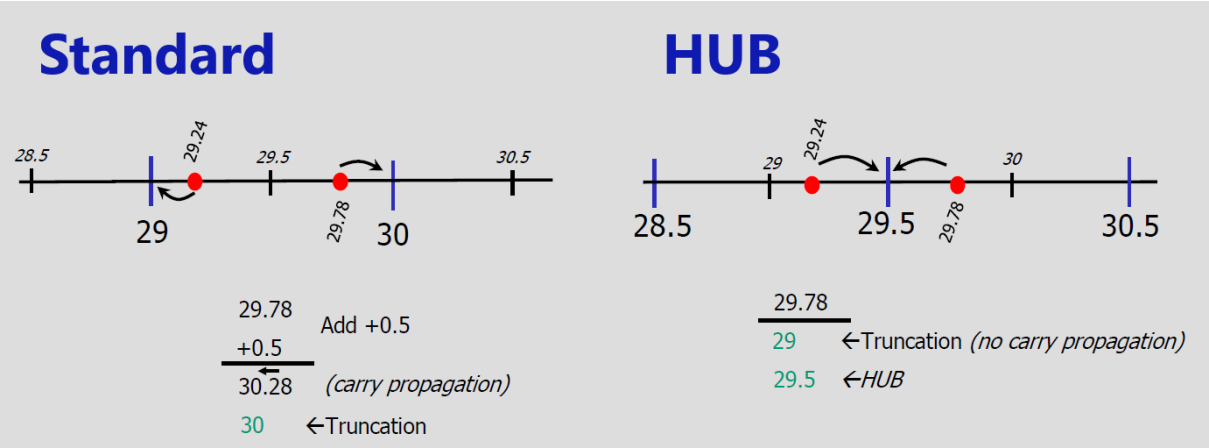

Figura 11: Redondeo en el formato estándar y en HUB [13]

No obstante, el formato HUB presenta ciertos inconvenientes, destacando especialmente su incapacidad para representar números enteros, y su falta de conformidad con el estándar IEEE-754.

2.6. Programas utilizados

Para el desarrollo de este TFG, se han utilizado los siguientes programas y herramientas:

2.6.1. Visual Studio Code

Visual Studio Code (o VSCode) es un editor de código desarrollado por Microsoft, y que permite el uso de extensiones para ampliar sus funcionalidades, así como depurar y ejecutar código. Para poder cargar el SweRV EH1 en la FPGA, se ha empleado la extensión PlatformIO.

2.6.2. Vivado

Vivado es un programa de Xilinx para la síntesis y análisis de diseños de lenguajes de descripción de hardware (HDL), y ha sido utilizado para, una vez modificado el sistema, generar los ficheros necesarios para PlatformIO.

2.6.3. Verilator

Verilator es una herramienta de código abierto que permite convertir código Verilog en un conjunto de ficheros que permiten su simulación.

2.6.4. GTKWave

GTKWave es un visualizador de señales de código abierto, con el que se pueden contemplar las trazas de ejecución de un programa generado con Verilator, permitiendo así comprobar el correcto funcionamiento del sistema antes de generar los ficheros con Vivado, un proceso que puede tomar bastante tiempo.

3

Implementación de Nuevas Funcionalidades

En esta sección, se lleva a cabo la implementación de nuevas funcionalidades en el SweRV EH1, empezando con el soporte para instrucciones de manipulación de bits, continuando con un contador de eventos hardware, y finalizando añadiendo la capacidad de realizar operaciones en punto flotante con el formato HUB.

Para modificar el sistema proporcionado por RVfpga, es necesario acceder a los distintos componentes del SweRV EH1. estos se encuentran en el directorio */src/SweRVolfSoC/SweRVEh1CoreComplex* en forma de archivos verilog, y se corresponden con la unidad de fetch (IFU), de decodificación (DEC) y de ejecución (EXU), entre otros.

3.1. Instrucciones de manipulación de bits

La extensión de RISC-V de manipulación de bits (extensión B, ver cuadro 2) proporciona nuevas instrucciones con el objetivo de reducir el tamaño del código y la energía consumida, a la vez que su uso supone una mejora en el rendimiento del sistema. En este TFG se ha optado por implementar las instrucciones *Max, Maxu, Min* y *Minu*, ya que pueden resultar de utilidad en un amplio rango de aplicaciones. Estas son instrucciones tipo R de números enteros, y devuelven el mayor o menor valor de entre dos operandos. Sus mnemónicos y codificaciones vienen ilustrados en la figura 12.

RV32	RV64	Mnemonic	Instruction
✓	✓	max *rd*, *rs1*, *rs2*	Maximum
✓	✓	maxu *rd*, *rs1*, *rs2*	Unsigned maximum
✓	✓	min *rd*, *rs1*, *rs2*	Minimum
✓	✓	minu *rd*, *rs1*, *rs2*	Unsigned minimum

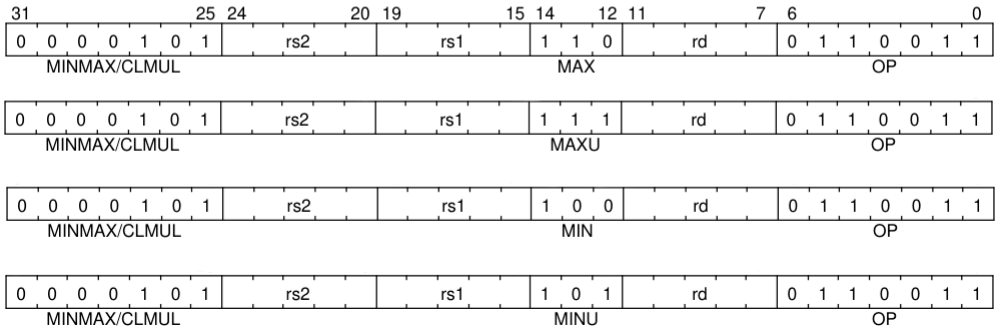

Figura 12: Instrucciones *Max, Maxu, Min* y *Minu* [14]

3.1.1. Unidad de Control

El primer paso para añadir nuevas instrucciones es crear las señales de control necesarias. Para ello, es necesario modificar el fichero verilog *SweRVEHh1CoreComplex/include/swerv_types.sv*. En este, se han de añadir nuevos bits para las instrucciones en las estructuras *dec_pkt_t* y *alu_pkt_t*.

```
typedef struct packed {
                    logic min;
                    logic max;
                    logic minu;
                    logic maxu;
                    ...
                    } dec_pkt_t;

typedef struct packed {
                    logic min;
                    logic max;
                    logic minu;
                    logic maxu;
                    ...
                    } alu_pkt_t;
```

A continuación, es necesario añadir la decodificación de las nuevas instrucciones en el módulo *SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*. Sin embargo, la sección que hay que sobreescribir en este archivo es algo compleja, por lo que en RVfpga se incluye el fichero */dec/decode*, en el que se guardan en un lenguaje natural las decodificaciones y que, junto con los ejecutables *coredecode, espresso.linux* y *addassign*, generan el código correspondiente. En dicho fichero, en la sección *.definition* han de incluirse los códigos de operación de cada instrucción, como se muestra a continuación. En la sección *.output* se asignan los bits que creamos anteriormente, y en la sección *.decode* hay que añadir nuevas líneas con los operandos de las instrucciones.

```
.definition
min  =    [0000101..........100.....0110011]
minu =    [0000101..........101.....0110011]
max  =    [0000101..........110.....0110011]
maxu =    [0000101..........111.....0110011]
...
.output
rv32i = {
      min
      minu
      max
      maxu
      ...
}
...
.decode
rv32i[min]    = { alu rs1 rs2   rd pm_alu min }
rv32i[minu]   = { alu rs1 rs2   rd pm_alu minu }
rv32i[max]    = { alu rs1 rs2   rd pm_alu max }
rv32i[maxu]   = { alu rs1 rs2   rd pm_alu maxu }
...
```

Una vez hecho esto, al correr los ejecutables anteriores se generan los ficheros *equations* y *legal_equations*, que contienen las secciones de código a sobreescribir en el fichero *dec_decode_ctl.sv*. En este, han de introducirse también nuevos bits para las señales de los pipes I0 e I1.

3.1.2. Unidad de Ejecución

La unidad de ejecución viene implementada en los módulos *exu, exu_mul_ctl, exu_div_ctl* y *exu_alu_ctl*. Debido a que las operaciones *Max/Maxu/Min/Minu* son operaciones ALU, solo hace falta modificar el contenido del último de los módulos. Para ello, se declaran las señales y cables necesarios:

```
module exu_alu_ctl
    ...
    logic   sel_MinMaxMinuMaxu
    ...
    assign sel_MinMaxMinuMaxu  = ap.min | ap.max | ap.minu | ap.maxu;
    wire [32:0] minmax_a = {(ap.minu | ap.maxu) ? 1'b0 : a_ff[31], a_ff};
    wire [32:0] minmax_b = {(ap.minu | ap.maxu) ? 1'b0 : b_ff[31], b_ff};
    wire minmax_a_larger_b = $signed(minmax_a) > $signed(minmax_b);
    wire minmax_choose_b = minmax_a_larger_b ^ (ap.max | ap.maxu);
    wire [31:0] minmax_dout = minmax_choose_b ? b_ff : a_ff;
    ...
   assign out[31:0] = sel_MinMaxMinuMaxu ? (minmax_dout) :
                  (({32{sel_logic}} & lout[31:0]) |
                  ({32{sel_shift}} & sout[31:0]) |
                  ({32{sel_adder}} & aout[31:0]) |
                  ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
                  ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
                  ({31'b0, slt_one}));
    ...
endmodule
```

3.2. Contador de Instrucciones tipo I

Más allá de la implementación de nuevas instrucciones, hay otras maneras de expandir la funcionalidad de un procesador. Una de ellas, y la que se ha llevado a cabo en este TFG, es la implementación de un nuevo contador de eventos hardware. Concretamente, un contador de instrucciones tipo I. Como ya se ha visto al final de la sección 2.1.2, estas son instrucciones que involucran algún tipo de inmediato. En este caso, sólo es necesario modificar elementos pertenecientes a la unidad de control.

Al igual que con las instrucciones de manipulación de bits, el primer paso es modificar el

fichero *swerv_types.sv*. En este hay que introducir un nuevo campo en la estructura *inst_t*, en la cual se guardan los distintos tipos de instrucciones, de manera que las que usen inmediatos cuenten como un tipo concreto de instrucción.

```
typedef enum logic [3:0] {
                          LOAD     = 4'b0010,
                          ALU      = 4'b0100,
                          ...
                          // Instrucciones con inmediatos
                          IMM      = 4'b1111
                          } inst_t;
```

El siguiente paso es asignar los bits de control, para lo cual se ha de acceder al módulo *dec_decode_ctl.sv*. La modificación consiste en incluir el nuevo tipo de instrucción en la asignación de las señales *i0_itype* e *i1_itype*, para que funcione en ambos pipes.

```
module dec_decode_ctl
...
if (i0_legal_decode_d) begin
    if (i0_dp.load)                i0_itype = LOAD;
    if (i0_dp.pm_alu)              i0_itype = ALU;
    ...
    // Instrucciones con inmediatos
    if (i0_dp.imm12 & i0_dp.pm_alu) i0_itype = IMM;
    ...
end

if (dec_i1_decode_d) begin
    ...
    // Instrucciones con inmediatos
    if (i1_dp.imm12 & i1_dp.pm_alu) i1_itype = IMM;
    ...
end
...
endmodule
```

Finalmente, hay que añadir el contador de instrucciones con inmediatos al módulo *dec_tlu_ctl.sv*, fichero en el que se encuentran guardados los contadores hardware, entre otros elementos. Inicialmente sólo hay 50 contadores, por lo que se produce desbordamiento si se referencia uno

con un índice mayor, de manera que hay que modificar la señal *event_saturate_wb*, en la cual se encuentra el parámetro que genera ese desbordamiento.

```
module dec_tlu_ctl
   ...
   `define MHPME_INST_IMM        6'd51 // OOP - out of pipe
   ...
   logic [5:0] event_saturate_wb;
   assign event_saturate_wb[5:0] = ((dec_csr_wrdata_wb[5:0] > 6'd51) | (|dec_csr_wrdata_wb[31:6]))
   ? 6'd51 : dec_csr_wrdata_wb[5:0];
...
endmodule
```

3.3. Operaciones en Punto Flotante

Como se ha explicado en la sección 2.4, el punto flotante sirve para representar números reales y realizar operaciones entre ellos. Este tipo de operaciones resulta de especial interés en aplicaciones científicas y en campos como la ingeniería y la arquitectura, ya que es necesario realizar diversos cálculos con una alta precisión. El SweRV EH1 no cuenta con una unidad especializada para las operaciones en punto flotante; sin embargo, permite el uso de las mismas mediante emulación por software. Esta técnica permite ahorrar costes en el diseño y fabricación de los chips, pero a cambio se obtiene un peor rendimiento que con un hardware específico.

A continuación se explica la implementación de tres instrucciones de la extensión F (Punto Flotante de Precisión Simple) de RISC-V: la suma, multiplicación y división. Para ello se ha hecho uso de la unidad de punto flotante desarrollada en [15]. Esta consiste en una serie de módulos Verilog, cada uno de los cuales realiza una operación diferente. En este caso se ha hecho uso de los ficheros *adder.v, divider.v* y *multiplier.v*, que han sido ubicados en el directorio */src/SweRVolfSoC/SweRVEh1CoreComplex/exu*. Ya que esta unidad no es segmentada, para las instrucciones de este tipo se usará el mismo pipe que para las divisiones.

Pese a que la extensión F asume que el procesador tiene 32 registros para punto flotante, por simplicidad se han usado los registros ya existentes en el SweRV EH1. Otras concesiones que se han realizado son permitir que sólo se ejecute una instrucción en punto flotante en un momento dado, y considerar que estas son bloqueantes.

```
fadd.s: 0000000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fmul.s: 0001000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fdiv.s: 0001100 | rs2 | rs1 | Rounding-Mode | rd | 1010011
```

Figura 13: Formato de las instrucciones FP implementadas [7]

3.3.1. Unidad de Control

Nuevamente, como se ha visto en las secciones 3.1.1 y 3.2, el primer fichero a modificar es el que se encuentra en *SweRVEHh1CoreComplex/include/swerv_types.sv*. En esta ocasión, además de incluir los bits de las nuevas instrucciones en la estructura *dec_pkt_t*, hay que crear una nueva estructura *fp_pkt_t*, en la que también se incluyen estos nuevos bits.

```
typedef struct packed {
                      logic fp_add;
                      logic fp_mul;
                      logic fp_div;
                      ...
                      } dec_pkt_t;


typedef struct packed {
                     logic fp_add;
                     logic fp_mul;
                     logic fp_div;
                     } fp_pkt_t;
```

El código para la decodificación de las instrucciones se realiza de la misma manera que como se ha explicado en la sección 3.1.1, es decir, añadiendo al fichero *decode* la información de las nuevas operaciones para que, con los ejecutables, se genere el código correspondiente. Como se usa el pipe de las divisiones, en la sección *.decode* hay que habilitar los mismos bits que en las operaciones de división, junto con los nuevos bits específicos para el punto flotante.

```
.definition
fadd =    [0000000.................1010011]
fmul =    [0001000.................1010011]
fdiv =    [0001100.................1010011]
...
.output
rv32i = {
      fp_add
```

```
      fp_mul
      fp_div
      ...
}
.decode
rv32i[fadd]   = { div rs1 rs2 rd        presync postsync  fp_add }
rv32i[fmul]   = { div rs1 rs2 rd        presync postsync  fp_mul }
rv32i[fdiv]   = { div rs1 rs2 rd        presync postsync  fp_div }
...
```

Tras esto, se incluye en el fichero *dec_decode_ctl.sv* el código generado. En este fichero, también se ha de añadir una nueva señal del tipo *fp_pkt_t* que se creó en el archivo *swerv_types.sv*, y asignarle los valores correspondientes a cada bit.

```
module dec_decode_ctl
    ...
    output fp_pkt_t     fp_p,
    ...
    assign fp_p.fp_add  = i0_dp.fp_add | i1_dp.fp_add;
    assign fp_p.fp_mul  = i0_dp.fp_mul | i1_dp.fp_mul;
    assign fp_p.fp_div  = i0_dp.fp_div | i1_dp.fp_div;
    ...
endmodule
```

3.3.2. Unidad de Ejecución

En la unidad de ejecución, es necesario incluir en el módulo *exu* las señales necesarias para indicar que se va a ejecutar una instrucción de punto flotante.

```
   input fp_pkt_t  fp_p,
   ...
   logic fp_enable, fp_enable_ff;

   assign fp_enable = exu_i0_flush_lower_e4 | exu_i1_flush_lower_e4 |
                      exu_i0_flush_upper_e1 | exu_i1_flush_upper_e1;

   rvdff #(1) final_predict_ff (.*, .clk(active_clk), .din(fp_enable), .dout(fp_enable_ff));

   rvdffe #($bits(predict_pkt_t)) predict_mp_ff (.*, .en(fp_enable | fp_enable_ff),
   .din(final_predict_mp), .dout(final_predict_mp_ff));
```

El grueso de la modificación se encuentra en el módulo *exu_div_ctl*, el pipe de las divisiones, ya que es el que se va a usar para llevar a cabo las operaciones de punto flotante. El primer paso es declarar las señales que sirven como entrada y salida de la unidad fp.

```
input fp_pkt_t      fp_p,
...
logic [5:0]   count_fp;
logic [5:0]   count_in_fp;
logic [31:0]  divisor_fp;
logic [31:0]  dividend_fp;
logic [31:0]  out_fp_add;
logic         out_fp_add_stb;
logic         out_fp_add_stb_delayed;
logic [31:0]  out_fp_mul;
logic         out_fp_mul_stb;
logic         out_fp_mul_stb_delayed;
logic [31:0]  out_fp_div;
logic         out_fp_div_stb;
logic         out_fp_div_stb_delayed;
logic         dp_fp_add_ini;
logic         dp_fp_mul_ini;
logic         dp_fp_div_ini;
```

A continuación, es necesario registrar y manejar los datos y señales de control para las operaciones en punto flotante, mediante el uso de flip-flops (estructura *rvdff(e)*). Para ello, se usan los campos que originalmente se corresponden con el dividendo y el divisor (*dividen_fp* y *divis_fp*) como operandos para la suma, multiplicación y división. Lo siguiente es registrar y retrasar las señales de *strobe* (*dp_fp_str_add, dp_fp_str_div* y *dp_fp_str_div*), que indican la finalización de las operaciones. Finalmente, hay que registrar los resultados de las mismas, asegurando que se capturan en el momento adecuado basado en las señales de *strobe* y la señal de validez *dp.valid*.

```
rvdffe #(32) divis_fp       (.*, .en(dp.valid), .din(divisor[31:0]),  .dout(divisor_fp[31:0]));
rvdffe #(32) dividen_fp     (.*, .en(dp.valid), .din(dividend[31:0]), .dout(dividend_fp[31:0]));
rvdff #(1)   dp_fp_str_add  (.*, .din(out_fp_add_stb),    .dout(out_fp_add_stb_delayed));
rvdff #(1)   dp_fp_str_mul  (.*, .din(out_fp_mul_stb),    .dout(out_fp_mul_stb_delayed));
rvdff #(1)   dp_fp_str_div  (.*, .din(out_fp_div_stb),    .dout(out_fp_div_stb_delayed));
rvdffe #(32) dp_fp_add      (.*, .en(dp.valid || out_fp_add_stb_delayed), .din(fp_p.fp_add),
```

```
   .dout(dp_fp_add_ini));
rvdffe #(32) dp_fp_mul      (.*, .en(dp.valid || out_fp_mul_stb_delayed),  .din(fp_p.fp_mul),
   .dout(dp_fp_mul_ini));
rvdffe #(32) dp_fp_div      (.*, .en(dp.valid || out_fp_div_stb_delayed),  .din(fp_p.fp_div),
   .dout(dp_fp_div_ini));
```

El siguiente paso es modificar la asignación de la señal global que indica que una operación ha finalizado, para que se active también cuando termine una de las operaciones de punto flotante implementadas.

```
assign finish  = dp_fp_add_ini
               ? (out_fp_add_stb)
               : (dp_fp_mul_ini
                  ? (out_fp_mul_stb)
                  : (dp_fp_div_ini
                     ? (out_fp_div_stb)
                     : ((smallnum_case | ((~rem_ff) ? (count[5:0] == 6'd32) :
                     (count[5:0] == 6'd33))) & ~flush_lower & ~flush_lower_ff) ) );
```

La última señal que hay que modificar es en la que se guarda el resultado de la operación realizada, de forma que tenga en cuenta que dicho resultado puede provenir de una instrucción de punto flotante.

```
assign out[31:0]   = dp_fp_add_ini
                   ? out_fp_add
                   : (dp_fp_mul_ini
                      ? out_fp_mul
                      : (dp_fp_div_ini
                         ? out_fp_div
                         : (({32{ smallnum_case_ff          }} & {28'b0, smallnum_ff[3:0]}) |
                   ({32{                     rem_ff}} &  a_ff_eff[31:0]          ) |
                   ({32{~smallnum_case_ff & ~rem_ff}} &  q_ff_eff[31:0]          )) ) );
```

Para finalizar, se definen referencias a los módulos de la unidad FP, con sus entradas y salidas correspondientes. A continuación se muestra sólo el caso del sumador, ya que para el multiplicador y el divisor el proceso es prácticamente idéntico.

```
   adder FloatingPointAdder(
        .input_a(divisor_fp),
        .input_b(dividend_fp),
```

```
.input_a_stb(dp_fp_add_ini),
.input_b_stb(1'b1),
.output_z_ack(1'b1),
.clk(active_clk),
.rst(1'b0),
.output_z(out_fp_add),
.output_z_stb(out_fp_add_stb),
.input_a_ack(),
.input_b_ack());
```

3.4. Adaptación al formato HUB

Una vez que el procesador está listo para ejecutar operaciones en punto flotante, se puede adaptar la unidad FP para que soporte el formato HUB. Si bien el proceso a seguir varía un poco en función de la operación, la metodología general es la siguiente:

1. Concatenar el ILSB a la mantisa de los dos operandos.
2. Realizar la operación de forma normal.
3. Una vez se obtiene el resultado, el redondeo se lleva a cabo simplemente por truncamiento.

3.4.1. Suma

Una suma y resta entre números en punto flotante convencionales normalizados, generalmente cuenta con los siguientes pasos:

1. Se comparan los exponentes de los operandos, y las mantisas se alinean en función de estos. En este proceso se obtiene también el *sticky bit*, que se usa posteriormente para la operación de resta.
2. Se realiza la suma o resta para los bits de la mantisa, junto con los bits *guard, rounding* y *sticky*. Normalmente, para llevar a cabo la resta, se realiza el complemento a dos del operando con el menor exponente, de forma que este quede negado.
3. Tras la operación se suma puede producirse un desbordamiento, por lo que el resultado tiene que ser normalizado desplazándose un bit a la derecha. En caso de una resta, puede

haber una serie de ceros a la izquierda (leading zeros), lo que implicaría un desplazamiento a la izquierda para la normalización del resultado. En ambos casos, el exponente debe ser convenientemente actualizado.

4. El resultado se redondea en función de los dos bits menos significativos y del *sticky bit* y, en paralelo, se genera el nuevo exponente. Si el redondeo se realiza hacia arriba, se puede producir desbordamiento, por lo que habría que calcular de nuevo el exponente.

Para adaptar la suma a HUB, se modifica el fichero *adder.v*. Originalmente, a los operandos *a_m* y *b_m* se les concatenan tres bits a 0 para almacenar los bits *rounding, guard* y *sticky*. Como en HUB no hace falta el bit de redondeo, se usa este como ILSB, poniéndolo a 1. De esta forma, se puede usar un bit más para las operaciones sin ensanchar el camino de datos:

```
a_m = {a[22 : 0], 3'd4};
b_m = {b[22 : 0], 3'd4};
```

Tras esto, se puede eliminar la lógica de los cálculos de los bits *rounding* y *sticky*, ya que no se utilizan. Lo mismo ocurre con el bit *guard* tras la etapa de normalización. Por último, se omite la etapa de redondeo, ya que este se realiza truncando el resultado de la suma.

3.4.2. Multiplicación

En el caso de la multiplicación, habría que concatenar el ILSB a las mantisas de los operandos. Sin embargo, eso requeriría ensanchar el camino de datos de la operación en un bit, por lo que se ha propuesto una solución alternativa: siendo *ma* la mantisa ordinaria del operando *a*, con 24 bits, y *mah* la mantisa en HUB de ese mismo operando (que es igual que la ordinaria, pero concatenando un 1 a la derecha), tenemos que $mah = ma + 2^{-24}$.

A partir de ahí, se puede obtener la siguiente ecuación:

$$mah{\cdot}mbh = (ma + 2^{-24})(mb + 2^{-24}) = ma{\cdot}mb + 2^{-24}(ma + mb) + 2^{-48}$$

En esta, *mah·mbh* sería el resultado final de la operación en HUB, y el valor de 2^{-48} se puede excluir, ya que está fuera del rango de la precisión, quedando la ecuación final de la siguiente forma:

$$mah{\cdot}mbh = ma{\cdot}mb + 2^{-24}(ma + mb)$$

Por lo tanto, para calcular el resultado en HUB, al producto de la multiplicación normal ($ma \cdot mb$), que ocupa 48 bits, habría que añadirle la suma de las mantisas originales desplazada 24 bits hacia la derecha. Tras esto, se trunca el valor obtenido para que ocupe 24 bits y se normaliza el resultado en caso de ser necesario, pero no hace falta llevar a cabo ningún redondeo.

La inserción de estas sumas hace que la mejora en el rendimiento no sea tan elevada como podría llegar a serlo en una adaptación a HUB plena, pero ha sido la mejor solución a la que se ha llegado sin la necesidad de incrementar el tamaño del camino de datos.

En el módulo *multiplier.v*, se introducen las modificaciones mediante las siguientes líneas:

```
product = product + ((a_m+b_m)>>24);
z_m = product[47:24];
```

Además, al igual que ocurre con la suma, los bits *rounding* y *sticky* no son necesarios al no haber redondeo, y el bit *guard* se usa por última vez tras la normalización.

3.4.3. División

Por último, para adaptar la división a HUB, simplemente hay que forzar el ILSB del cociente a 1: tras realizar la división, se obtiene un cociente de 27 bits. Los 24 bits más significativos se guardan en la variable que almacena el resultado, y de los 3 bits restantes del cociente, se pone el más significativo (que se correspondería con el ILSB) a 1.

En el fichero *divider.v*, estos cambios se ven reflejados en las líneas de código que se muestran a continuación:

```
z_m = quotient[26:3];
quotient[2] = 1;
```

De la misma manera que se ha hecho para la suma y la multiplicación, no se requiere de la lógica para calcular los bits *rounding* y *sticky*. Nuevamente, al no haber redondeo, el bit *guard* no se usa tras la etapa de normalización.

4

Testeo y Verificación

En esta sección se muestran las distintas pruebas que se han llevado a cabo para verificar el correcto funcionamiento de las nuevas funcionalidades implementadas. Las comprobaciones se pueden realizar de dos maneras distintas: mediante la simulación a través de Verilator y visualizando las trazas de ejecución en GTKWave, o directamente ejecutando las pruebas en la FPGA y mostrando los resultados obtenidos. En función del caso concreto, será más conveniente emplear un método u otro.

4.1. Instrucciones de manipulación de bits

Debido a que las instrucciones implementadas son instrucciones de comparación entre dos valores, el primer paso para elaborar las pruebas consiste en identificar las distintas casuísticas que pueden tener lugar. Para ello, se establecen las relaciones que puede haber entre los dos valores a comparar:

- Los dos tienen el mismo valor
- Los dos son números positivos diferentes
- Los dos son números negativos diferentes
- Un número positivo y uno negativo, teniendo el positivo un mayor valor absoluto
- Un número positivo y uno negativo, teniendo el negativo un mayor valor absoluto

Teniendo esto en cuenta, se han escrito códigos en ensamblador que evalúan estos cinco casos para las instrucciones *min, minu, max* y *maxu*.

4.1.1. Instrucción min

En la figura 14 se muestra el código en ensamblador para testear la instrucción *min*. Las instrucciones se muestran en hexadecimal, ya que para poder introducir los mnemónicos habría que modificar el compilador, lo cual no es objeto de este proyecto. En esta ocasión se van a comprobar los resultados mediante Verilator. Para ello, se genera la traza de ejecución y se visualiza en GTKWave, procedimiento que se explica en el apéndice A. El resultado que se

```
.globl main
main:

li t1, 0x2
li t2, 0x4
li t3, -0x3
li t4, -0x5

nop
nop
nop
.word 0x0a634f33    # min t5, t1, t1     0000101  00110  00110  100  11110  0110011     min(2,2)
nop
.word 0x0a63cf33    # min t5, t1, t2     0000101  00110  00111  100  11110  0110011     min(2,4)
nop
.word 0x0bde4f33    # min t5, t3, t4     0000101  11101 11100   100  11110  0110011     min(-3,-5)
nop
.word 0x0a7e4f33    # min t5, t2, t3     0000101  00111  11100  100  11110  0110011     min(4,-3)
nop
.word 0x0bd34f33    # min t5, t1, t4     0000101  11101  00110  100  11110  0110011     min(2,-5)
nop
nop
nop

.end
```

Figura 14: Código ensamblador para el test de la instrucción *min*

espera es que en el registro *t5* se almacenen consecutivamente los valores *2, 2, -5, -3, -5.*

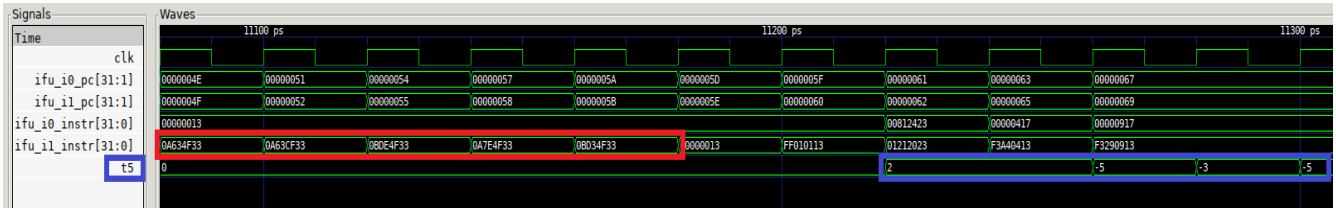

Figura 15: Traza de ejecución de las instrucciones *min*

En la figura 15 aparece remarcada en rojo la codificación hexadecimal de las instrucciones, y en azul los valores del registro *t5*. Como se puede apreciar, la secuencia de valores se corresponde con el resultado propuesto.

4.1.2. Instrucción minu

El código empleado para comprobar esta instrucción, así como las posteriores, es idéntico al de la figura 14, cambiando sólo la operación realizada. Para la instrucción *minu*, al tener en cuenta únicamente el valor absoluto de los registros, la secuencia a obtener debería ser la siguiente: *2, 2, -3, -3, 2*. Dicha secuencia coincide con la que aparece en la figura 16.

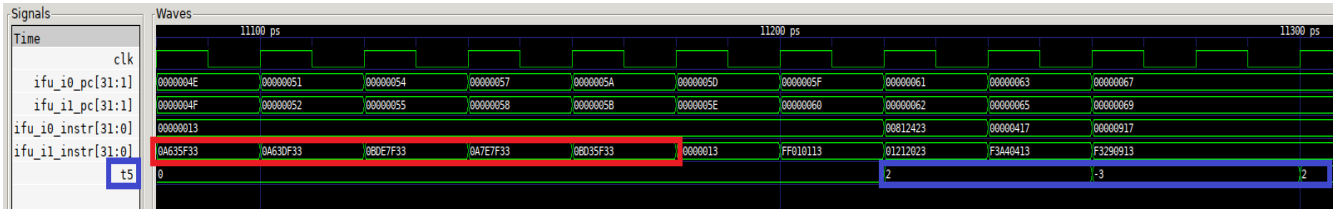

Figura 16: Traza de ejecución de las instrucciones *minu*

4.1.3. Instrucción max

En el caso de la instrucción *max*, la lista de valores que debería obtenerse es la siguiente: *2, 4, -3, 4, 2*. A la vista de la figura 17, se puede comprobar que, efectivamente, los resultados encajan con la solución propuesta.

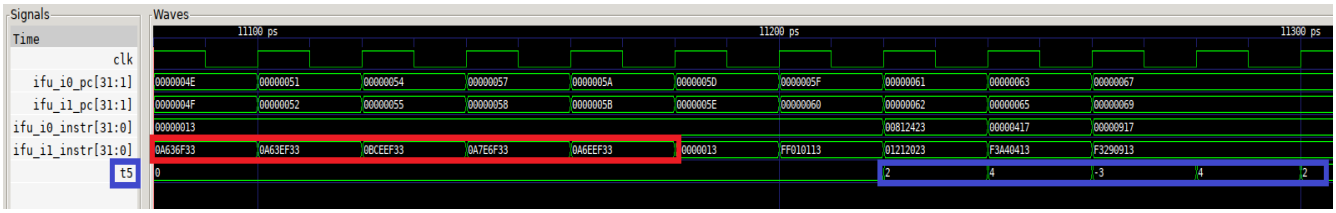

Figura 17: Traza de ejecución de las instrucciones *max*

4.1.4. Instrucción maxu

Finalmente, para la instrucción *maxu*, cabe esperar que en la secuencia aparezcan los números con un mayor valor absoluto, quedando con la siguiente forma: *2, 4, -5, 4, -5*.

Tras verificar en la figura 18 que la secuencia generada con la instrucción *maxu* también coincide con la planteada, se puede concluir que el funcionamiento de las instrucciones *min, minu, max* y *maxu* es el esperado.

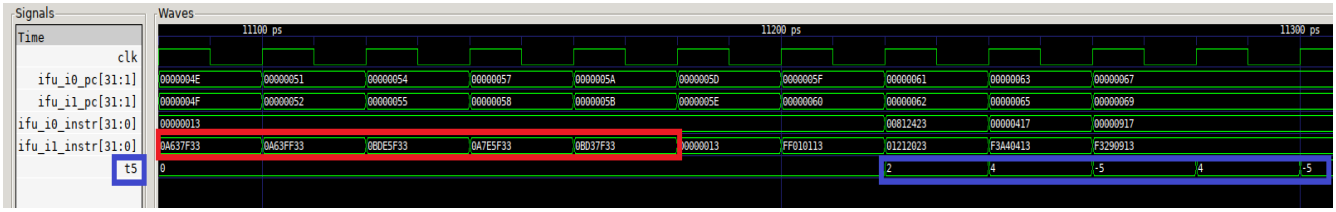

Figura 18: Traza de ejecución de las instrucciones *maxu*

4.2. Contador de Instrucciones tipo I

Para validar el adecuado funcionamiento del contador de instrucciones con inmediatos implementado, se ha escrito un código en C (ver figura 19) en el que se activan varios contadores de eventos para comprobar el rendimiento del sistema, incluyendo el desarrollado en este TFG. A lo largo del código, se obtienen los valores iniciales de los contadores, se llama a la función en ensamblador *Test_Ensamblador* para que se ejecuten distintas instrucciones, y al acabar esta se a obtienen los valores finales de los contadores. De esta forma, se pueden luego comparar los valores iniciales con los finales y así obtener el número de instrucciones que se han ejecutado en *Test_Ensamblador*.

```
// Activar los distintos contadores de eventos
pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
pspPerformanceCounterSet(D_PSP_COUNTER3, 51); // Contador de instrucciones con inmediatos

// Obtener los ciclos y número de instrucciones iniciales
ciclos_ini = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_ini = pspPerformanceCounterGet(D_PSP_COUNTER1);
branch_ini = pspPerformanceCounterGet(D_PSP_COUNTER2);
imm_ini = pspPerformanceCounterGet(D_PSP_COUNTER3);

Test_Ensamblador();

// Obtener los ciclos y número de instrucciones finales
imm_fin = pspPerformanceCounterGet(D_PSP_COUNTER3);
ciclos_fin = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_fin = pspPerformanceCounterGet(D_PSP_COUNTER1);
branch_fin = pspPerformanceCounterGet(D_PSP_COUNTER2);

//Mostrar los resultados
printfNexys("Ciclos = %d", ciclos_fin-ciclos_ini);
printfNexys("Instructiones = %d", instr_fin-instr_ini);
printfNexys("Instrucciones Branch = %d", branch_fin-branch_ini);
printfNexys("Instrucciones con Inmediatos = %d", imm_fin-imm_ini);
```

Figura 19: Código en C para la comprobación del contador I

La función *Test_Ensamblador* es un código en el que se cargan varios valores en distintos registros mediante la instrucción con inmediatos *li*, y en la que a continuación se ejecutan 1000

iteraciones de un bucle que cuenta con dos sumas con inmediatos y otra instrucción de carga *li*. A la vista de la figura 20, se puede predecir que deberían ejecutarse unas 3005 instrucciones con inmediatos: las cinco iniciales, más las 3000 del bucle.

```
.globl Test_Ensamblador

.text

Test_Ensamblador

li a0, 0x0
li t1, 0x1
li t4, 0x4
li t3, 0x3
li a1, 1000
nop

 LOOP:
   add a0, a0, 1
   add t3, t3, 2
   li t4, 0x4
   bne a0, a1, LOOP
.end
```

Figura 20: Función *Test_Ensamblador*

En este caso, la ejecución se va a llevar a cabo en la FPGA, a la cual es posible conectarse por un puerto serie y así visualizar la salida generada. Esta queda reflejada en la figura 21.

```
--- Terminal on /dev/ttyUSB1 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify,
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Ciclos = 4406
Instructiones = 4082
Instrucciones Branch = 1023
Instrucciones con Inmediatos = 3015
```

Figura 21: Salida generada en el test del contador I

Como se puede observar, se obtiene un número de instrucciones con inmediatos de 3015, lo cual no coincide exactamente con las 3005 planeadas. No obstante, es bastante probable que esto se deba a algún tipo de *overhead* entre el código en C y la llamada a la función *Test_Ensamblador*. Es decir, que al compilarse y generarse el código máquina, probablemente se haga uso de instrucciones con inmediatos que no aparecen en la función. Prueba de ello es

que el contador de saltos tipo *branch* devuelve un valor de 1023, cuando en la función sólo se han producido 1000.

Para verificar que así ocurre, se ha eliminado una de las sumas con inmediatos que hay dentro del bucle, de manera que deberían ejecutarse exactamente 1000 instrucciones tipo I menos, por lo que ahora el resultado debería de ser 2015.

```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Ciclos = 3386
Instructiones = 3082
Instrucciones Branch = 1023
Instrucciones con Inmediatos = 2015
```

Figura 22: Nueva salida generada en el test del contador I

Tras ejecutar el test de nuevo y visualizar el resultado de la figura 22, se aprecia que, efectivamente, se han ejecutado 2015 instrucciones con inmediatos. Por lo tanto, se puede afirmar que el funcionamiento de este contador es correcto.

4.3. Operaciones en Punto Flotante

Para verificar el buen funcionamiento las operaciones en punto flotante, hay dos aspectos relevantes a tener en cuenta: que los resultados de las operaciones sean correctos, y que la implementación de la unidad FP suponga una mejora sustancial en el rendimiento, ya que, en caso contrario, el gasto en el hardware extra necesario carecería de sentido.

4.3.1. Exactitud de los resultados

Con el fin de validar la correcta operatividad del punto flotante, se ha escrito un código en ensamblador en el que intervienen las distintas operaciones implementadas, como aparece en la figura 23. Si bien las pruebas no se han llevado a cabo de una manera exhaustiva, sí que se han valorado los casos más representativos. En los comentarios se muestran las instrucciones en binario y sus mnemónicos, así como los resultados esperados. En este apartado, la verificación se va a llevar a cabo mediante la simulación en Verilator y GTKWave. Como se ha realizado anteriormente, en la figura 24 aparecen remarcadas en rojo las codificaciones de las instrucciones en hexadecimal, y en azul los resultados, almacenados en el registro *t5*. Como

```
.globl main
main:

li t3, 0x416a0000  # 14,625
li t4, 0x40500000  # 3,25

    nop
    nop
    nop
    .word 0x01ce8f53  # 0000000 | 11100 | 11101 | 000 | 11110 | 1010011 -> fadd.s t5, t3, t4 ->  14,625 + 3,25 = 17,875
    nop
    .word 0x11ce8f53  # 0001000 | 11100 | 11101 | 000 | 11110 | 1010011 -> fmul.s t5, t3, t4 ->  14,625 * 3,25 = 47,53125
    nop
    .word 0x19ce8f53  # 0001100 | 11100 | 11101 | 000 | 11110 | 1010011 -> fdiv.s t5, t3, t4 ->  14,625 / 3,25 = 4,5
    nop

.end
```

Figura 23: Código en ensamblador para las operaciones en punto flotante

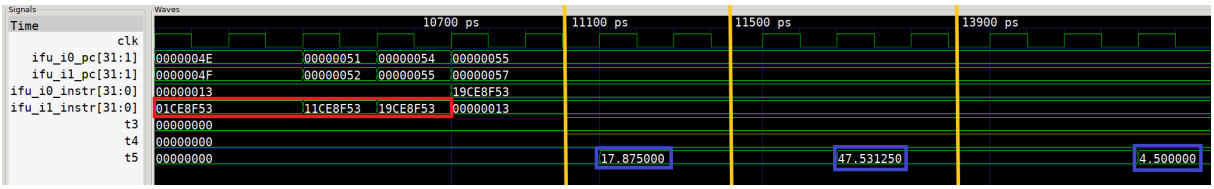

Figura 24: Simulación de las operaciones en punto flotante

estas instrucciones requieren de una mayor cantidad de tiempo para ejecutarse, en la figura se han recortado los resultados originales para que quepan en una imagen.

Para pasar rápidamente de IEEE-754 a decimal, se puede usar un convertidor como el disponible en [16]. Además, GTKWave permite representar los valores de las señales como números reales, facilitando así las pruebas. Al observar la figura 24, se obtienen los valores *17.875, 47.53125,* y *4.5*, los cuales coinciden con los planteados en la figura 23. Siguiendo esta misma metodología, se han realizado más pruebas con distintos valores numéricos y casos límite, obteniéndose siempre valores acertados. Así pues, en este aspecto, la unidad FP funciona de forma correcta.

4.3.2. Mejora del rendimiento

De cara a evaluar el rendimiento del sistema, se ha escrito nuevamente un programa en C, que cuenta con dos secciones: primero se realizan las operaciones FP en una función escrita también en C, de manera que sean emuladas por software, y se miden los ciclos que transcurren durante la misma.

A continuación, se realiza el mismo proceso pero llamando a una función en ensamblador,

```c
void emulacion_sw(){

    for (int i=0; i<DIM; i++) {
        sum += x[i] * y[i];
        division = x[i] / y[i];
    }
}

int main(void)
{
    // llenar los Arrays con 1000 floats aleatorios
    fillArrayWithRandomFloats(x,DIM);
    fillArrayWithRandomFloats(y,DIM);
    int ciclos_inic_sw, ciclos_fin_sw;
    int ciclos_inic_hw, ciclos_fin_hw;

    uartInit();

    // Activar contadores de eventos Hardware
    pspEnableAllPerformanceMonitor(1);
    pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);

    // Medir los ciclos de emulación
    ciclos_inic_sw = pspPerformanceCounterGet(D_PSP_COUNTER0);
    emulacion_sw();
    ciclos_fin_sw = pspPerformanceCounterGet(D_PSP_COUNTER0);

    // Medir los ciclos Hw
    ciclos_inic_hw = pspPerformanceCounterGet(D_PSP_COUNTER0);
    test_ensamblador(DIM, x, y);
    ciclos_fin_hw = pspPerformanceCounterGet(D_PSP_COUNTER0);

    printfNexys("Ciclos Sw= %d\n", ciclos_fin_sw-ciclos_inic_sw);
    printfNexys("Ciclos Hw= %d\n", ciclos_fin_hw-ciclos_inic_hw);
```

Figura 25: Código en C para la prueba de rendimiento del punto flotante

```
test_ensamblador:

  loop: beq a0, zero, endloop
    lw t3, (a1)
    lw t4, (a2)

    .word 0x11ce8f53  # 0001000 11100 11101 000 11110 1010011  f.mult t5, t3, t4

    .word 0x01ce8f53  # 0000000 11100 11101 000 11110 1010011  f.add t5, t3, t4

    .word 0x19ce83d3 #  0001100 11100 11101 000 00111 1010011  f.div t2, t3, t4

    add a0, a0, -1
    add a1, a1, 4
    add a2, a2, 4

    j loop
```

Figura 26: Código en ensamblador para la prueba de rendimiento del punto flotante

para así hacer uso del hardware implementado.

Se ha llevado a cabo este procedimiento realizando 1000 operaciones de cada tipo en una misma ejecución, así como ejecutando cada una por separado, modificando ligeramente el código que aparece en las figuras 25 y 26. Además, se han hecho las pruebas antes y después de adaptar la unidad FP al formato HUB.

En la figura 27 se muestra la diferencia de rendimiento entre la emulación por software,

las operaciones en punto flotante convencionales, y las adaptadas al formato HUB.

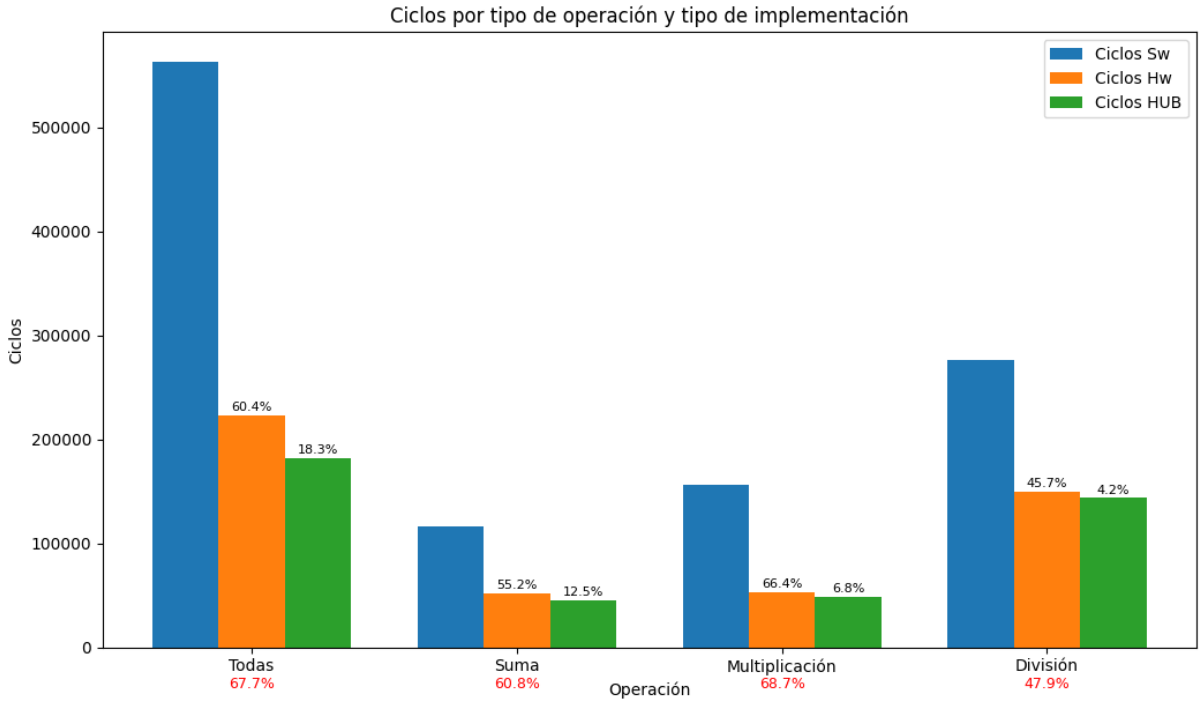

Figura 27: Rendimiento de las operaciones FP

A la vista de la gráfica, se aprecia una mejora del rendimiento muy importante al implementar la unidad FP: del 60,4 % cuando se utilizan todas las operaciones. Por otro lado, al implementar HUB, el salto de rendimiento no es tan elevado, especialmente si se analizan los resultados de cada operación por separado. Sin embargo, al usar todas las operaciones en conjunto se produce una mejora nada despreciable, del 18,3 % con respecto a la implementación de punto flotante convencional. Cabe destacar el 12,5 % de incremento en la velocidad que se produce en la suma HUB con respecto a la estándar, ya que se acerca bastante al 14 % de mejora prometido en [13].

En total, usando todas las operaciones, la mejora en el rendimiento conseguida entre la emulación y el uso de HUB es del 67,7 %.

5

Conclusiones y Líneas Futuras

5.1. Conclusiones

Uno de los fines personales que condujeron a la elaboración de este trabajo de fin de grado era conocer el funcionamiento de la arquitectura RISC-V y el proceso de desarrollo que lleva asociado. Este objetivo ha sido gratamente cumplido y, además, el conocimiento adquirido puede aplicarse a otras arquitecturas y tecnologías.

En cuanto a la implementación de nuevas funcionalidades, el resultado obtenido ha sido satisfactorio: se han conseguido implementar en el SweRV EH1 de forma correcta todas las características propuestas y, especialmente con las operaciones de punto flotante, el rendimiento obtenido justifica su incorporación al sistema.

Al lado de la mejora que se produce entre la emulación por software y la unidad FP convencional, el incremento en el rendimiento en HUB puede no parecer impresionante. Sin embargo, teniendo en cuenta la facilidad con la que se puede llevar a cabo la adaptación, y que esta supone además una reducción en el área y energía requeridas, consideramos que el uso de HUB es una opción viable y eficiente para muchas aplicaciones, especialmente en entornos donde el ahorro de recursos sea crítico.

Durante el transcurso de este TFG, la mayoría de las complicaciones encontradas han residido en la configuración adecuada del sistema y de los proyectos de Vivado y PlatformIO, y es que ejecutar un procesador entero en una FPGA no es tarea ordinaria. Tras solventarlas, uno de los mayores desafíos ha estado en el uso del lenguaje Verilog. Este es un lenguaje que no se había visto antes a lo largo de la carrera, y los lenguajes de descripción de hardware suponen ya de por sí un paradigma bastante diferente al de los lenguajes de programación tradiciona-

les. Con todo, tras un estudio de este y mediante experimentación con diversos ejemplos, se ha obtenido un nivel suficiente para poder llevar a cabo el proyecto.

5.2. Líneas Futuras

Pese a que el resultado general obtenido ha sido más que satisfactorio, siempre hay cabida para la mejora en este tipo de proyectos:

- La unidad de punto flotante usada, aunque sencilla y fácil de adaptar, no es la más eficiente, ya que está basada en una máquina de estados. Una mejora que se podría realizar sería la de buscar otra unidad FP más capaz (por ejemplo, empleando bloques combinacionales) para así incrementar aún más el rendimiento.

- Además, para realizar las operaciones FP se ha utilizado el pipe que originalmente estaba destinado a las divisiones. Se podría introducir un pipe exclusivo para las operaciones de este tipo, junto con un nuevo banco de registros de números en punto flotante.

- Una vez añadidas las nuevas instrucciones, para usarlas en lenguaje ensamblador hay que escribir su codificación en hexadecimal. Su uso sería bastante más sencillo si se modificase el compilador, de forma que se pudiesen escribir los mnemónicos de las operaciones.

- Por otro lado, una vez familiarizado con el entorno, el proceso de implementar nuevas instrucciones no es demasiado complejo, aunque sí resulta algo tedioso. De cara a acelerar el desarrollo de nuevas funcionalidades, se podría crear algún proceso de automatización que, dados los operandos y el código de operación, genere automáticamente el código necesario, por ejemplo mediante scripts.

- También se podría valorar la posibilidad de hacer funcionar el proyecto RVfpga en otras placas FPGA. La Nexys A7, si bien ofrece mucho por los 300€ que cuesta, puede resultar algo cara para estudiantes o particulares que quieran aprender por su cuenta. Existen otras placas de desarrollo más económicas que pueden cumplir con los requisitos necesarios para ejecutar el SweRV EH1.

Referencias

[1] I. Insights, "Advanced Technology Key to Strong Foundry Revenue per Wafer," 2018.

[2] I. U. Programme, *RVfpga: Understanding Computer Architecture.*, 2020. [Online]. Available: https://university.imgtec.com/teaching-download/#rvfpga

[3] J. Villalba-Moreno and J. Hormigo, "New Formats for Computing with Real-Numbers under Round-to-Nearest," 2015.

[4] J. Prosise, "RISC vs. CISC: The Real Story," *PC Magazine, 15*, 1995.

[5] S. O. Aletan, "An Overview of RISC Architecture," in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, 1992, pp. 11–20.

[6] R.-V. Foundation, "RISC-V: The Free and Open Instruction Set," 2016.

[7] S. I. Andrew Waterman, Krste Asanović, *The RISC-V Instruction Set Manual*, 2019.

[8] C. Celio, J. Zhao, A. Gonzalez, and B. Korpan, "The Berkeley Out-of-Order Machine (BOOM)," 2021.

[9] C. Alliance, *RISC-V VeeR EH1 Programmer's Reference Manual*, 2022. [Online]. Available: https://github.com/chipsalliance/Cores-VeeR-EH1/blob/main/docs/RISC-V_VeeR_EH1_PRM.pdf

[10] P. A. Simpson, *FPGA design: Best practices for team-based reuse, second edition*, 2015.

[11] Digilent, *Nexys A7 Reference Manual*, 2019. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[12] I. C. Society, "IEEE Standard for Floating-Point Arithmetic," 2008.

[13] J. Villalba-Moreno and J. Hormigo, "Measuring Improvement When Using HUB Formats to Implement Floating-Point Systems Under Round-to-Nearest," 2016.

[14] RISC-V, "RISC-V Bit-Manipulation ISA-extensions," 2021.

[15] J. P. Dawson, "Synthesiseable IEEE 754 Floating Point Library in Verilog," 2021. [Online]. Available: https://github.com/dawsonjon/fpu

[16] *IEEE-754 Floating Point Converter*. [Online]. Available: https://www.h-schmidt.net/FloatConverter/IEEE754.html

Apéndice A
Manual de Usuario

En este anexo se explica el proceso de instalación y configuración del software necesario en un entorno **Linux**, así como el procedimiento para simular las trazas de ejecución mediante Verilator y la síntesis del sistema en Vivado para la FPGA **Nexys A7**.

A.1. Instalación de VSCode y PlatformIO

Para instalar VSCode, hay que seguir los siguientes pasos:

- Descarga del archivo .deb desde el enlace https://code.visualstudio.com/Download
- En un terminal, se instala y ejecuta VSCode mediante los siguientes comandos:

```
> cd ~/Downloads
> sudo dpkp -i code*.deb
> code
```

PlatformIO es un IDE para sistemas empotrados que se puede instalar como una extensión de VSCode. Es multiplataforma e incluye un debugger. Para configurarlo, el primer paso es instalar las utilidades de python3, mediante el siguiente comando:

```
> sudo apt install -y python3-distutils python3-venv
```

A continuación, en VSCode, hay que acceder al menú de extensiones (remarcado en rojo en la figura 28). Se introduce "*PlatformIO*" en el buscador que aparece y se hace click en *install*.

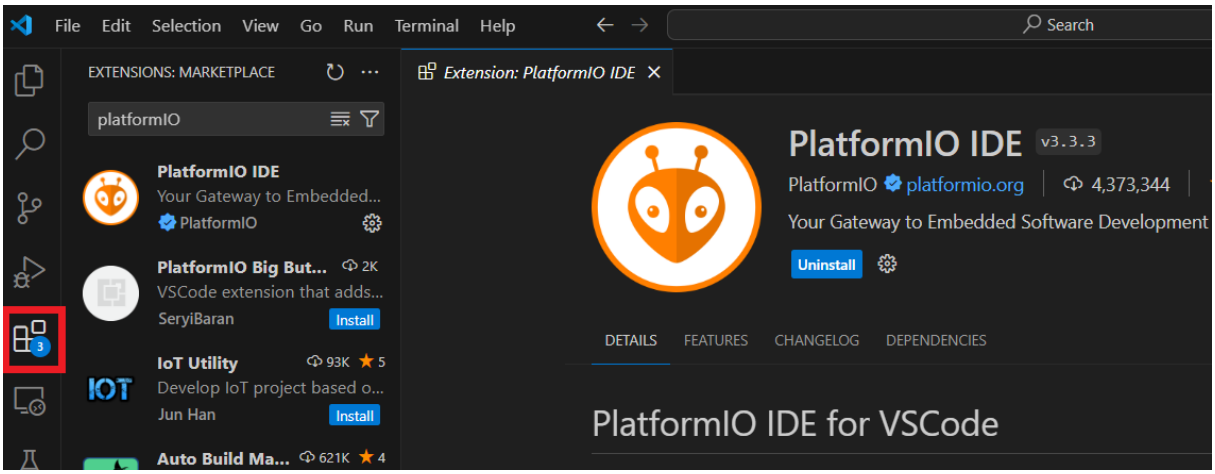

Figura 28: Instalación de PlatformIO en VSCode

A.2. Instalación de Verilator y GTKWave

El proceso para instalar Verilator puede encontrarse en https://verilator.org/guide/latest/install.html, pero se resume en los comandos que se muestran a continuación, junto con la instalación de GTKWave:

```
> sudo apt-get install git make autoconf g++ flex bison libfl2 libfl-dev
> sudo apt-get install -y gtkwave
> git clone https://git.veripool.org/git/verilator
> cd verilator
> git pull
> git checkout v4.106
> autoconf
> ./configure
```

A.3. Instalación de Vivado

Al igual que con Verilator, las instrucciones detalladas para la instalación de Vivado 2019.2 pueden encontrarse en https://reference.digilentinc.com/vivado/installing-vivado/start, y se encuentran resumidas en los siguientes pasos:

1. Se accede a la página de descargas https://www.xilinx.com/support/download.html, y se busca el "*Self Extracting Web Installer*"

2. Para descargar el instalador se pedirá acceder con una cuenta de usuario de Xilinx, por lo que si no se dispone de una, es necesario registrarse.

3. Una vez descargado, se ejecuta con permisos de *root*, introduciendo en un terminal el siguiente comando:

```
> sudo ./Xilinx\_Unified\_2019.2\_1106\_2127\_Lin64.bin
```

4. Se siguen los pasos que aparecen en el instalador de Vivado. En este, hay que seleccionar **Vivado** como producto a instalar (no Vitis), y seleccionamos también Vivado HL **Webpack**. El resto de opciones se dejan por defecto.

5. Una vez instalado, hay que configurar el entorno con el comando que se muestra a continuación. Para que se guarde la configuración tras reiniciar el sistema, se puede añadir esta línea al fichero `/.bashrc`.

```
> source /tools/Xilinx/Vivado/2019.2/settings64.sh
```

6. El siguiente paso es instalar los drivers de los cables para la FPGA Nexys A7. Para ello, se introducen en una consola los siguientes comandos:

```
> cd /tools/Xilinx/Vivado/2019.2/data/xicom/cable_drivers/lin64
/install_script/install_drivers/
> sudo ./install_drivers
```

7. También es necesario instalar los archivos de las placas de Digilent. Tras descargar el contenido del repositorio de GitHub, el cual consiste en una serie de carpetas con todas las placas, se copia en el directorio */tools/Xilinx/Vivado/2019.2/data/boards/board_files*

A.4. Simulación mediante Verilator y GTKWave

Para este apartado y para la sección A.5.2, se han usado los archivos proporcionados por el proyecto RVfpga. Se puede acceder a ellos solicitándolos en https://university.imgtec.com/rvfpga-el2-v3-0-english-downloads-page/.

1. El directorio */RVfpga/verilatorSIM* contiene los ficheros necesarios para generar los binarios de la simulación. Es un proceso sencillo, y será necesario realizarlo cada vez que se modifiquen los componentes del SweRV EH1. Tras ejecutar los comandos que aparecen a continuación, se generará el fichero **Vrvfpgasim**.

```
> cd /ruta_a_rvfpga/RVfpga/verilatorSIM
> make clean
> make
```

2. En este caso se va a partir de un ejemplo proporcionado por RVfpga. Con VSCode y PlatformIO abiertos, en la barra superior se selecciona *File ->Open Folder...*, y se busca el directorio */RVfpga/examples/*.

3. Se selecciona el directorio *AL_Operations y se hace click en OK. El ejemplo se abrirá en PlatformIO.*

4. En el fichero *platformio.ini*, hay que modificar la línea en la que se asigna la variable *board_debug.verilator.binary*, la cual tiene que pasar a ser el directorio completo que lleva al fichero *Vrvfpgasim* generado, como se muestra en la figura 29

5. En el menú de PlatformIO, se hace click en *Generate Trace* y, si todo ha salido bien, aparecerá un mensaje indicándolo, junto con el tiempo empleado, tal y como aparece en la figura 30.

6. Tras el paso anterior, debería haberse generado el fichero *trace.vcd* en el directorio */RVfpga/examples/AL_Operations/.pio/build/swervolf_nexys*. Este puede abrirse con GTKWave mediante el siguiente comando:

```
> gtkwave trace.vcd
```

```
[env:swervolf_nexys]
platform = chipsalliance
board = swervolf_nexys
framework = wd-riscv-sdk

monitor_speed = 115200

#debug_tool = whisper

board_build.bitstream_file = /home/alfonso/Desktop/RVfpga/src/rvfpganexys.bit

board_debug.verilator.binary = /home/alfonso/Desktop/RVfpga/verilatorSIM/Vrvfpgasim
```

Figura 29: Modificación del fichero *platformio.ini* para referenciar el binario *Vrvfpgasim*

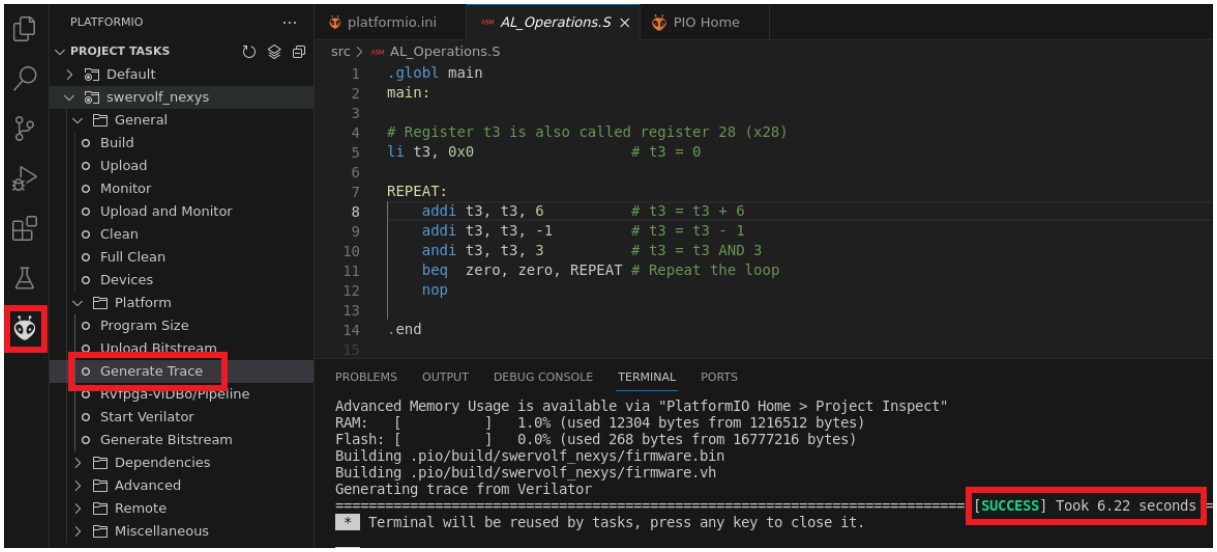

Figura 30: Generación de la traza de ejecución en PlatformIO

7. Una vez en GTKWave, hay que añadir las señales que se desean visualizar. Ello puede hacerse expandiendo la jerarquía en el menú de la izquierda y buscando manualmente las que nos interesan. Para facilitar este proceso, En el proyecto RVfpga se incluye el archivo *RVfpga/examples/AL_Operations/test.tcl*, que se puede leer desde GTKWave y sirve para añadir las señales concretas que resultan de interés en este caso. Se abre haciendo click en *File ->Read Tcl Script File.*

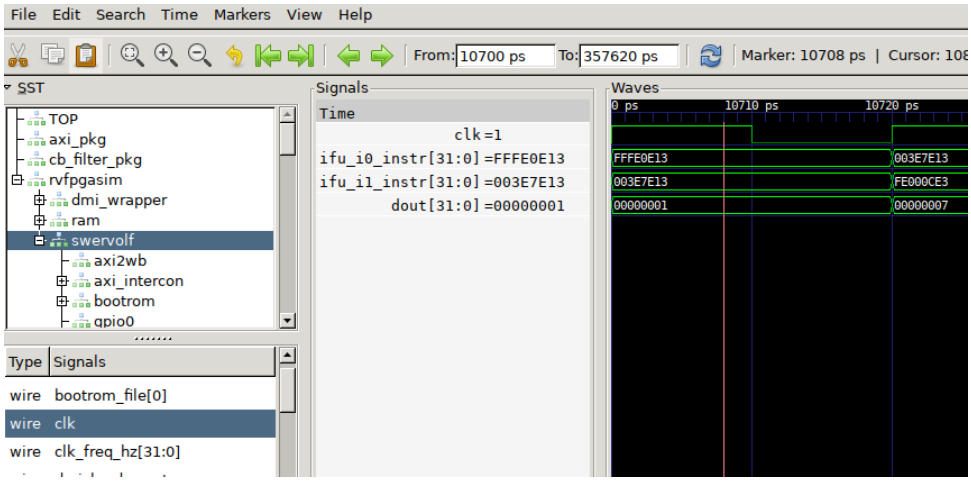

Figura 31: Interfaz de GTKWave

A.5. Síntesis y ejecución en la FPGA

Al igual que se ha hecho en la sección A.4, Para la ejecución de código en la FPGA, se ha partido de ejemplos ya proporcionados.

A.5.1. Generación del Bitstream en Vivado

Para realizar la síntesis de los binarios y generar el bitstream necesario de cara a ejecutar el SweRV EH1 en la Nexys A7, es necesario crear y configurar un proyecto en Vivado. Esta configuración sólo será necesaria realizarla una vez, ya que el mismo proyecto puede reutilizarse para llevar a cabo la síntesis tantas veces como se desee.

1. En Vivado, se crea un nuevo proyecto. Tras indicar el nombre y la ubicación de destino, aparecerá una pestaña en la que hay que indicar que es un proyecto **RTL**.

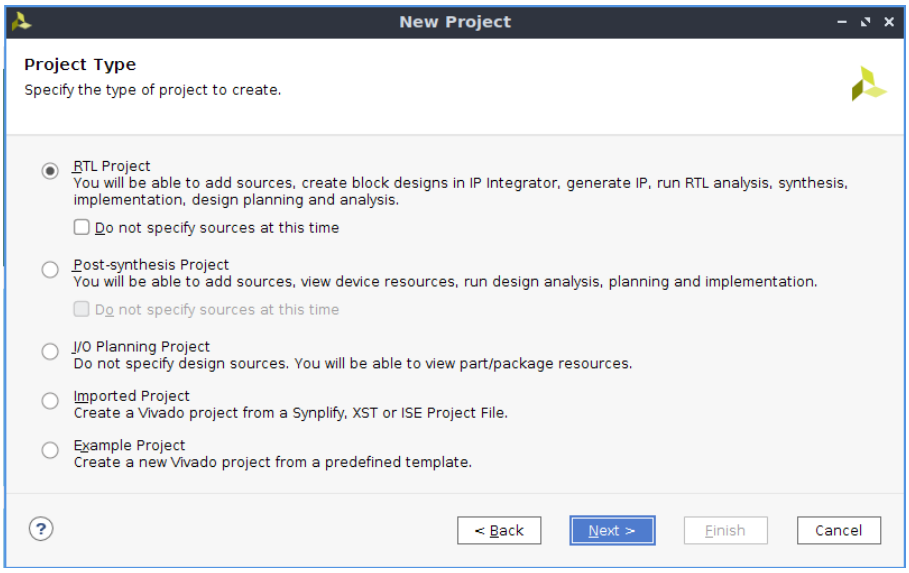

Figura 32: Selección del proyecto RTL en Vivado

2. En la ventana que aparece a continuación, se añaden las fuentes del proyecto, haciendo click en *Add Directories*, y se añade el directorio */ruta_a_rvfpfa/RVfpga/src*. Las casillas *Scan and add RTL include files into project* y *Add sources from subdirectories* deben estar marcadas.

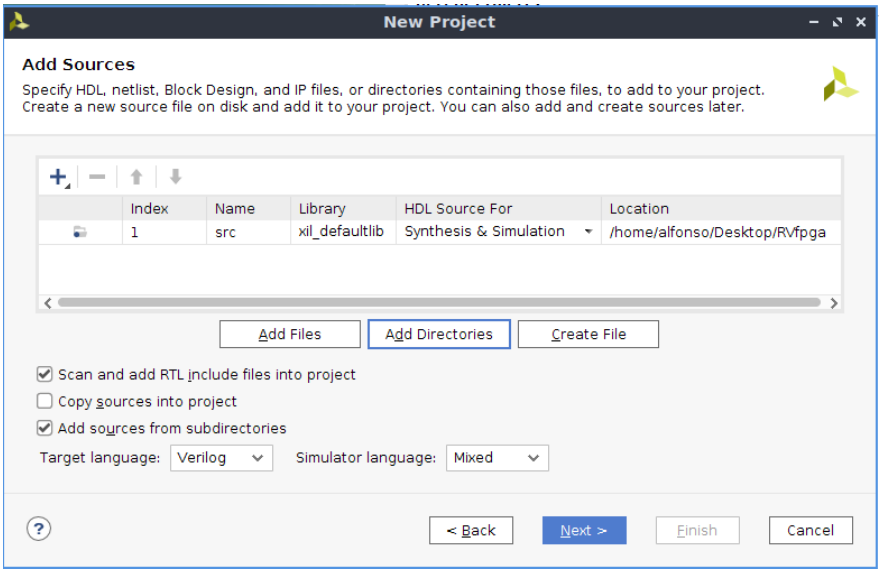

Figura 33: Selección de las fuentes en Vivado

3. En la siguiente pestaña, se deben incluir las limitaciones del sistema. Los ficheros que se van incluir pulsando en *Add Files* mapean los nombres de las señales con los pi-

nes de la placa, y se encuentran en */ruta_a_rvfpfa/RVfpga/src/rvfpganexys.xdc* y */ruta_a_rvfpfa/RVfpga/src/LiteDRAM/liteDRAM.xdc*

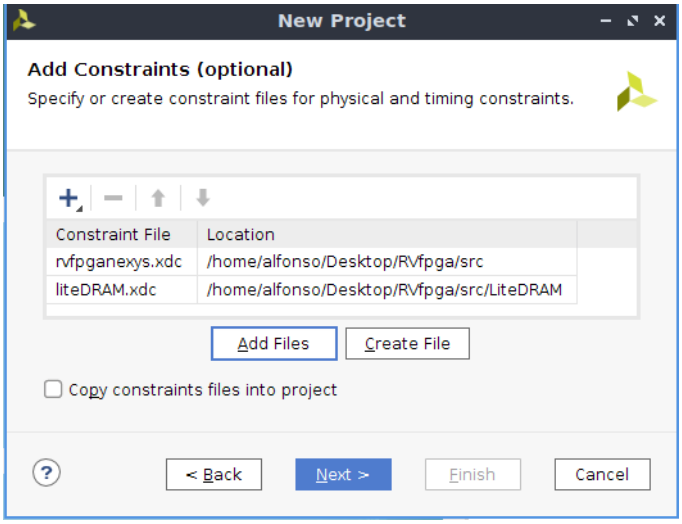

Figura 34: Selección de las limitaciones en Vivado

4. El próximo paso es seleccionar la Nexys A7 como placa objetivo. En la ventana *Default Part*, se hace click en *Boards*. Hay que seleccionar la Nexys A7-100T. Al hacer click en *Next*, aparecerá el resumen de la configuración del proyecto. Si todo está correcto, se pulsa el botón *Finish*.

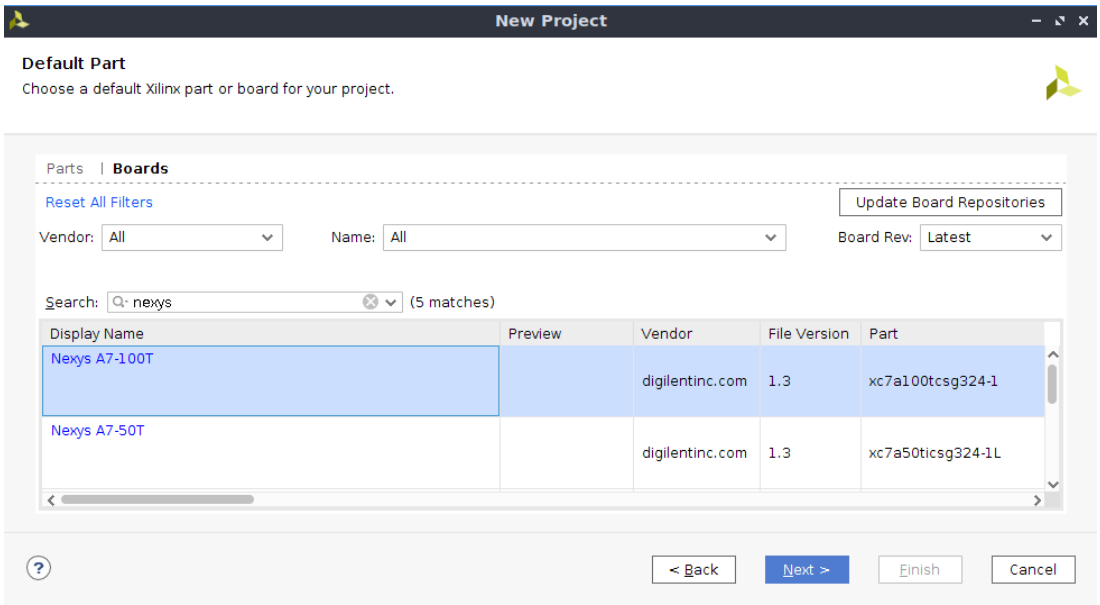

Figura 35: Selección de la Nexys A7 en Vivado

5. Una vez creado el proyecto, es necesario configurar *rvfpganexys* como módulo principal. Para ello, en el panel *Sources*, se busca el desplegable *Design Sources*, se hace click derecho en *rvfpganexys* y se marca la opción *Set as top*.

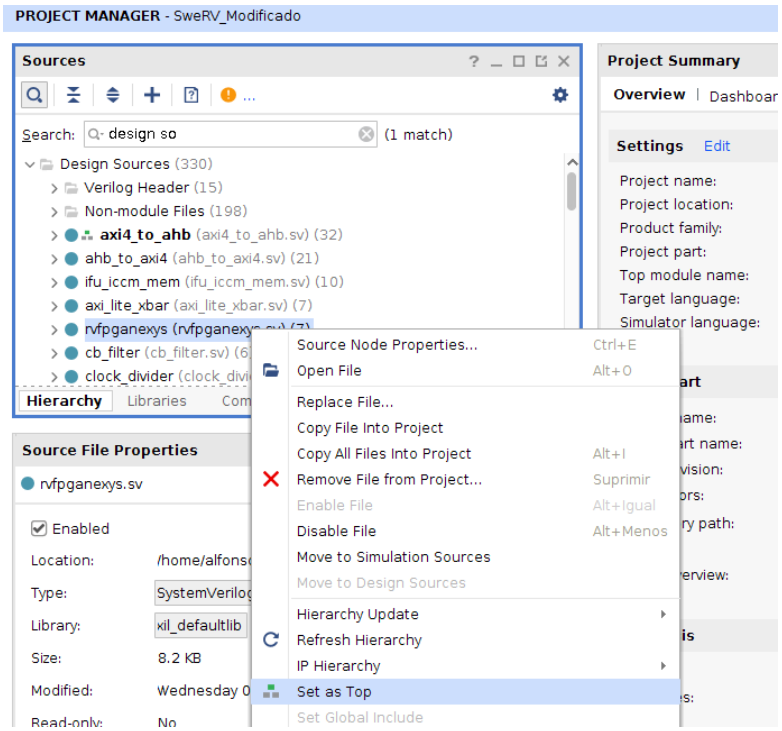

Figura 36: Configuración del módulo *rvfpganexys* como principal en Vivado

6. El siguiente paso a realizar es marcar el fichero *common_defines.vh* como un *Global Include*. Este se encuentra en el desplegable *Non-module Files* y, tras seleccionarlo, hay que activar la susodicha casilla (figura 37).

7. A continuación hay que añadir el fichero *boot_main.mem* al proyecto. En el menú de la izquierda, se hace click en *Add Sources*, y en la ventana que aparece en *Add Files*. Se ha de añadir el fichero */ruta_a_rvfpga/RVfpga/src/SweRVolfSoC/BootROM/sw/boot_main.mem* (figura 38).

8. Finalmente, se incluyen dos carpetas para la plataforma Pulp. En el menú de la izquierda, se selecciona *Settings*. En la ventana que aparece, se hace click en *General*, y a continuación en el cuadro que aparece a la derecha de*Verilog options*. En esta pestaña, se añaden los siguientes directorios: */ruta_a_rvfpga/RVfpga/src/SweRVolfSoC/Interconnect /AxiInterconnect/pulp-platform.org__axi_0.25.0/include*

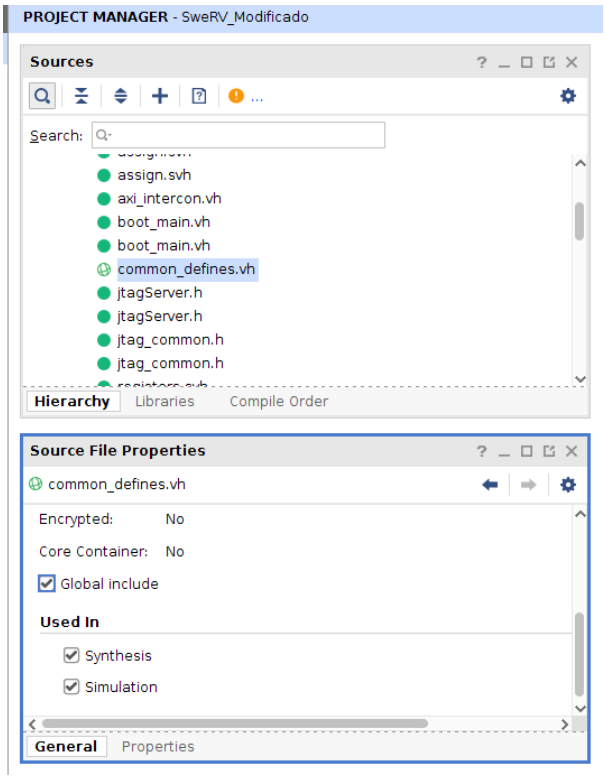

Figura 37: Selección del fichero *common_defines.vh* como *Global Include*

/ruta_a_rvfpga/RVfpga/src/OtherSources/pulp-platform.org__common_cells_1.20.0/include (figura 39)

9. Tras configurar el proyecto, se puede generar el bitstream en la pestaña *Flow ->Generate Bitstream*. Se dejan todas las opciones por defecto y se acepta. Este proceso puede tardar entre 15 y 25 minutos. Una vez finalizado, en la carpeta en la que se guardó el proyecto se genera el archivo que hará falta para la ejecución en la FPGA. Para un proyecto llamado *SweRV_modificado*, este se encuentra en */SweRV_modificado/SweRV_modificado.runs /impl_1/rvfpganexys.bit* (figura 40).

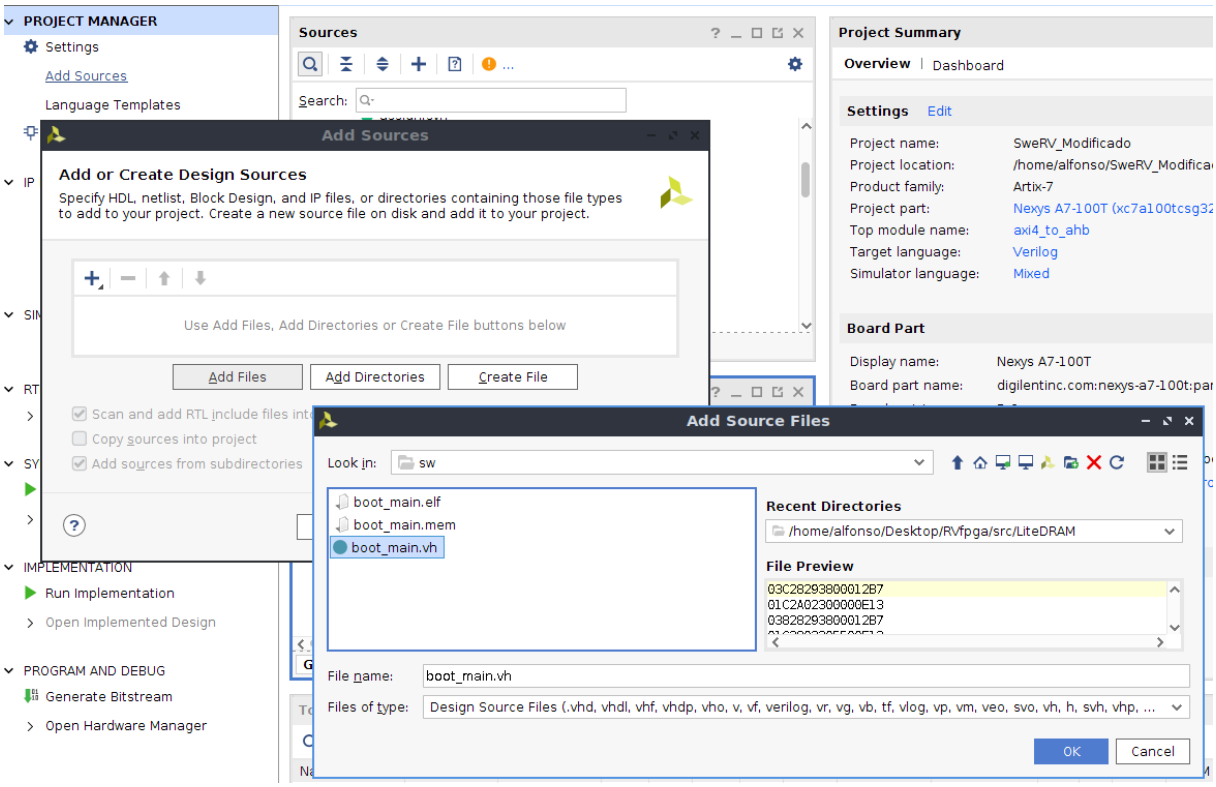

Figura 38: Selección del fichero *boot_main.mem* en Vivado

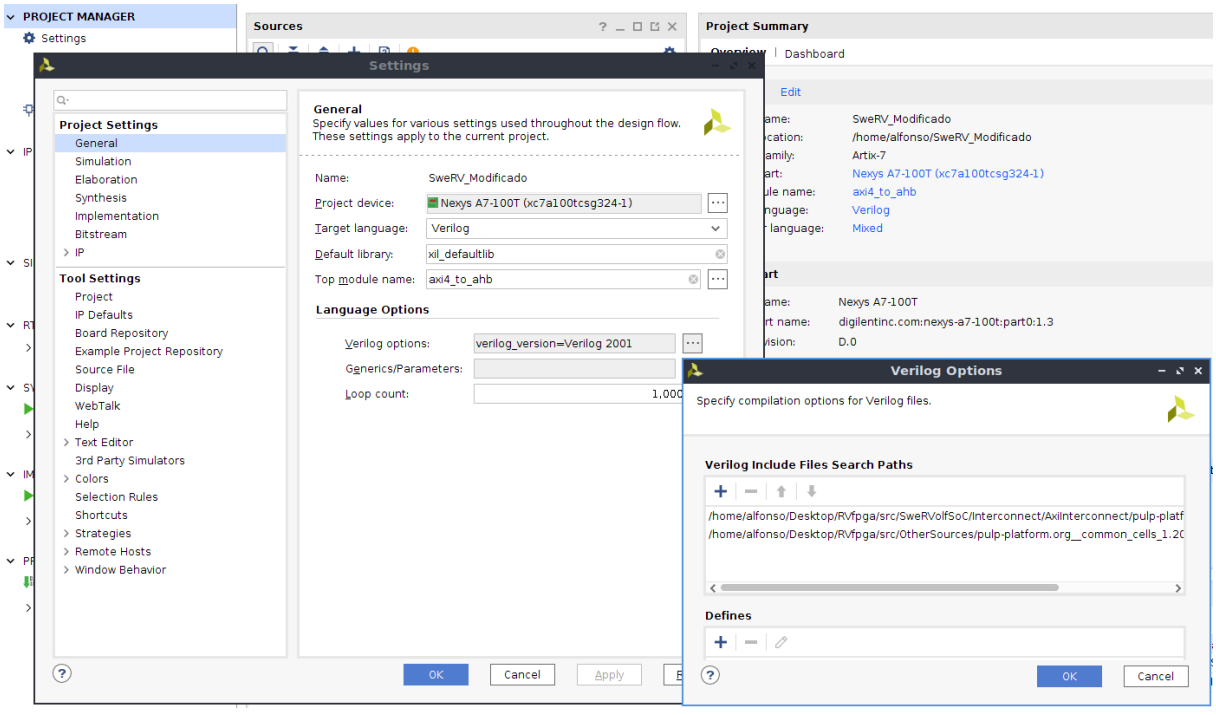

Figura 39: Inclusión de las carpetas Pulp en Vivado

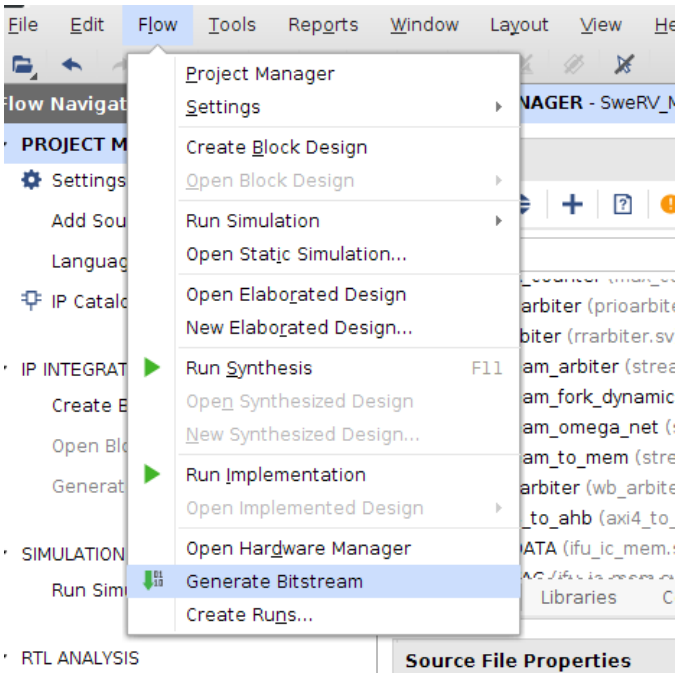

Figura 40: Generación del bitstream en Vivado

A.5.2. Ejecución en la FPGA

La programación de la FPGA se realiza mediante VSCode y PlatformIO, siguiendo los pasos que se muestran a continuación:

1. Con la Nexys A7 encendida y conectada al ordenador, se abre VSCode y PlatformIO.

2. De igual manera que como se hizo en el apartado A.4, hay que abrir un proyecto de PlatformIO.

3. En el fichero *Platformio.ini*, a la variable *board_build.bitstream_file* se le asigna la ruta completa al binario generado con Vivado.

```
[env:swervolf_nexys]
platform = chipsalliance
board = swervolf_nexys
framework = wd-riscv-sdk

monitor_speed = 115200
#debug_tool = whisper

board_build.bitstream_file = /home/alfonso/SweRV_Modificado/SweRV_Modificado.runs/impl_1/rvfpganexys.bit

board_debug.verilator.binary = /home/alfonso/Desktop/RVfpga/verilatorSIM/Vrvfpgasim
```

Figura 41: Modificación del fichero *platform.ini* para referenciar el binario *rvfpganexys.bit*

4. En el menú de PlatformIO, se hace click en *Upload Bitstream*. De esta forma, se carga en la FPGA el SweRV EH1.

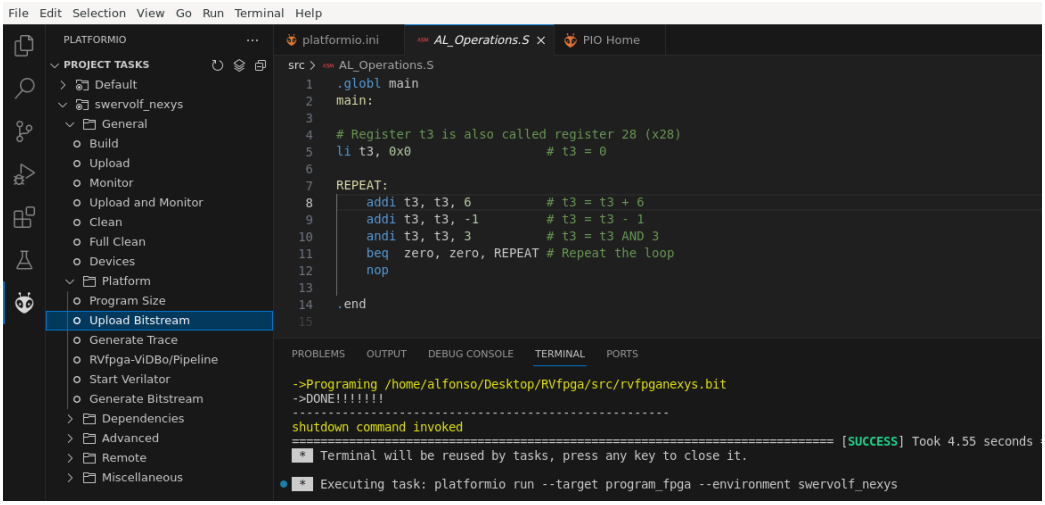

Figura 42: Carga del bitstream en la FPGA

5. Finalmente, al acceder a la pestaña *Run and Debug* de VSCode, se puede ejecutar el código del proyecto que esté abierto haciendo click en el icono verde que aparece en la parte superior izquierda. Si hace falta conectarse por puerto serie a la Nexys A7 para visualizar la salida del programa, se puede hacer pulsando el botón de *Serial Monitor*, que se encuentra en la esquina superior derecha.

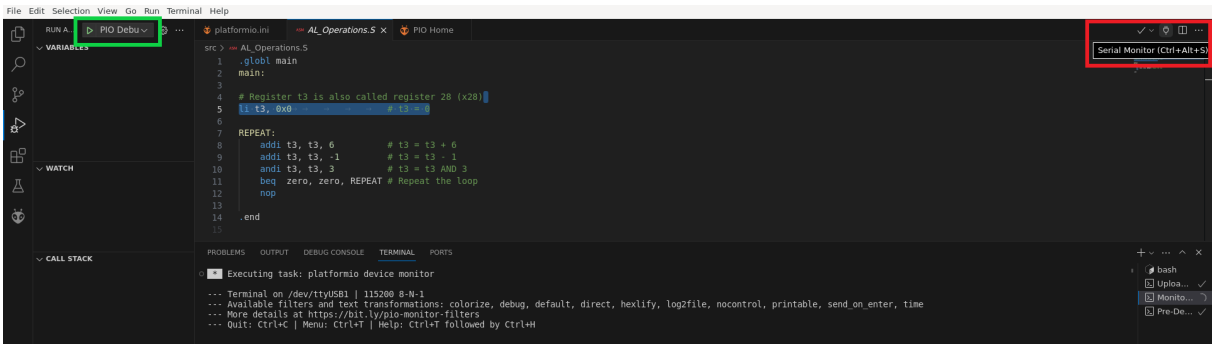

Figura 43: Ejecución en la FPGA y comunicación serie con esta

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA