Extensiones de punto flotante para el core SweRV EH1

Floating-point extensions for the SweRV EH1 core

Trabajo de Fin de Grado
Curso 2022–2023

Autor

Alejandro Perea Rodríguez

Directores

Alberto Antonio del Barrio García
Daniel Chaver Martínez

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Resumen

Extensiones de punto flotante para el core SweRV EH1

Las operaciones de punto flotante son muy importantes en muchos campos de la computación, como la inteligencia artificial o gráficos 3D. Sin embargo, muchos procesadores no están optimizados para realizar este tipo de operaciones, lo que puede limitar significativamente su rendimiento en ciertas aplicaciones.

En este Trabajo Fin de Grado se lleva a cabo la adición de la unidad de punto flotante FPnew en el System on Chip (SoC) SweRVolf. Este SoC integra el procesador SweRV EH1 que utiliza la arquitectura RISC-V.

Una vez completadas las modificaciones en el procesador, se efectúan una serie de pruebas de verificación para comprobar el correcto funcionamiento de la unidad integrada.

El código desarrollado en este trabajo se puede encontrar en el repositorio de GitHub: `https://github.com/aperea01/TFG-SweRV-EH1-FP`

Palabras clave

Punto flotante, RISC-V, SweRV EH1, Swervolf, SoC, FPnew, *pipeline*.

Abstract

Floating-point extensions for the SweRV EH1 core

Floating point operations are very important in many fields of computing, such as artificial intelligence or 3D graphics. However, many processors are not optimized to perform these types of operations, which can significantly limit their performance in certain applications.

In this Final Degree Project, the addition of the FPnew floating point unit in the SweRVolf System on Chip (SoC) is carried out. This SoC integrates the SweRV EH1 processor using RISC-V architecture.

Once the processor modifications have been completed, a series of verification tests are performed to check the correct operation of the integrated unit.

The code developed in this project can be found on the GitHub repository: `https://github.com/aperea01/TFG-SweRV-EH1-FP`

Keywords

Floating Point, RISC-V, SweRV EH1, Swervolf, SoC, FPnew, pipeline.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo, se presentarán los antecedentes que han motivado la realización de este TFG, los objetivos fijados y el plan de trabajo para alcanzar dichos objetivos.

1.1. Antecedentes

En 1971 se produce el lanzamiento del Intel 4004, el primer microprocesador comercial de la historia. En los años posteriores, Intel lanzó una serie de procesadores, el 8008 y el 8080, que asentaron las bases de la primera generación de computadoras personales.

Hasta ese momento, todos los procesadores que surgieron usaban la arquitectura CISC (Complex Instruction Set Computer) u otras basadas en ella, como por ejemplo: Intel 8086 y 8088 (arquitectura x86), Motorola 68000, entre otros. Pero a medida que mejoraban en términos de rendimiento y cómputo, también se volvían más complejos y menos eficientes energéticamente. Lo que propició el nacimiento de la arquitectura RISC (Reduced instruction set computing), que destacaba por la eliminación de instrucciones complejas, permitiendo hacer unos procesadores más simples y eficientes. Algunos de los procesadores que surgieron son: MIPS R2000, IBM POWER1, DEC Alpha, etc.

La falta de una arquitectura de código abierto y libre de patentes (todas las arquitecturas hasta ese momento estaban sujetas a patentes muy costosas), llevó a la creación de RISC-V en 2010.

Otro de los problemas que surgieron con los primeros procesadores, fue la necesidad de poder representar números reales muy grandes o pequeños de una manera fácil y eficiente. La primera solución ideada fue en 1914 por Torres y Quevedo, quien describió el primer concepto de representación en punto flotante, pero no fue hasta 1938 que se creó la primera máquina que lo incluía, el Z1. En los años posteriores,

todas las máquinas hacían que las operaciones de números reales se llevaran a cabo a nivel de software o con hardware adicional. No fue hasta la década de los 80, que se empezó a incluir unidades de punto flotante en los procesadores para uso doméstico. Esto llevó a que en 1985, el Instituto de Ingenieros Eléctricos estableciera un estándar llamado IEEE 754 que estaba fundamentado en gran medida por el Intel 8087 creado por Kahan et al. en 1980. Este estándar, tenía como fin poner una serie de normas sobre cómo representar números en punto flotante, ya que cada empresa tenía sus propias pautas para definir estos números, lo que traía incompatibilidades y problemas.

Respecto al procesador que se usará en el proyecto, el SweRV EH1 es un procesador de arquitectura de código abierto (RISC-V) desarrollado por Western Digital en el año 2019. Aunque es un procesador potente y eficiente en términos de energía, carece de soporte extensiones de punto flotante, lo que limita su capacidad de cálculo en situaciones en las que se requieren operaciones precisas y complejas.

Actualmente, existe una gran variedad de unidades de punto flotante desarrolladas por la comunidad de forma abierta que pueden ser implementadas en procesadores RISC-V. En este trabajo, se ha optado por implementar la unidad FPnew[7] de Stefan Mach sobre el procesador SweRV EH1.

1.2. Objetivos

El objetivo principal de este trabajo es añadir soporte para números en punto flotante mediante las extensiones de D y F en el procesador SweRV EH1. Para ello se utiliza el SoC SwervolfX desarrollado por Imagination University Programme, una adaptación del SoC original Swervolf de Chips Alliance. Para llevar a cabo el aprendizaje y entendimiento del procesador se usa el curso RVfpga proporcionado, también, por Imagination University Programme.

Los objetivos para efectuar la implementación de la unidad de punto flotante en el procesador son los siguientes:

- No alterar el correcto funcionamiento del SoC al integrar la unidad.
- Ampliar la funcionalidad del procesador. Al soportar las operaciones básicas (suma, resta, multiplicación y división) en punto flotante, el procesador mejora el rendimiento en tareas que requieran una gran capacidad de cómputo.
- El SoC ha de funcionar en la placa Nexys 4 DDR.

1.3. Plan de trabajo

Tras mantener la primera reunión con los directores, se acuerdan las bases sobre las que empezar el desarrollo del proyecto. Tras esto, se divide el proyecto en varias fases:

- Preparación. Se solicita acceso al curso RVfpga de Imagination University Programme. Se lee la documentación inicial y se instalan los entornos de desarrollo recomendados.
- Estudio del SoC SwervolfX. El curso contiene veinte prácticas orientadas al aprendizaje sobre el SoC y el procesador SweRV EH1. Se realizan algunas de las prácticas que pueden ayudar a la hora de realizar los cambios necesarios para el trabajo. Entre ellos, destaca la práctica 18, que relata cómo implementar una unidad de punto flotante básica usando el *pipe* del divisor.
- Integración de la unidad básica. Se implementa la unidad proporcionada en el curso creando un nuevo *pipe* exclusivo para la unidad de punto flotante.
- Estudio de la unidad FPnew. Se lee la documentación para comprender el funcionamiento y los protocolos de comunicación.
- Integración de la unidad FPnew. Se usa la integración creada anteriormente y se adapta la nueva unidad.

- Depuración de la integración. Se realizan pruebas básicas para detectar y corregir errores.
- Pruebas sobre la placa Nexys 4 DDR. Se comprueba que el funcionamiento es igual que al obtenido en simulación.
- Verificación. Se ejecutan los test oficiales de la fundación RISC-V con el fin de asegurar que el funcionamiento es el correcto en todas las situaciones posibles.
- Evaluación de la implementación por medio del Método de la Bisección para buscar las raíces de una función.
- Redacción. Se redacta la presente memoria.

Capítulo 2

Estado del arte

2.1. RISC-V

A finales de la década de los 70, los procesadores CISC (Complex Instruction Set Computing) estaban a la orden del día. Estos procesadores destacaban por poder realizar múltiples operaciones en una sola instrucción debido al conjunto de instrucciones complejas. Pero esto, tenía la desventaja de que hacía que los procesadores fueran bastante complejos tanto a nivel de hardware como de programación y depuración. Para solucionarlo, surgió en la Universidad de Berkeley un proyecto llamado RISC (*Reduced Instruction Set Computer*) dirigido por David A. Patterson. En 1981, Patterson y su equipo presentaron el primer procesador llamado RISC I.

Mientras tanto, en la Universidad de Stanford, se inicia un proyecto similar liderado por John L. Hennessy llamado MIPS (*Microprocessor without Interlocked Pipeline Stages*), otra arquitectura RISC que se empezó a desarrollar en 1981. Años más tarde, en 1984, decidieron crear *MIPS Computer Systems* para comercializar sus propios procesadores usando su arquitectura.

Tras esto, empezaron a surgir multitud de arquitecturas basadas en RISC, como ARM, PowerPC, SPARC, DEC Alpha, etc. El problema de todas estas arquitecturas, es que estaban protegidas por patentes. Para resolverlo, en 2010 se inicia un proyecto en la universidad de Berkeley dirigido por Krste Asanović llamado RISC-V. En el proyecto colaboró David Patterson.
RISC-V es una arquitectura de conjunto de instrucciones (ISA) de código abierto con licencia BSD (Berkeley Software Distribution) y basada en RISC. Nació con la premisa de crear una arquitectura que pudiera ser utilizada en el mundo académico y profesional sin restricciones de propiedad intelectual, para que cualquier persona pudiera implementar y mejorar la arquitectura sin tener que pagar patentes. Además, se quería crear una arquitectura modular que permitiese agregar solo los bloques de instrucciones que se necesitasen, para así lograr unas implementaciones más sencillas y eficientes.

Además, *RISC-V International* es una fundación sin ánimo de lucro que cuenta con la colaboración de importantes empresas como Google, Huawei, IBM, Western Digital, entre otras. Diversas universidades de renombre como la Universidad de California, la Universidad de Cambridge o la Universidad Complutense de Madrid también forman parte de esta fundación. En concreto, esta última ha contribuido con proyectos destacados, como Percival[8] desarrollado por el grupo Artecs o RVfpga[9] de Imagination University Programme.

Como se ha destacado antes, el diseño modular de RISC-V es una de las grandes novedades que posee. La arquitectura consta de varios repertorios base (ISA base) y de un gran número extensiones que permiten agregar características opcionales para adaptarla a diferentes necesidades.

En la siguiente tabla se muestran las bases ISA disponibles:

Nombre	Descripción
RV32I	Conjunto de instrucciones base con enteros de 32 bits.
RV32E	Conjunto de instrucciones base con enteros de 32 bits para sistemas empotrados, consta de 16 registros.
RV64I	Conjunto de instrucciones base con enteros de 64 bits.
RV128I	Conjunto de instrucciones base con enteros de 128 bits.

Tabla 2.1: ISA base de RISC-V[1]

La base RV32I, tiene bastante relevancia en este trabajo ya que es la usada por el procesador SweRV EH1. Esta base, como se puede leer en la tabla 2.1, es para enteros de 32 bits y además dispone de 47 instrucciones. Adicionalmente, incluye 32 registros de propósito general. Estos registros van desde x0 a x31, aunque el registro x0 es un registro especial de no escritura que alberga el valor 0. También existe otro registro especial llamado *PC* (Contador de programa), que contiene la siguiente instrucción a ejecutar. Este registro, al igual que el x0, también es de solo lectura. En la figura 2.1 se muestran los registros nombrados.

En la tabla 2.2 se muestran algunas de las extensiones disponibles. Estas extensiones son un conjunto de características opcionales para añadir a la arquitectura base y personalizarla acorde a la necesidades. Están preparadas para funcionar entre sí y con cualquiera de las bases.

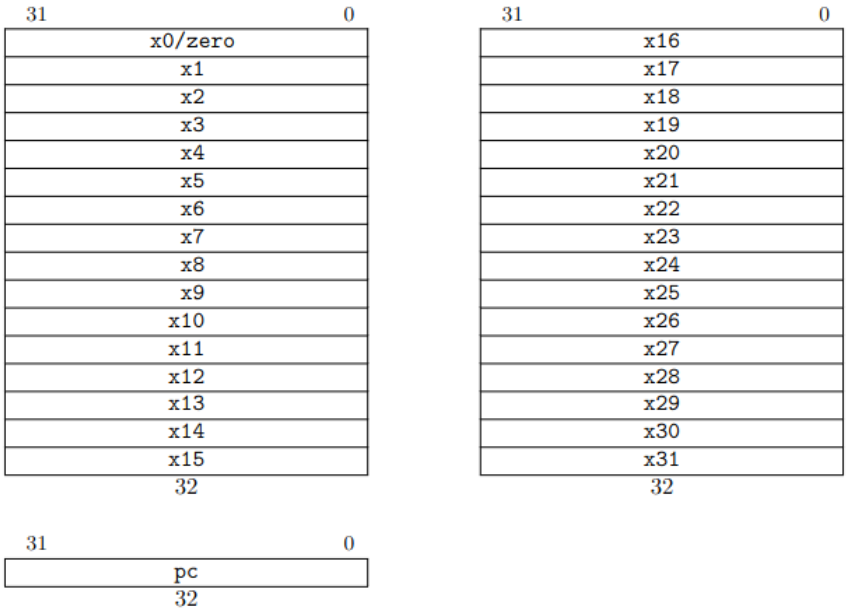

Figura 2.1: Registros correspondientes a la base RV32I[12]

Nombre	Descripción
M	Multiplicación y división de enteros.
A	Instrucciones atómicas.
F	Punto flotante de precisión simple.
D	Punto flotante de precisión doble.
Q	Punto flotante de precisión cuádruple.
L	Punto flotante decimal.
C	Instrucciones comprimidas.
B	Manipulación de bits.
J	Lenguajes traducidos dinámicamente.
T	Memoria Transaccional.
P	Instrucciones Packed-SIMD.
V	Operaciones vectoriales.
N	Interrupciones a nivel de usuario.
Zfinx	Punto flotante de precisión simple en registros de enteros.
Zdinx	Punto flotante de precisión doble en registros de enteros.
Zhinx	Punto flotante de precisión media en registros de enteros.
Zhinxmin	Punto flotante de mínima precisión media de 16 bits en registros de enteros.

Tabla 2.2: Extensiones de RISC-V[1;2]

RISC-V dispone de seis tipos de instrucciones diferentes. Estos tipos son: R, I, S, U, SB y UJ. Cada uno dispone de un formato distinto, aunque la mayoría disponen

de *rs1* y *rs2*, que representan los registros de entrada, y *rd*, que es el registro de salida. También se utilizan inmediatos en algunos tipos de instrucciones para realizar operaciones. Todas disponen del campo *opcode* para indicar el tipo de instrucción. En la figura 2.2 se muestran los formatos para cada uno.

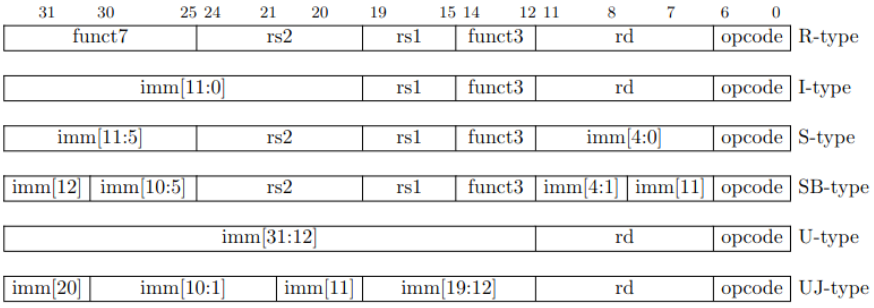

Figura 2.2: Formatos de cada tipo de instrucción en RISC-V. [12]

2.2. SweRV EH1

Western Digital ha desarrollado tres procesadores RISC-V en los últimos años: SweRV EH1, SweRV EH2 y SweRV EL2. Cada uno con diferentes características que se muestran en la siguiente tabla.

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 2.3: Características de los procesadores SweRV [9]

En este proyecto se usará el SweRV EH1, que tiene licencia Apache 2.0. Este procesador, usa la base ISA RV32I (Conjunto de instrucciones base con enteros de 32 bits) junto con las extensiones M (multiplicación y división de enteros) y C (instrucciones comprimidas).

Es un procesador superescalar con una ruta de datos de 9 etapas, 2 vías de ejecución y una arquitectura de 32 bits (ver la figura 2.4). También dispone de 4 *pipes*, I0 e I1 destinados a realizar operaciones aritmético-lógicas y saltos, otro para procesar instrucciones de *load* y *store* y un último para instrucciones de multiplicación. Además cuenta con un divisor no segmentado de 34 ciclos.
Existen cuatro *stall points* situados en las etapas de *Fetch 1*, *Align*, *Decode* y *Commit*.

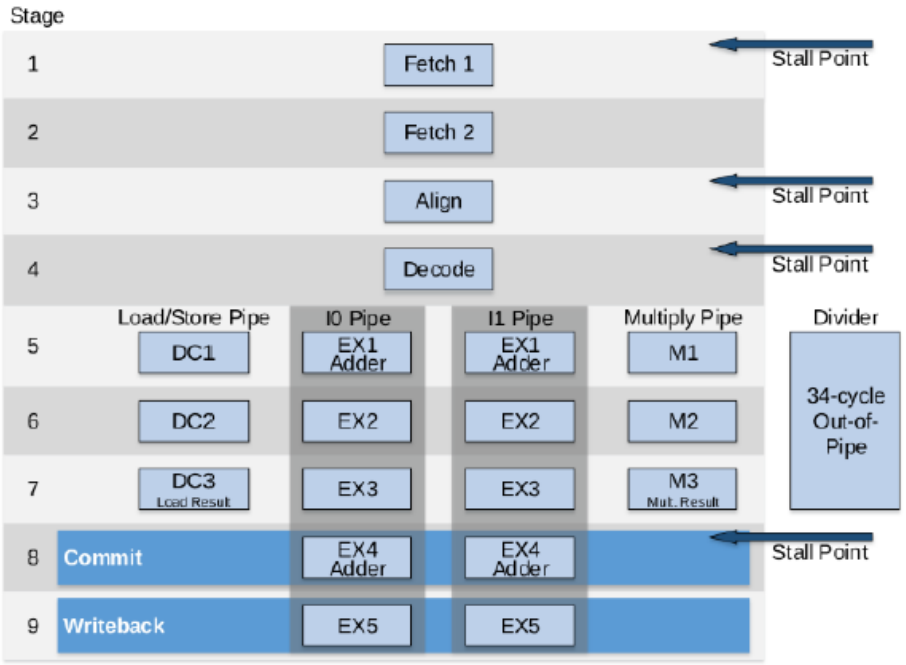

Figura 2.4: Etapas del procesador SweRV EH1 [9]

En este TFG, se estudiarán en profundidad las etapas de decodificación (*decode*) y ejecución (*execution*), que serán fundamentales a la hora de agregar las nuevas instrucciones de punto flotante y de integrar la unidad FPnew en el SweRV.

Existe una versión extendida del SweRV EH1 desarrollado también por Western Digital, llamado SweRV EH1 Core Complex. Esta extensión añade las siguientes características (ver la figura 2.5) al procesador:

- Dos memorias dedicadas muy cercanas al procesador, lo que proporciona un muy bajo tiempo de acceso. Una de las memorias es para instrucciones (ICCM) y la otra para datos (DCCM). Cada una puede ser configurada con hasta 512 KB de almacenamiento.
- Una caché de instrucciones opcional asociativa de 4 vías que dispone de protección de paridad o ECC.
- Un PIC programable opcional que puede manejar hasta 255 interrupciones externas.
- Cuatro interfaces de bus del sistema para el acceso a instrucciones (Maestro de bus IFU), acceso a datos (Maestro de bus LSU), acceso a depuración (Maestro de bus de depuración) y acceso a DMA externo (Puerto esclavo de DMA).
- Una unidad de depuración central.

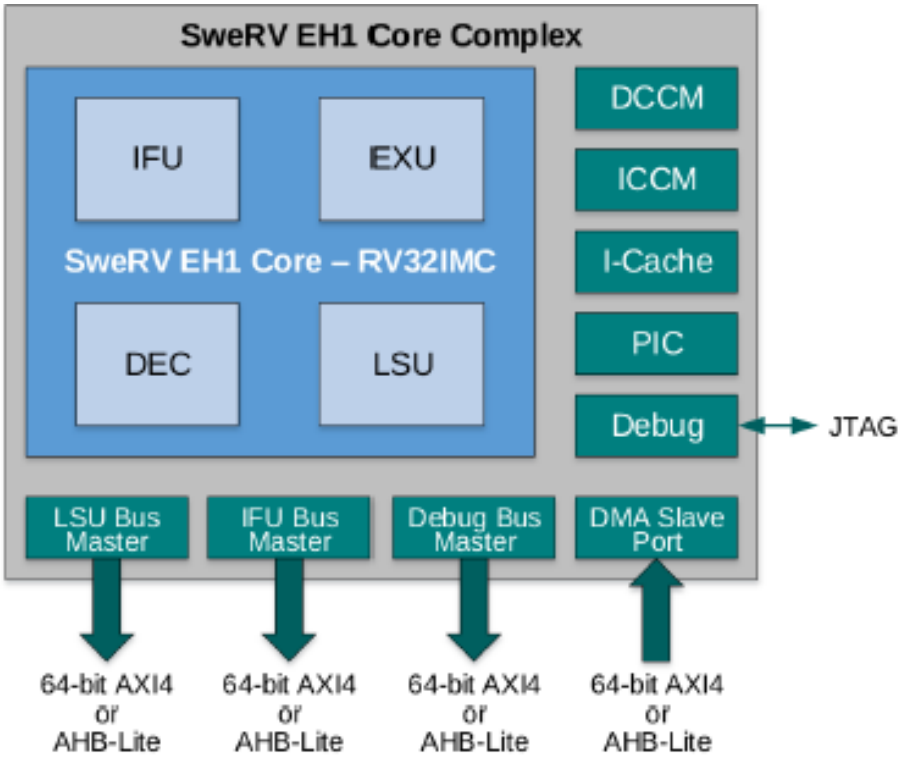

Figura 2.5: Etapas del procesador SweRV EH1 [9]

2.2.1. SweRVolf

El SweRVolf es un SoC desarrollado por Olof Kindgren en el marco de la iniciativa Chips Alliance, una organización de código abierto que promueve la colaboración en el diseño de chips. Este SoC utiliza el SweRV EH1 Core Complex, además añade (ver la figura 2.6):

- Una memoria de arranque (*Boot ROM*).
- Un controlador de memoria.
- Un controlador UART.
- Un controlador de sistema.
- Un controlador SPI.
- Bus AXI (Advanced eXtensible Interface) utilizado por el procesador, mientras que los periféricos usan bus Wishbone. El SoC dispone de un Bridge AXI-Wishbone para permitir la comunicación entre ambos buses.

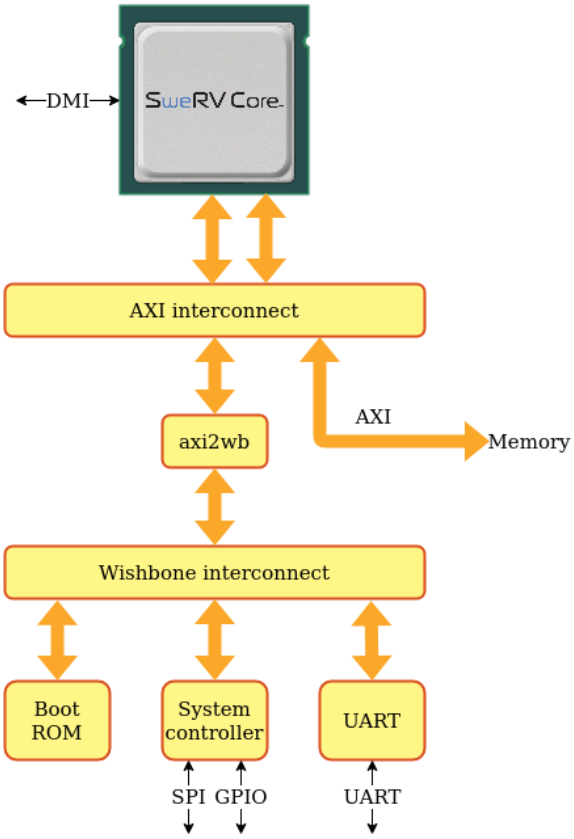

Figura 2.6: SoC SweRVolf[3]

Aunque para este proyecto, se va a usar el SoC SweRVolfX que incluye alguna funcionalidad más. Está desarrollado por Imagination University Programme para el curso RVfpga[9]. Esta versión modificada del SoC dispone de los siguientes nuevos periféricos:

- Un segundo controlador SPI.
- Un controlador Entrada/Salida de Propósito General (GPIO).
- Un módulo PTC (PWM/Temporizador/Contador) .
- Un controlador para *display* de 7 segmentos.

A continuación, en la figura 2.7, se muestran en rojo las nuevas funcionalidades (excepto el controlador 7 segmentos) que incluye el SweRVolfX.

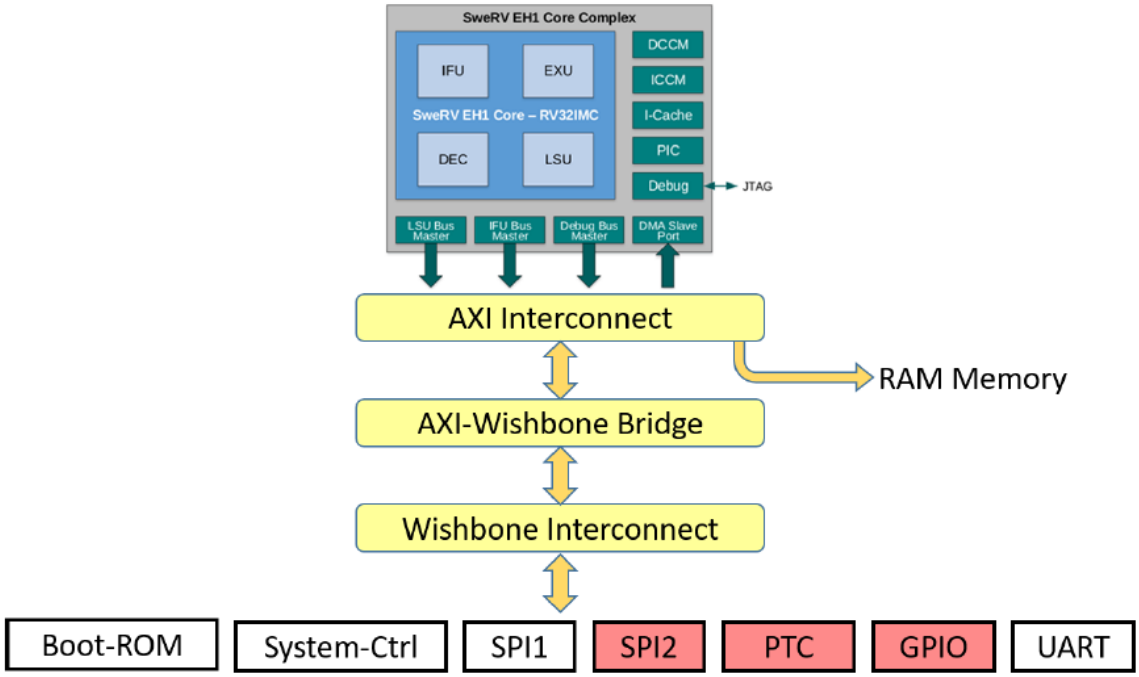

Figura 2.7: SoC SweRVolfX[9]

2.3. Punto flotante

El punto flotante es una forma eficiente de representar números reales, ya sean muy grandes o muy pequeños, en un computador. Es especialmente útil en campos como la ingeniería o la ciencia, donde es muy importante la precisión en los cálculos.

En el año 1914, el inventor y científico español, Leonardo Torres y Quevedo, propuso el primer concepto de punto flotante. Aunque no fue hasta el año 1938, cuando Konrad Zuse diseñó el Z1, la primera computadora mecánica programable binaria que utilizaba punto flotante. Más tarde, en 1941, saldría el Z3, que disponía de representación para números infinitos. Pero no fue hasta 1945, cuando estuvo disponible el Z4, el primer computador comercial con hardware de punto flotante.

Años después, IBM sacaría el primer ordenador comercial de gran escala que incluía hardware de punto flotante. Se trataba del IBM 704, introducido en el año 1954. Tras esto, empezaron a surgir computadores como el IBM 7094 y la serie UNIVAC 1100/2200, que disponían de dos representaciones de punto flotante: precisión simple y precisión doble.

El problema de estas representaciones en los dispositivos nombrados, era que no seguían el mismo formato de representación de punto flotante. Este dilema siguió presente en todos los computadores creados hasta 1985, cuando el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, por sus siglas en inglés) publicó el estándar IEEE 754, una norma para representar números en punto flotante en sistemas informáticos. El estándar también establece cómo representar las operaciones

aritméticas y lógicas y define dos formatos de punto flotante: precisión simple (32 bits) y precisión doble (64 bits).

El IEEE 754 ha sido revisado varias veces desde su salida para mejorar la precisión y eficiencia, la última gran revisión fue en el año 2008 (hubo una revisión menor en el 2019), en la que se añadió el formato de precisión media (16 bits) y el de precisión cuádruple (128 bits). Este estándar representó un gran avance en la representación y procesamiento de números en punto flotante y sigue teniendo una gran importancia, al ser el más utilizado en los sistemas actuales.

La representación de números en punto flotante se basa en el uso de un signo, una mantisa y un exponente. El signo sirve para indicar si un número es positivo o negativo, la mantisa representa la magnitud del número y el exponente determina la posición del punto decimal (o binario) en relación a la mantisa. Así se consigue representar una amplia variedad de números enteros.

Como se ha explicado anteriormente, en el estándar IEEE 754 se establece cómo representar números en punto flotante de precisión simple (32 bits) y doble (64 bits). En el formato de precisión simple, el primer bit se utiliza para indicar el signo del número (0 para positivo y 1 para negativo), los siguientes 8 bits para el exponente y los últimos 23 bits para la mantisa. En el formato de precisión doble, el primer bit indica el signo, los siguientes 11 bits el exponente y los últimos 52 bits la mantisa.

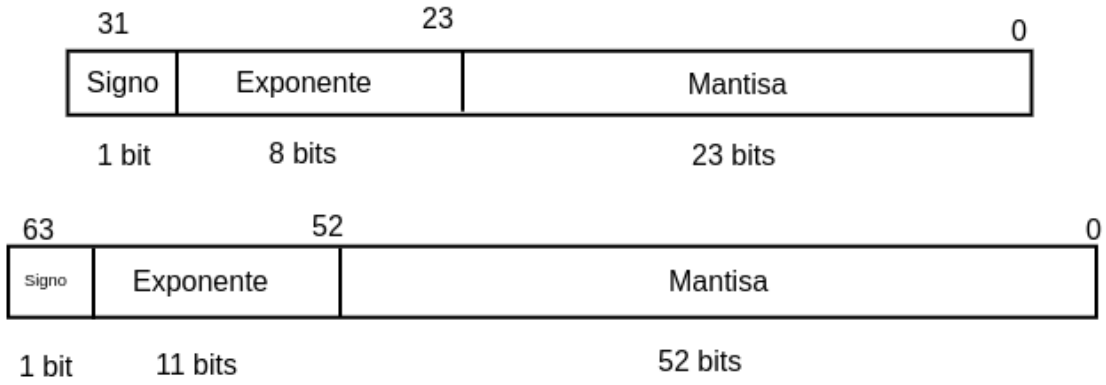

Figura 2.8: Precisión simple y doble en punto flotante. (Elaboración propia)

Uno de los problemas a tener en cuenta con los números en punto flotante es la falta de precisión debido a la limitada capacidad de representación. Esto se debe a que los números en punto flotante tienen una anchura fija y finita, lo que implica que los números reales muy grandes o muy pequeños no pueden ser representados de manera exacta. Una solución a este problema es utilizar diferentes tipos de redondeo [11], para ello el estándar IEEE 754 define cinco modos:

- *roundTiesToEven*: Se redondea al número de punto flotante más cercano. Si el

resultado se encuentra exactamente entremedias de dos números, se redondea al número par más cercano.

- *roundTiesToAway*: Se redondea también al número de punto flotante más cercano. Si el resultado se encuentra exactamente entremedias de dos números, se elige el número con la mayor magnitud.
- *roundTowardPositive*: Se redondea al número más cercano que sea igual o mayor que el valor exacto.
- *roundTowardNegative*: Se redondea al número más cercano que sea igual o menor que el valor exacto.
- *roundTowardZero*: Se redondea al número más cercano y no mayor en magnitud que el valor exacto.

Pese a que el estándar IEEE-754 es el utilizado de forma nativa en los computadores actuales, en los últimos años ha surgido un interés creciente por parte de la comunidad científica en buscar alternativas de representación. Estas alternativas tienen como objetivo abordar diferentes aspectos, como la disminución de la anchura de los datos para optimizar el rendimiento en áreas como la inteligencia artificial, donde se han propuesto formatos como bfloat16 de Google o FP8 de NVIDIA. También se ha enfocado la búsqueda en mitigar los problemas inherentes al IEEE-754, como la falta de asociatividad en las operaciones aritméticas. Por ello, surgen los números Posits[6], propuestos por John Gustafson. Éstos ofrecen mayor precisión cerca del cero, mayor capacidad de representación y una operación de redondeo más consistente.

2.3.1. Unidad de punto flotante FPnew

La unidad de punto flotante FPnew[7], es una unidad diseñada en SystemVerilog mantenida por OpenHW Group que utiliza el procesador RISC-V llamado RI5CY. FPnew presenta varias unidades funcionales que pueden estar segmentadas o utilizar una implementación bloqueante. Esta unidad paramétrica, soporta los formatos y operaciones estándar de RISC-V, así como formatos de transprecisión, manteniendo siempre la eficiencia. La unidad está diseñada para cumplir con el IEEE 754-2008, por lo que soporta cualquier formato incluido en el estándar como: precisión simple, doble precisión, cuádruple precisión y precisión media. Además, la unidad soporta múltiples formatos concurrentemente.

FPnew es compatible con una alta variedad de operaciones, entre las que se encuentran:

- Suma y resta.
- Multiplicación y división.

- Operaciones FMA, que son las operaciones de multiplicación y acumulación fusionadas en una sola operación. Hay cuatro variantes de este tipo de instrucción: *fmadd* (rs1 * rs2 + rs3), *fmsub* (rs1 * rs2 - rs3), *fnmadd* (-rs1 * rs2 + rs3), *fnmsub* (-rs1 * rs2 - rs3).
- Raíz cuadrada.
- Mínimo/Máximo.
- Comparaciones.
- Operaciones de manipulación de signo (*copy*, *abs*, *negate*, *copySign*).
- Conversiones entre formatos.
- Clasificación.

La unidad además soporta los cinco modos de redondeo indicados en el IEEE-754 de 2008; así como los *flags* de estado correspondientes, que son:

- Operación inválida (NV).
- División por cero (DZ).
- Desbordamiento (OF).
- Subdesbordamiento (UF).
- Inexactitud (NX).

Capítulo 3

Adaptación de la unidad en el SweRV EH1

3.1. Herramientas utilizadas

Como se relata en el plan de trabajo, tras acceder al curso RVfpga, es necesario instalar los entornos para el desarrollo de las prácticas. Las herramientas son:

- Vivado: Es un programa producido por Xilinx para diseño y desarrollo de sistemas hardware usando lenguajes de descripción de hardware (HDL). En este proyecto se usará para generar el bitstream que se vuelca en la placa Nexys 4 DDR.
- Visual Studio Code: Editor de código gratuito y multiplataforma.
- PlatformIO: Es una extensión para Visual Studio Code. PlatformIO es un IDE de código abierto para el desarrollo de sistemas empotrados. En este TFG se utiliza para realizar diversas tareas que incluyen, escribir el código ensamblador, compilar, cargar el bitstream en la FPGA y realizar la depuración paso a paso.
- RISC-V *Toolchain*: Son un conjunto de herramientas de compilación para la arquitectura RISC-V. Se encuentra incluida en el entorno de PlatformIO por medio de la plataforma ChipsAlliance.
- Verilator: Es una herramienta capaz de convertir código Verilog en un archivo de simulación.
- GTKWave: Permite visualizar y analizar las señales de un archivo de simulación. En este trabajo, interpreta los archivos generados por Verilator.
- Sigasi Studio: Es un IDE para desarrollo de diseños hardware en VHDL y Verilog.

3.2. Estudio del SweRV EH1

Dado que para añadir la unidad de punto flotante hay que modificar el procesador, se estudia el SweRV EH1 usando tres prácticas de las veinte que incluye el curso RVfpga.

3.2.1. Práctica 11 y 12

La práctica 11 se divide en dos secciones, una para la descripción de la organización RTL en Verilog y detalles de cada etapa del *pipeline*, y la otra sobre cómo utilizar contadores de rendimiento. La parte realmente importante para este trabajo es la primera. En cuanto a la práctica 12, se enseña en detalle cómo se ejecuta la instrucción *add* por las diferentes etapas.

En la figura 3.1 se muestra la jerarquía de los principales módulos Verilog del procesador SweRV EH1. El módulo *mem* contiene la jerarquía de memoria, mientras que en el módulo *swerv*, se pueden encontrar todos los módulos que componen el procesador, incluyendo la unidad de búsqueda de instrucciones (IFU), decodificación (DEC), ejecución (EXU), *load/store* (LSU), etc. Los módulos se encuentran en el directorio *src/SweRVolfSoC/SweRVEh1CoreComplex*.

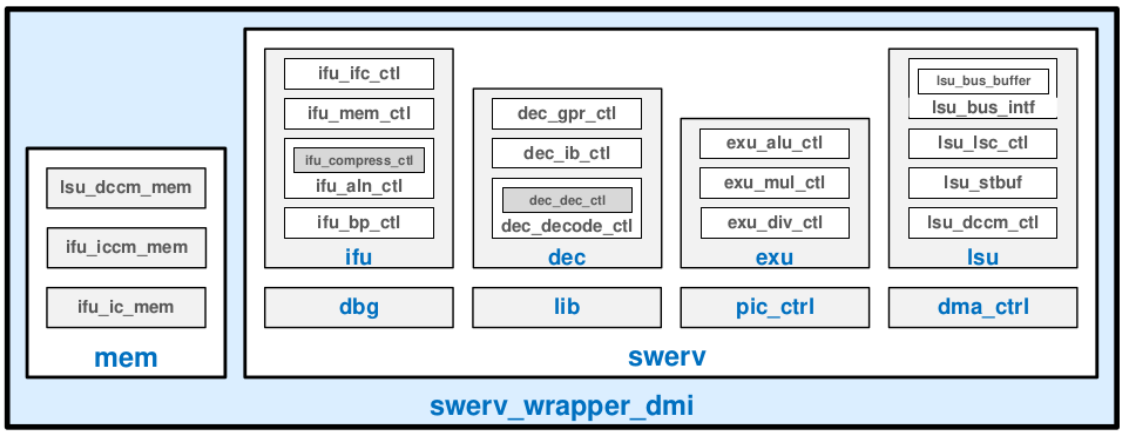

Figura 3.1: Módulos principales del procesador SweRV EH1 [9]

En cuanto a las etapas del procesador por las que pasa una instrucción, son:

- **Fetch (FC1 y FC2) y Align**

  La etapa *Fetch* se encarga de leer las instrucciones de la memoria de instrucciones (formada por una ICCM, una caché de instrucciones y la memoria DDR). En la etapa FC1 se calcula la dirección de la instrucción y se le envía al controlador de memoria. En la segunda etapa Fetch (FC2), se lee la instrucción desde la memoria de instrucciones, en caso de que no esté en caché, se produce un

stall hasta que se traiga de la memoria externa. Si no hay parones, se pueden leer cuatro instrucciones cada dos ciclos, es decir, dos instrucciones por ciclo.

La memoria está configurada de tal forma que la caché de instrucciones dispone de 16 KiB, ICCM de 512 KiB y la memoria externa de 128 MiB. La ICCM está deshabilitada pero se puede activar manualmente.

En cuanto a la etapa de alineamiento (*align*), extrae dos instrucciones de 32 bits por ciclo de los paquetes de 128 bits enviados por la memoria de instrucciones y los asigna a los dos caminos disponibles. Las dos instrucciones, además, son guardadas en los registros de instrucciones.

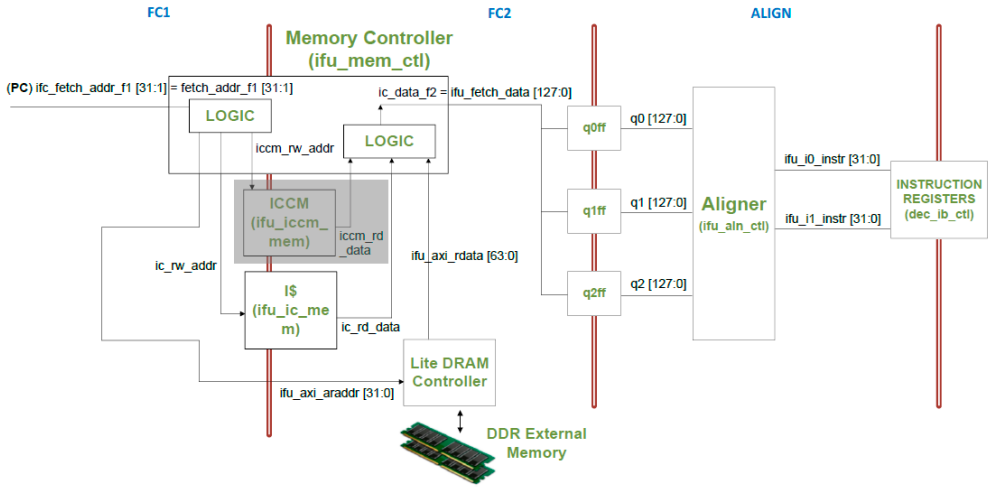

Figura 3.2: Etapas de *Fetch* y *Align* en el SweRV EH1 [9]

- **Decodificación (Decode)**

  Esta etapa es responsable de decodificar las instrucciones, generar las señales de control, extraer los operandos fuente y asignar cada una de las dos instrucciones a un *pipeline*. Estas señales se organizan en estructuras creadas en el archivo *src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv*. Cada estructura está asociada a una unidad específica, ALU (*alu_pkt_t*), multiplicación (*mul_pkt_t*), división (*div_pkt_t*), etc.

  La unidad de control está implementada en el módulo *dec_decode_ctl*, que se encarga de recibir dos instrucciones de 32 bits, decodificarlas y generar las señales. Esta unidad consta de dos etapas (ver la figura 3.3), en la primera, los módulos *i0_dec* e *i1_dec* utilizan las instrucciones para producir señales de control generales, representadas por *i0_dp* e *i1_dp* respectivamente. En la segunda etapa (*decode*), se utilizan estas señales para generar las señales de control específicas para cada unidad de la etapa de ejecución.

  También se proporcionan los operandos utilizando multiplexores 3 a 1 y 4 a 1 para seleccionarlos y facilitarlos a las etapas siguientes mediante registros de *pipeline*.

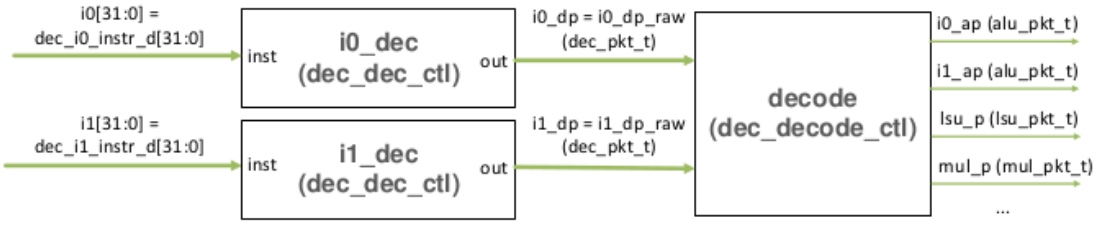

Figura 3.3: Etapa de decodificación en el SweRV EH1[9]

- **Ejecución (Execution)**

  Como se enseña en la figura 3.4, en la etapa de ejecución hay cinco caminos disponibles.

  Los caminos resaltados en azul llamados I0 *pipe* e I1 *pipe*, corresponden a las dos ALU disponibles en el procesador. Se dividen en tres etapas cada uno: EX1 que contiene la ALU y realiza las operaciones de enteros, EX2 y EX3 que son necesarias para sincronizar con otras instrucciones que requieren tres ciclos para ejecutarse.

  El camino en rojo representa el *pipe* que realiza la multiplicación. Se divide en tres etapas (M1, M2 y M3) necesarias para realizar el cálculo.

  El camino de *load/store*, subrayado en verde. También se divide en tres etapas: en DC1 se calcula la dirección, en DC2 las instrucciones de *load* leen en la dirección calculada (si no se encuentra en la DCCM, se produce una parada *stall* hasta que se traiga de la memoria principal) y en DC3 los datos se alinean y fusionan *(align and merge)*. Los datos son guardados en memoria durante varios ciclos, dependiendo de si se deben escribir en caché o en memoria principal. Si es requerida por una instrucción previa, se reenvían los datos.

  La división, resaltada en blanco, no está segmentada. Puede llegar a necesitar hasta 34 ciclos para realizar el cálculo.

  Para seleccionar la salida adecuada de cada *pipe* se usan dos multiplexores 3 a 1.

- **Commit**

  En la etapa de Commit, se dispone de dos multiplexores 3 a 1, uno para *Way-0* y otro para *Way-1*. Seleccionan el resultado a escribir en el banco de registros.

- **Writeback**

  En esta etapa, se escriben los resultados en el banco de registros utilizando los caminos correspondientes (*Way-0* e *Way-1*).

3.2.2. Práctica 18

Es un laboratorio destinado a aplicar los conocimientos adquiridos anteriormente. Para ello, se enseña cómo añadir nuevas instrucciones así como la suma, multipli-

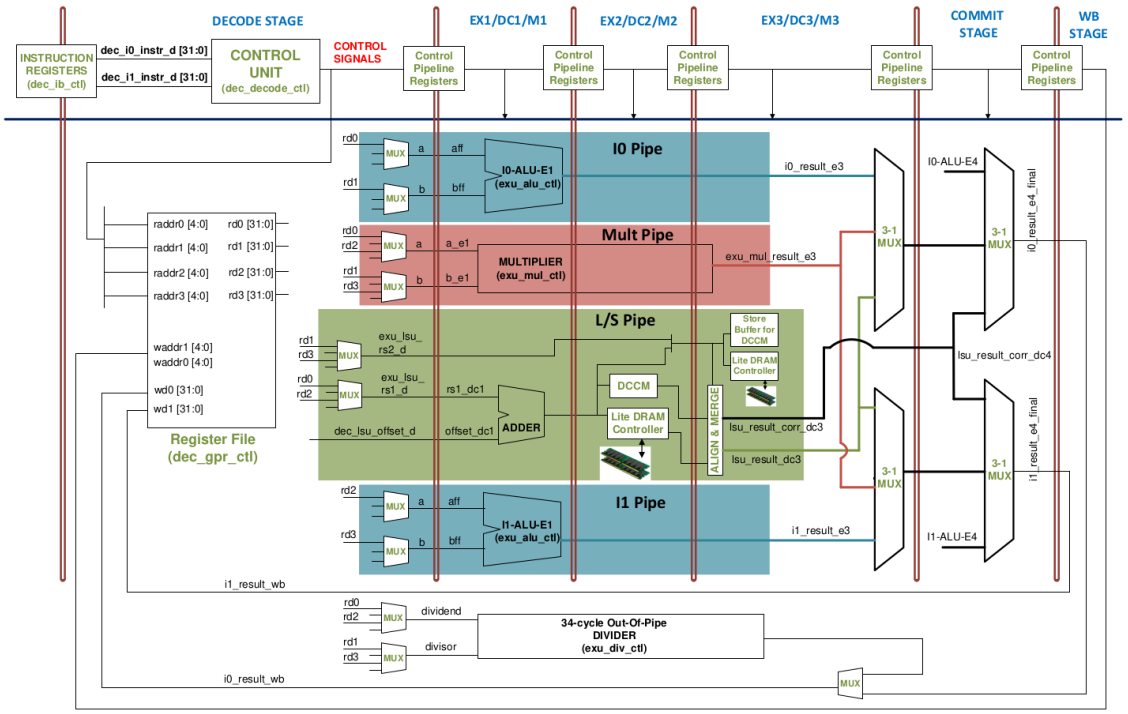

Figura 3.4: Etapas detalladas del SweRV EH1 (No incluye *fetch* y *align*) [9]

cación y división en punto flotante de precisión simple (extensión F), entre otras. Esta práctica es altamente importante para el TFG, debido a que las modificaciones empleadas en la etapa de decodificación para añadir las instrucciones de punto flotante, servirán también a la hora de integrar la unidad FPnew.

Para poder añadir las nuevas instrucciones y poder probarlas, el curso propone integrar una unidad de punto flotante. De esta unidad *fpu* [4] desarrollada por Jonathan P. Dawson, tan solo se usan los módulos de suma, multiplicación y división. La *fpu* se encuentra dividida en módulos, permitiendo implementar solo el módulo que realiza la suma, por ejemplo. Pese a esto, es una unidad que tiene un comportamiento basado en una máquina de estados y requiere de muchos ciclos para realizar las operaciones; mientras que la FPnew es bastante más eficiente en este aspecto, debido al diseño explicado en el capítulo 2.

Para añadir las tres instrucciones, se crea una estructura en el archivo *swerv_types.sv*, llamada *fp_pkt_t*, que incluye los tres bits correspondientes a las nuevas instrucciones:

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

Se incluyen también estos bits en la estructura correspondiente a la decodificación (*dec_pkt_t*), situada en el mismo archivo.

Después, se ha de asignar valores a los nuevos bits en las señales *i0_dp_raw* e *i1_dp_raw*. Para ello, hay que modificar el módulo *dec_dec_ctl* para regenerar las ecuaciones de las instrucciones. Este módulo está incluido en el archivo *dec_decode_ctl* (ubicado en el directorio *src/SweRVolfSoc/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*). En el archivo *decode*, localizado en el mismo directorio que el archivo anterior, se encuentran las definiciones con los consiguientes formatos de cada instrucción. Los formatos de las nuevas instrucciones se muestran en la figura 3.5.

0000000	rs2	rs1	rm	rd	1010011	FADD.S
0000100	rs2	rs1	rm	rd	1010011	FSUB.S
0001000	rs2	rs1	rm	rd	1010011	FMUL.S
0001100	rs2	rs1	rm	rd	1010011	FDIV.S

Figura 3.5: Formato de las instrucciones *fadd.s*, *fsub.s*, *fmul.s* y *fdiv.s*[1]

En el mismo archivo anterior, se añade en las correspondientes secciones los valores necesarios para las nuevas instrucciones:

- `.definition`

```
fadd =      [0000000.................1010011]
fmul =      [0001000.................1010011]
fdiv =      [0001100.................1010011]
...
```

  Las instrucciones de aritmética en punto flotante con uno o dos operandos fuente utilizan el formato R (ver la figura 2.2) usando los bits 25 y 26 para el campo de formato de punto flotante (*fmt*). Para la extensión F los bits del campo *fmt* corresponden al 00[1].

- `.output`

```
rv32i = {
        fp_add
        fp_mul
        fp_div
        ...
        }
```

- `.decode`

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

Para generar las ecuaciones que se buscan, se ejecutan los archivos *coredecode*, *espresso* y *addassign* con las instrucciones indicadas en la práctica. Las ecuaciones devueltas sustituyen a las incluidas en el módulo *dec_dec_ctl*.

En el mismo archivo, se asignan valores a los bits de las nuevas instrucciones utilizando las señales *i0_dp* e *i1_dp*.

```
assign fp_p.fp_add  = (i0_legal_decode_d & i0_dp.fp_add) |
(dec_i1_decode_d & i1_dp.fp_add);
assign dec_fp_add1      = i0_dp.fp_add;
assign dec_fp_add2      = i1_dp.fp_add;

assign fp_p.fp_mul  = (i0_legal_decode_d & i0_dp.fp_mul) |
(dec_i1_decode_d & i1_dp.fp_mul);
assign dec_fp_mul1      = i0_dp.fp_mul;
assign dec_fp_mul2      = i1_dp.fp_mul;

assign fp_p.fp_div  = (i0_legal_decode_d & i0_dp.fp_div) |
(dec_i1_decode_d & i1_dp.fp_div);
assign dec_fp_div1      = i0_dp.fp_div;
assign dec_fp_div2      = i1_dp.fp_div;
```

En cuanto a la etapa de ejecución, en la práctica 18 se propone integrar la unidad de punto flotante usando el *pipe* del divisor. Para ello, se instancian los módulos correspondientes de la unidad, se añade la señal de control de las instrucciones de punto flotante y se multiplexan las salidas generadas por cada módulo de la unidad y la de la propia división.

3.3. Estudio e integración de la unidad FPnew

Tras realizar la práctica 18 y antes de integrar la unidad FPnew, se decide crear un *pipe* independiente para la unidad de punto flotante usando la unidad proporcionada en la práctica, y así facilitar los siguientes pasos del trabajo pudiendo reutilizar este *pipe*.

Una de las dificultades que surgieron, fue el hacer que el *pipe* fuese capaz de aceptar las tres instrucciones de punto flotante. Para solucionarlo, en el archivo *exu.sv* (ubicado en el directorio *src/SweRVolfSoc/SweRVEh1CoreComplex/exu*), se crean dos señales de 32 bits llamadas *fp_rs1_d* y *fp_rs2_d* que, usando una asignación con triple *OR* lógica, permite seleccionar los operandos según la instrucción.

```
assign fp_rs1_d[31:0] = fpadd_rs1_d | fpmul_rs1_d | fpdiv_rs1_d;

assign fp_rs2_d[31:0] = fpadd_rs2_d | fpmul_rs2_d | fpdiv_rs2_d;
```

En cuanto a la instanciación del módulo que representa el *pipe* de la unidad de punto flotante (se encuentra en *src/SweRVolfSoC/SweRVEh1CoreComplex/exu/ exu_fp_ctl.sv*), tiene tres entradas y tres salidas. Dos de las tres entradas corresponden a los operandos, *fp_rs1_d* y *fp_rs2_d*. La entrada restante es la señal que viene de la etapa de decodificación de tipo *fp_pkt_t* que indica qué instrucción de punto flotante se está ejecutando en ese momento. De las tres salidas, la primera, llamada *fp_stall*, indica al procesador que se ha de parar la ejecución de instrucciones mientras se ejecuta el cálculo. Esto ocurre debido a que no está segmentada la unidad, al igual que sucede con el divisor. De las otras dos salidas, la de un bit, llamada *finish*, indica que el cálculo ha terminado y el resultado está disponible. Mientras que la última (*out*) de 32 bits contiene el resultado de la operación.

```
exu_fp_ctl fp_e1 (.*,
      .fp_p           ( fp_p                    ),    // I
      .operand1       ( fp_rs1_d[31:0]           ),    // I
      .operand2       ( fp_rs2_d[31:0]           ),    // I
      .fp_stall       ( exu_fp_stall            ),    // O
      .finish         ( exu_fp_finish           ),    // O
      .out            ( exu_fp_result[31:0]     ));   // O
```

En el módulo *exu_fp_ctl* se encuentran las instanciaciones de los tres módulos de la unidad de punto flotante, uno para cada instrucción. Para seleccionar la salida adecuada de cada módulo, se utiliza un multiplexor. Mientras que para la señal *finish*, se usa una asignación con triple *OR* lógica con las señales de finalización de cada módulo. Con el fin de que la unidad funcione correctamente, se usan varios biestables con señales internas.

Esta implementación fue testada en simulación usando varias pruebas en código en ensamblador creadas manualmente. También se verificó el funcionamiento en la placa Nexys 4 DDR usando las pruebas anteriores.

Posteriormente, se procedió al estudio de la unidad FPnew. Para integrarla usando el *pipe* ya creado, es necesario estudiar los puertos (ver tabla 3.1) de los que dispone, así como los protocolos de comunicación que utiliza.

El protocolo usado es *handshake*, esta interfaz controla el flujo de datos de entrada y salida y se comporta de la siguiente forma[7]:

- La señal *valid* indica que los datos en la interfaz son válidos y estables.
- Una vez activada la señal *valid*, no debe desactivarse hasta que se complete el handshake.
- La señal *ready* indica que la interfaz está lista para procesar los datos en el siguiente flanco ascendente de reloj.
- Una vez que las señales *valid* y *ready* están activas en el mismo flanco ascendente de reloj, la transacción se considera completa.

- Después de una transacción completada, la señal *valid* puede permanecer activa para indicar que hay nuevos datos disponibles para transferir.

Hay que destacar que el protocolo de *handshake* se desarrolla en una dirección de arriba hacia abajo, donde la activación de *ready* puede depender de *valid*, pero no al revés.

Nombre	**Tipo**	**Descripción**
clk_i	Entrada	Señal de reloj que funciona por flanco ascendente.
rst_ni	Entrada	*Reset* asíncrono, activo a baja.
operands_i	Entrada	Matriz que admite hasta tres operandos a la vez.
rnd_mode_i	Entrada	Modo de redondeo en punto flotante.
op_i	Entrada	Selector del tipo de operación.
op_mod_i	Entrada	Modificador de operación. Ejemplo: Si se quiere usar la resta, el tipo de operación es suma y se activa el modificador para indicar que es una resta.
src_fmt_i	Entrada	El formato de la entrada en punto flotante (32 bits, 64 bits, etc.).
dst_fmt_i	Entrada	El formato de la salida en punto flotante.
int_fmt_i	Entrada	El formato de los enteros.
vectorial_op_i	Entrada	Selector del tipo de operación vectorial (no usado).
tag_i	Entrada	Entrada de etiqueta de operación (no usado).
simd_mask_i	Entrada	Entrada de máscara vectorial para los *flags* de estado (no usado).
in_valid_i	Entrada	Indica que la entrada proporcionada es válida (protocolo de comunicación *handshake*).
in_ready_o	Salida	Indica que la interfaz de entrada (la unidad de PF) está lista para recibir nuevos datos (protocolo de comunicación *handshake*).
flush_i	Entrada	Reset síncrono del *pipeline*.
result_o	Salida	Resultado de la operación.
status_o	Salida	*Flags* de estado de punto flotante de RISC-V (no usado).
tag_o	Salida	Salida de etiqueta de operación(no usado).
eout_valid_o	Salida	Indica que los datos de salida son válidos (protocolo de comunicación *handshake*).
out_ready_i	Entrada	Indica que la interfaz de salida (el procesador) está lista para recibir nuevos datos (protocolo de comunicación *handshake*).
busy_o	Salida	Indica si la unidad se encuentra en uso.

Tabla 3.1: Puertos de la unidad FPnew[7]

A la hora de integrar la unidad, no es necesario hacer grandes cambios en la etapa de decodificación ya que las nuevas instrucciones ya se han introducido al realizar el laboratorio 18. Tan solo hace falta crear una señal del tipo de datos usado en la unidad, para indicar el tipo de operación (*fpnew_pkg::operation_e*). También se crea una señal de un bit para indicar el modificador de operación.

```
output fpnew_pkg::operation_e op_i,
output logic          op_mod_i,
```

Para traducir el tipo de instrucción en punto flotante al tipo *fpnew_pkg::operation_e*, se implementa en el archivo *dec_decode_ctl.sv* un bloque *always_ff* que actúa como un *flip-flop* que se activa cada flanco de subida del reloj. Usando una secuencia de *if-else* se consigue asignar el valor correcto a la señal. A continuación se muestra la implementación (la instrucción *fsub* fue añadida posteriormente).

```
always_ff @ ( posedge clk or negedge rst_l) begin

        if (rst_l == 0) begin
            op_i  <= fpnew_pkg::SGNJ;
            op_mod_i <= 1'b0;
            end
        else begin
             if(fp_p.fp_add) begin
            op_i <= fpnew_pkg::ADD;
            op_mod_i <= 1'b0;
            end
        else if(fp_p.fp_mul) begin
            op_i <= fpnew_pkg::MUL;
            op_mod_i <= 1'b0;
            end
        else if(fp_p.fp_div) begin
            op_i <= fpnew_pkg::DIV;
            op_mod_i <= 1'b0;
            end
        else if(fp_p.fp_sub) begin
            op_i <= fpnew_pkg::ADD;
            op_mod_i <= 1'b1;
            end
        end
  end
```

En el módulo que representa el *pipe* (*exu_fp_ctl.sv*), se realizan bastantes cambios para lograr una correcta comunicación con la unidad. Para introducir los operandos, se crea una matriz que es la señal que se envía. Los operandos son asignados con un bloque *always_ff* muy similar al mostrado en el párrafo anterior.

En la primera etapa de codificación, se usaron varios biestables para así retrasar los ciclos necesarios, ciertas señales para el protocolo *handshake*. Tras realizar las primeras pruebas en simulación, no se consiguió el resultado acertado. Por lo que

se depuraron exhaustivamente las señales, y se corrigieron los diferentes errores encontrados.

Finalmente, la implementación queda de la siguiente forma:

- Se instancia el módulo de la unidad con las señales correspondientes.

```
fpnew_top #(
   .Features       ( fpnew_pkg::RV32F             ),
   .Implementation ( fpnew_pkg::DEFAULT_NOREGS ),
   .TagType        ( logic                        )
) i_fpnew_top (
   .clk_i(clk),
   .rst_ni(rst_l),
   .operands_i(operands_i),
   .rnd_mode_i(fpnew_pkg::RNE),
   .op_i(fpu_op_i),
   .op_mod_i(fpu_op_mod_i),
   .src_fmt_i(fpnew_pkg::FP32),
   .dst_fmt_i(fpnew_pkg::FP32),
   .int_fmt_i(fpnew_pkg::INT32),
   .vectorial_op_i(),
   .tag_i(),
   .in_valid_i(in_valid),
   .in_ready_o(),
   .flush_i(inverted_rst_l),
   .result_o(out_fp),
   .status_o(),
   .tag_o(),
   .out_valid_o(finish_operation),
   .out_ready_i(1'b1),
   .busy_o(busy)
);
```

- Se usa un *flip-flop* de reloj para indicar que los operandos de entrada son válidos en el siguiente ciclo, que es cuándo los operandos han sido ya asignados a la matriz.

```
rvdff #(1) in_valid_fp (.*, .clk(active_clk),
.din(fp_p.fp_add | fp_p.fp_mul | fp_p.fp_div | fp_p.fp_sub),
.dout(in_valid));
```

- Se usa otro *flip-flop* de reloj para retrasar un ciclo la señal de finalización de la unidad. Así se consigue evitar que la señal de finalización se active en el mismo ciclo en el que se introducen las señales en la unidad.

```
rvdff #(1) finish_delayed (.*, .clk(clk),
.din(finish_operation),        .dout(finish));
```

- Por último, se utiliza un *flip-flop* de reloj con un *enable* para retrasar un ciclo el resultado. Dado que el SweRV EH1 recoge el resultado en el siguiente ciclo en el que la señal *finish* se activa, y que, como se ha explicado en el punto anterior, se retrasa un ciclo la señal *finish*, se ha de almacenar en un biestable el resultado para el ciclo siguiente, que es en el que *finish* se activa. Esto es necesario porque la salida de la unidad con el resultado, se pone a cero en el ciclo siguiente.

```
rvdffe #(32) out_delayed (.*, .en(finish_operation),
.clk(clk),      .din(out_fp),          .dout(out));
```

Capítulo 4

Verificación

Como se menciona en el capítulo anterior, a medida que se iba codificando, se depuraban los errores que iban surgiendo. Una vez que las primeras pruebas en simulación fueron satisfactorias, se pasó a la etapa de verificación.

4.1. Test manuales

Las primeras pruebas en esta etapa son utilizando pequeños códigos en ensamblador, probando las tres instrucciones con diferentes valores como operandos y en diferente orden. En la figura 4.1 se enseña el entorno de PlatformIO en Visual Studio Code y un ejemplo de las pruebas usadas. Las instrucciones de punto flotante se introducen directamente en hexadecimal para que el compilador no dé errores, ya que no se compila usando la extensión F.

Para comprobar que todo se está ejecutando de forma correcta, se hacen las comprobaciones sobre el archivo de simulación generado con Verilator (se crea en la carpeta */verilatorSIM* usando el *makefile*), utilizando la aplicación GTKWave. En la figura 4.2, se muestran las formas de onda ejecutando el código mostrado en la figura 4.1.
Al principio, la señal *ifu_i0_instr* contiene la instrucción 0x01CE8F53, que corresponde a la instrucción *fadd*. En el ciclo siguiente, la señal *dec_i0_instr_d* contiene la instrucción, lo que quiere decir, que en ese momento se encuentra en la etapa de decodificación. En esa etapa, se le asigna el valor de la instrucción en punto flotante a la señal *fp_p*. En este mismo ciclo, se introducen los operandos en la matriz *operands_i* por medio del *flip-flop* comentado en el capítulo anterior. En el ciclo posterior, la señal *in_valid* indica que la instrucción se encuentra en el *pipe* de punto flotante y que puede empezar a calcular. Al ser la instrucción de suma, la salida (*out_fp*) se genera en el mismo ciclo. Usando otro *flip-flop*, se asigna la salida a la señal *out*, así se consigue que el ciclo siguiente al que *finish* se activa, también esté disponible el resultado. Acto seguido, el *enable* del registro *t5* se establece a uno y

el registro recoge el valor.

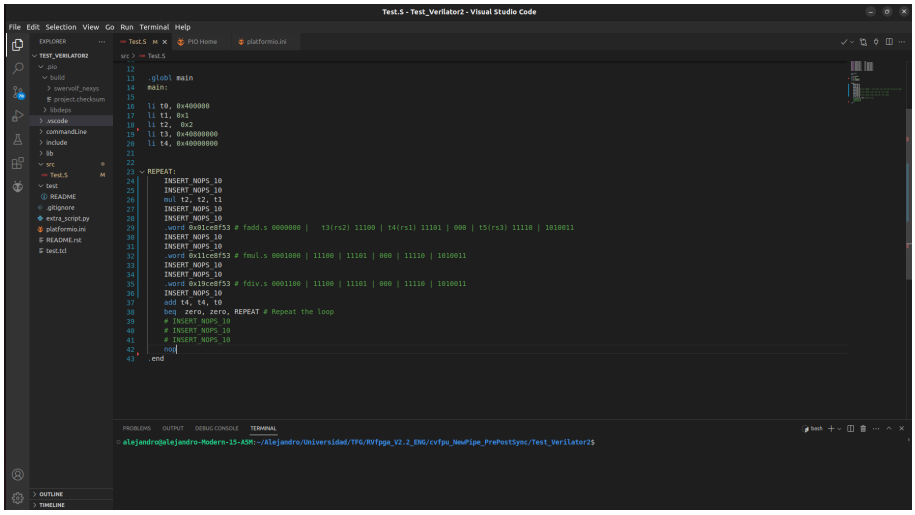

Figura 4.1: IDE platformIO utilizado para generar los códigos de prueba. (Elaboración propia)

Tras corregir errores menores, se intenta generar el *bitstream* para probarlo en la FPGA. No se consigue debido a que Vivado advierte que hay caminos críticos (ver la figura 4.3) que no cumplen con los requisitos de tiempo, lo que indica que hay peligro de que el diseño no funcione como debería. La solución pasaría por analizar exhaustivamente la unidad y su integración en el *pipeline* para tratar de eliminar esos caminos críticos, pero eso requeriría mucho más tiempo del que se dispone en este trabajo, así que se opta por una solución más sencilla y rápida como disminuir la frecuencia del reloj modificando el archivo *clk_gen_nexys.v* (ubicado en el directorio *src/OtherSources/*). La frecuencia original es de 50 MHz, tras el cambio pasa a ser de 25 MHz.

Una vez generado el *bitstream*, se vuelca en la placa y se utiliza la herramienta de *run & debug* de Visual Studio Code para comprobar que los valores de los registros son los correctos tras ejecutar las nuevas instrucciones. En la figura 4.4 se muestra un ejemplo de las pruebas que se han realizado. Todas las pruebas manuales siguen el mismo esquema, se inicializan los registros a utilizar como operandos con diferentes valores, en el caso de los registros que se van a usar en las operaciones en punto flotante, se inicializan con valores en dicho formato. En cuanto a las instrucciones, se intercalan instrucciones regulares con instrucciones de punto flotante para testear el comportamiento en diferentes situaciones. También se hace uso de instrucciones NOP, las cuales no realizan ninguna operación útil y se usan para crear retrasos específicos en el flujo de ejecución y poder estudiar las formas de onda de la simulación de una forma más sencilla.
En particular, en la misma figura se resalta con un rectángulo rojo el contenido del registro destino, *t5* en este caso. Tras haberse ejecutado la instrucción anterior, que corresponde a una suma en punto flotante (*fadd*) de los registros *t3* y *t4*, el resultado que contiene *t5* es el correcto, teniendo en cuenta el estándar IEEE 754.

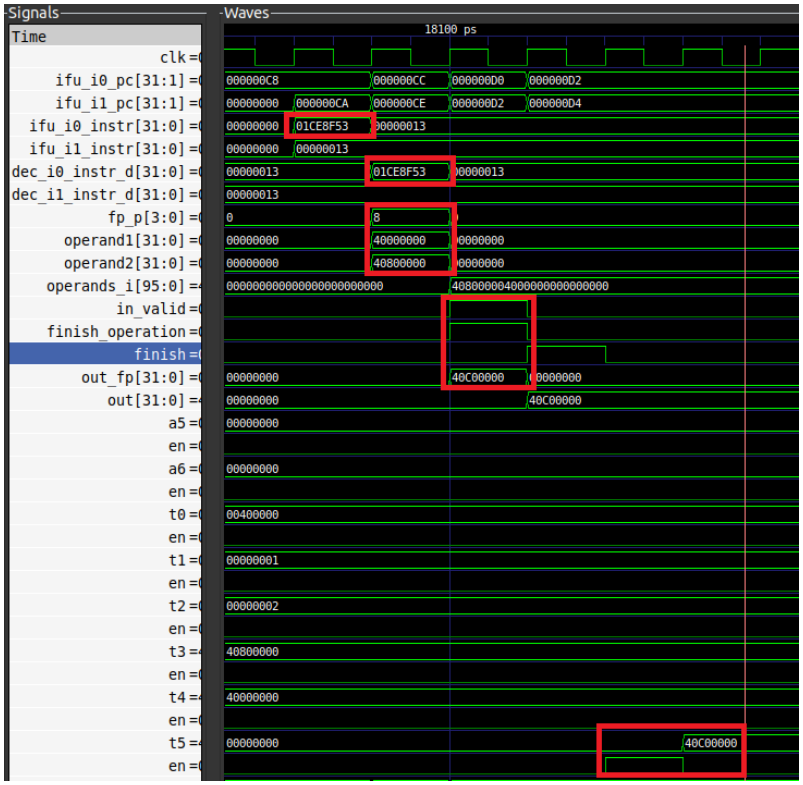

Figura 4.2: Formas de onda de la instrucción *fadd* en GTKWave. (Elaboración propia)

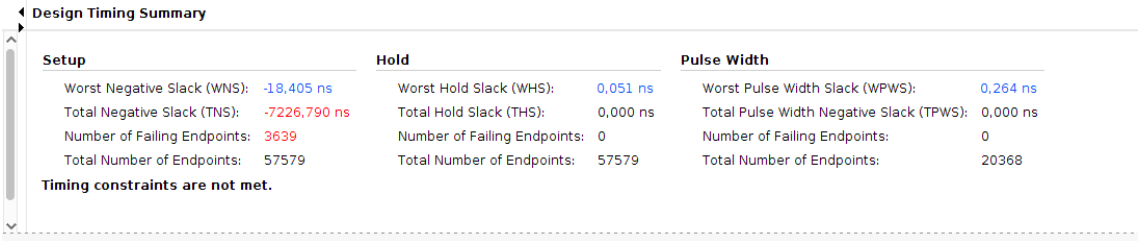

Figura 4.3: Temporización de diseño. (Elaboración propia)

4.2. Test oficiales de RISC-V

Tras superar las primeras pruebas, se decide ejecutar los test oficiales de la organización RISC-V llamados *RISC-V Architecture Test SIG* [5] para así, tener la certeza de que tanto el procesador como la unidad integrada tienen un funcionamiento adecuado para una amplia variedad de situaciones.

La principal dificultad de esta tarea es conseguir adaptar las nuevas versiones de las pruebas al SweRVolf. En el repositorio oficial del SoC [3], se informa de que dispone de soporte para la versión 1.0 de estos test. Sin embargo, esta versión no soporta la extensión F de RISC-V, por lo que es necesario adaptar una de las nuevas versiones con soporte para instrucciones de punto flotante. No es hasta la versión 2.5 que se añaden test para esta extensión.

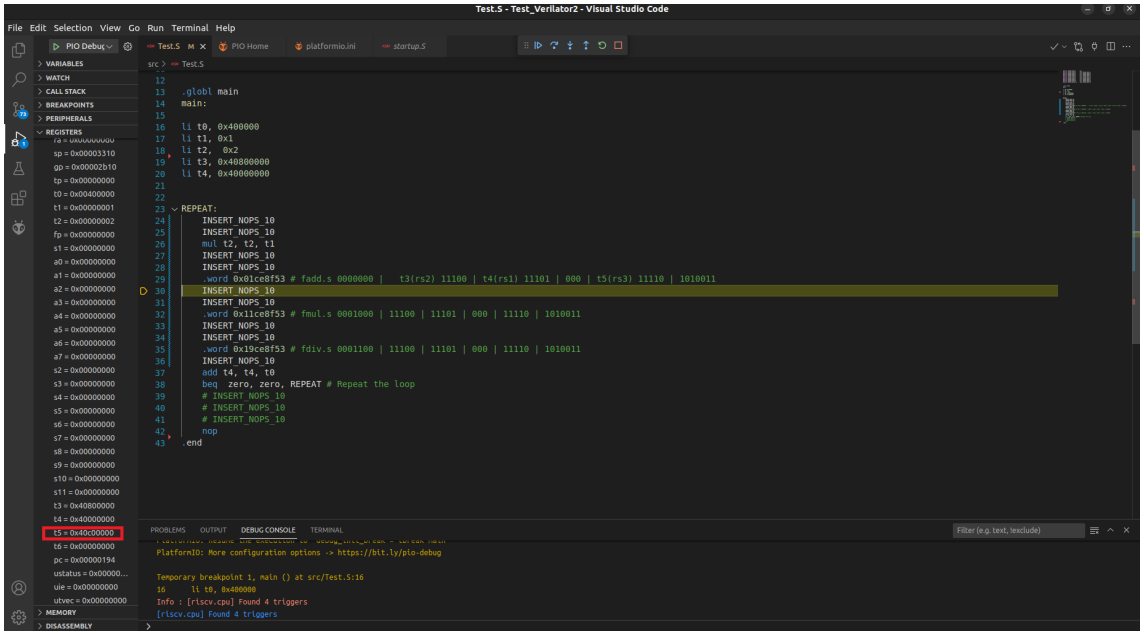

Figura 4.4: Ejecución y depuración en VS Code sobre la placa Nexys 4 DDR (Elaboración propia)

La primera idea que surge es adaptar los test de punto flotante de la versión 2.5 a la 1.0. Pero existe el problema de que la estructura interna de los test es totalmente diferente de una versión a otra, ya que los archivos de encabezado, las funciones de inicialización (ver la figura 4.5) y la estructura de directorios en la que se dividen los test, han sido modificados. Por esta razón, se acaba desechando la idea para adaptar otra versión más actualizada.

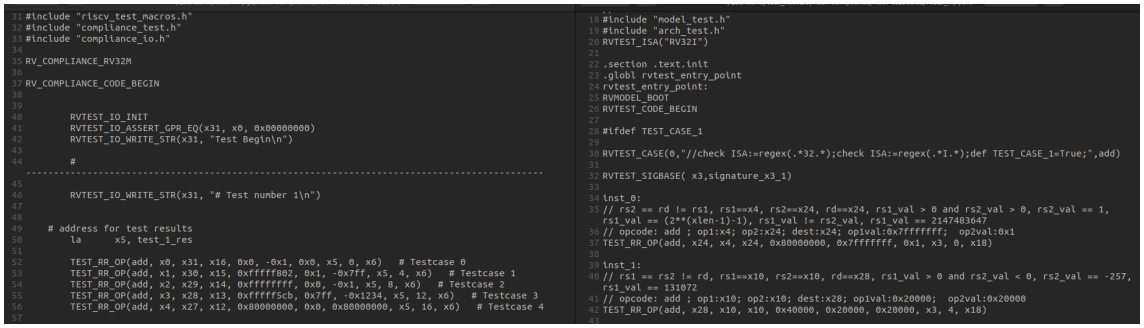

Figura 4.5: Código de los test para la instrucción *add*. Versión 1.0 (izq.) y 2.5 (dcha.). (Elaboración propia)

Dado que la mayoría de modelos disponibles usan la última versión de los test, se decide intentar crear un modelo para el SoC. Estas nuevas versiones usan RISCOF (*RISC-V Compatibility Framework*)[10], un marco basado en Python que permite probar un diseño desarrollado en RISC-V comparándolo con un modelo de referencia RISC-V utilizando un conjunto de pruebas en ensamblador.

Después de instalar las herramientas que se indican, se crea un modelo siguiendo el manual de RISCOF[10]. En este modelo se generan varios archivos necesarios

para la ejecución de los test, de éstos hay que modificar los siguientes archivos para adaptarlos al dispositivo a probar:

- *config.ini*: Hay que incluir la ruta de la carpeta que contiene los archivos creados.
- *model_test.h*: Es necesario actualizar las macros para que se adapten al dispositivo.
- *link.ld*: El *script* de enlazado que se usa durante la compilación de las pruebas.
- *riscof_target-name.py*: Hay que completar las tres funciones que incluye el archivo. La primera es la función de inicialización (*initialize*), que sirve para recibir y asignar las rutas de compilación necesarios para ejecutar las pruebas. La segunda es la función de llamada *build* que tiene la función de ejecutar los comandos para compilar el dispositivo. Y por último, la tercera función (*runTests*) encargada de compilar los test y correrlos en la unidad a probar.

Basándose en modelos ya utilizados para otros dispositivos, se implementan las funciones necesarias de los archivos expuestos.

Después de varios intentos de hacerlo funcionar sin éxito, se descarta, de nuevo, el utilizar las últimas versiones de los test debido a que, pese a tener varios modelos ya creados para otras unidades, no se asemejan a los comandos necesarios para hacerlo funcionar en el SweRVolf.

La última opción es adaptar la versión 2.5, que es la primera versión que contiene soporte para extensión de punto flotante. Como se ha descrito anteriormente, los archivos de encabezado cambian, así como los de configuración del dispositivo.

La 2.5 funciona usando una serie de *makefiles*, por ello, tras clonar el repositorio, se ha de completar el archivo *Makefile.include* (ver la figura 4.6). En él, hay varios parámetros para indicar la ruta a la carpeta que incluye el modelo del dispositivo, el nombre del dispositivo, el ancho de palabra (32 o 64 bits), la extensión que se quiere probar, entre otros.

La ruta donde se debe encontrar el modelo, ha de tener una carpeta llamada *riscv-target*. Esta carpeta tiene que albergar en su interior el archivo de enlazado (*link.ld*), el *model_test.h* que contiene las macros, como se ha explicado antes, además de cualquier archivo extra que sea necesario para el dispositivo.

Para configurar correctamente el entorno, se requiere crear una carpeta llamada *device* dentro de *riscv-target*. Además, se debe crear una carpeta correspondiente a los bits del dispositivo, en este caso *device/rv32i_m*. Dentro de esta carpeta, se deben crear subdirectorios con el nombre de las extensiones admitidas. Cada uno de estos subdirectorios debe contener un archivo llamado *Makefile.include*, el cual debe incluir las variables necesarias:

```
# set TARGETDIR to point to the directory which contains a sub-folder in the same name as the target
export TARGETDIR ?= /home/alejandro/Escritorio/test_riscv2.5/fusesoc_libraries/swervolf/riscv-target

# set XLEN to max supported XLEN. Allowed values are 32 and 64
export XLEN               ?= 32

# name of the target. Note a folder of the same name must exist in the TARGETDIR directory
export RISCV_TARGET       ?= swerv

#set the RISCV_DEVICE environment to the extension you want to compile, simulate and verify. Leave
#this blank if you want to iterate through all the supported extensions of your target. Allowed values
#are the individual names of the extensions supported by your target like: I, M, C or Zifencei, etc.
#Only one extension can be selected at once.
export RISCV_DEVICE       ?=

# set RISCV_TEST to the name of a test to run only that single test in the selected device. It is an error
# to specify a test that doesn't exist in the selected device.
# Leave blank to run all selected tests.
export RISCV_TEST         ?=

# set this to a string which needs to be passed to your target Makefile.include files
export RISCV_TARGET_FLAGS ?=

# set this if you want to enable assertions on the test-suites. Currently no tests support
# assertions.
export RISCV_ASSERT       ?= 0

# set the number of parallel jobs (along with any other arguments) you would like to execute. Note that the target needs to ensure
# that no common files across jobs are created/overwritten leading to unknown behavior
JOBS                      ?= -j1

```

Figura 4.6: *Makefile.include* con los parámetros de configuración (Elaboración propia)

- *RUN_ TARGET*: Incluye los comandos para ejecutar los .ELF en el dispositivo objetivo.
- *COMPILE_ TARGET*: Contiene los comandos para compilar las pruebas en ensamblador.

La dificultad que surge a la hora de adaptar los test, está relacionada con el archivo *model_ test.h*, ya que en la versión 1.0 no existe. Dado que nadie hasta la fecha había adaptado este SoC a estos test, no existe documentación alguna sobre cómo se ha de adaptar.
Tras una laboriosa tarea de investigación y estudio, que ha llevado un tiempo considerable, sobre la adaptación de la versión 1.0 y la de otros dispositivos desarrollados por la comunidad, se comprendió la forma de adaptar correctamente los test.

La clave para adaptar estos test reside en dos archivos que sirven para la configuración de las macros necesarias, llamados *compliance_ io.h* y *compliance_ test.h*. Y es que, tras el estudio ya nombrado, se descubre que pese al haber definido nuevas macros y usar un archivo nuevo para albergarlas (*model_ test.h*), la versión 2.5 acepta las macros de las primeras versiones. Por lo que se crea el archivo *model_ test.h*, usando las macros de los dos archivos disponibles, en este nuevo.

Haciendo lo anterior, se consigue que el SoC SweRVolf tenga soporte desde la versión 2.5 hasta la versión previa a la actualización a RISCOF, algo que, como se ha descrito antes, no se había logrado hasta ahora en este SoC.
Además, se han realizado los test para todas las extensiones compatibles con el SweRVolf, y se ha confirmado que todos los resultados obtenidos son correctos.

Respecto a las pruebas de los test con la extensión F, los resultados no son los esperados al no conseguir generar ninguna salida. Por lo que se investiga sobre cuál

```
TARGET_SIM ?= $(TARGETDIR)/$(RISCV_TARGET)/Vrvfpgasim
ifeq ($(shell command -v $(TARGET_SIM) 2> /dev/null),)
    $(error Target simulator executable '$(TARGET_SIM)` not found)
endif

RUN_TARGET=\
    $(TARGET_SIM) \
        +signature=$(*).signature.output \
        +ram_init_file=$(<).hex \
        +timeout=1000000000 \
        2> $@

RISCV_PREFIX   ?= riscv32-unknown-elf-
RISCV_GCC      ?= $(RISCV_PREFIX)gcc
RISCV_OBJCOPY  ?= $(RISCV_PREFIX)objcopy
RISCV_OBJDUMP  ?= $(RISCV_PREFIX)objdump
RISCV_GCC_OPTS ?= -static -mcmodel=medany -fvisibility=hidden -nostdlib -nostartfiles -march=rv32imc_zfinx -mabi=ilp32

COMPILE_TARGET=\
        $$(RISCV_GCC) $(1) $$(RISCV_GCC_OPTS) \
                -I$(ROOTDIR)/riscv-test-env/ \
                -I$(TARGETDIR)/$(RISCV_TARGET)/ \
                -T$(TARGETDIR)/$(RISCV_TARGET)/link.ld $$< \
                -o $$@; \
        $$(RISCV_OBJCOPY) -O binary $$@  $$@.bin; \
        $$(RISCV_OBJDUMP) -D $$@ > $$@.objdump; \
        python3 $(TARGETDIR)/$(RISCV_TARGET)/makehex.py $$@.bin > $$@.hex;
```

Figura 4.7: *Makefile.include* con los comandos para la compilación y ejecución (Elaboración propia)

podía ser la razón por la que los test en punto flotante no estaban funcionando.

De nuevo, se vuelve a comprobar que la integración funciona correctamente mediante simulación y placa, pero no se halla nada anormal. Por lo que se intentan realizar pruebas manuales compilando con la extensión F. Tras conseguir compilar y cargar las pruebas, se obtiene que las instrucciones no están siendo ejecutadas por el procesador, así que se procede a estudiar qué puede estar causando este comportamiento, que ocurre solo al compilar usando dicha extensión.

Después de revisar exhaustivamente documentación sobre RISC-V y explorar foros relacionados, se descubre que las instrucciones correspondientes a la extensión F hacen uso de registros específicos para operaciones de punto flotante. A continuación, se muestra un ejemplo de una instrucción de suma: (*fadd.s*):

```
fadd.s  ft2, ft0, ft1
```

Esto no se había descubierto antes porque las pruebas se realizaban introduciendo las instrucciones como *.word* en hexadecimal, lo que implicaba que no se compilaban utilizando la extensión F. Como resultado, los registros utilizados en las instrucciones eran los registros de enteros, en lugar de los registros de punto flotante. Después de esto, se realizaron pruebas manuales para verificar si las instrucciones de la extensión F permitían usar los registros de enteros en lugar de los de punto flotante, dado que el procesador no cuenta con un banco de registros de punto flotante. Estas pruebas no consiguen el resultado esperado, así que se recurre al manual de RISC-V [1] en busca de información adicional. Pero en el manual se indica que la extensión necesita un banco de registros de punto flotante para funcionar, no aceptando los registros de enteros.

A partir de ahí, se buscan soluciones al problema. Tras indagar en muchos foros de RISC-V y leer bastante documentación, se encuentra que, en 2021, se creó una

extensión llamada *Zfinx* [2]. Esta extensión tiene como objetivo usar instrucciones de la extensión F, pero usando los registros de enteros en vez de los de punto flotante.

Tras este descubrimiento, se plantea el objetivo de adaptar manualmente una pequeña parte de los test oficiales para utilizar la extensión *Zfinx*. Así que se adaptan las primeras 40 instrucciones del primer archivo en ensamblador de la suma, resta, multiplicación y división. En la figura 4.8 se muestra la adaptación del test de *fadd*.

```
//
// This assembly file tests the fadd.s instruction of the RISC-V F extension for the fadd_b1 covergroup.
//
#include "model_test.h"
#include "arch_test.h"
RVTEST_ISA("RV32I_Zfinx")

.section .text.init
.globl rvtest_entry_point
rvtest_entry_point:
RVMODEL_BOOT
RVTEST_CODE_BEGIN

#ifdef TEST_CASE_1

RVTEST_CASE(0,"//check ISA:=regex(.*32.*);check ISA:=regex(.*I.*Zfinx.*);def TEST_CASE_1=True;",fadd_b1)

RVTEST_FP_ENABLE()
RVTEST_VALBASEUPD(x16,test_fp)
RVTEST_SIGBASE(x15,signature_x15_1)

inst_0:

lw      a0,0(a6)
lw      a0,4(a6)
fadd.s  a0,a0,a0
sw      a0,0(a5)

inst_1:
lw      a1,8(a6)
lw      a2,12(a6)
fadd.s  a2,a1,a2
sw      a2,8(a5)

inst_2:
lw      a1,16(a6)
lw      a2,20(a6)
fadd.s  a2,a1,a2
sw      a2,16(a5)

inst_3:
lw      a1,24(a6)
lw      a2,28(a6)
fadd.s  a2,a1,a2
sw      a2,24(a5)

inst_4:
lw      t2,32(a6)
lw      t2,36(a6)
fadd.s  s10,t2,t2
sw      s10,32(a5)
```

Figura 4.8: Test para la instrucción *fadd* adaptado a la extensión *Zfinx* (Elaboración propia)

Después de esto, se compilan las adaptaciones con la extensión *Zfinx*.
Tras ejecutar los test y generarse la salida correspondiente, se comparan los resultados obtenidos con los de referencia de los test. Éstos se pueden ver en la figura 4.9, concretamente en las líneas con número impar, las líneas pares en cambio, corresponden con la salida generada por instrucciones que indican códigos de excepción. La salida es *deadbeef*, ya que se han omitido de las pruebas al no ser compatibles.

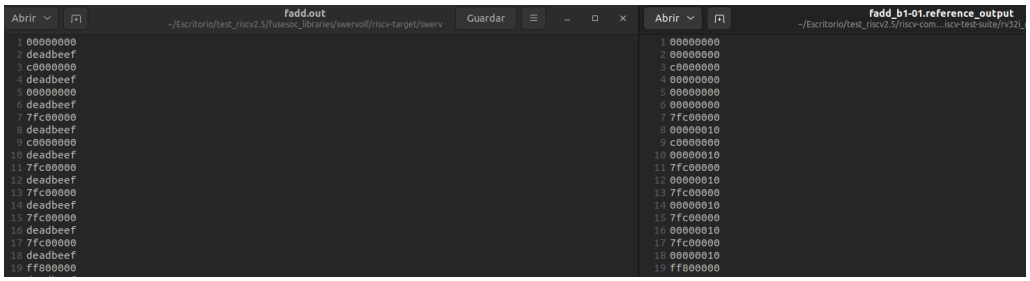

Figura 4.9: Comparativa de los resultados para la instrucción *fadd*. (Elaboración propia)

Durante la realización de los test se detecta un error en algunas salidas. Pese a que los resultados son correctos, hay ocasiones en las que no se imprime el primer dígito hexadecimal, Esto ocurre en cualquier extensión a probar, no solo con *Zfinx*. Para asegurarse de que no es un error fruto de la integración de la unidad, se ejecutan los test en las tres versiones disponibles: SweRVolf original, SweRVolfX y el SweRVolfX modificado con la unidad.

En la ejecución de los test, se obtiene que en el SweRVolf original no se produce el fallo, pero en el SweRVolfX proporcionado por el curso, sí ocurre. Por lo que se descarta que sea un fallo producido por este trabajo. En la figura 4.10 se muestra el error usando el SweRVolfX (mismo resultado que con la versión modificada con la unidad). Este error ya ha sido reportado.

Pese a este error externo al trabajo, se consigue, tras mucho esfuerzo y tiempo, adaptar satisfactoriamente los test oficiales de RISC-V, obteniendo los resultados esperados con las extensiones ya soportadas por el SoC y con la extensión añadida para este trabajo, *Zfinx*.

```
Check                    C-ADD ... OK
Check                   C-ANDI ... OK
Check                    C-AND ... OK
Check                   C-BEQZ ... OK
Check                   C-BNEZ ... OK
Check                    C-JAL ... OK
Check                   C-JALR3c3
< ffffffff
---
> fffffff
 ... FAIL
Check                      C-J ... OK
Check                    C-JR3c3
< ffffffff
---
> fffffff
 ... FAIL
Check                      C-LI ... OK
Check                     C-LUI ... OK
Check                      C-LW ... OK
Check                    C-LWSP ... OK
Check                      C-MV ... OK
Check                      C-OR ... OK
Check                   C-SLLI ... OK
```

Figura 4.10: Error en la salida en el SweRVolfX. (Elaboración propia)

Capítulo 5

Evaluación de la implementación

Para realizar la evaluación, se ejecuta el método de bisección en las implementaciones disponibles.

Este método es proporcionado por el curso RVfpga[9] en la práctica 18 en forma de entorno para PlatformIO. Éste usa un archivo .C que implementa el algoritmo y un .S que contiene las instrucciones para las operaciones en punto flotante. La ejecución se realiza sobre la placa Nexys 4 DDR.

Para medir el rendimiento, se hace uso de contadores hardware disponibles en el procesador, los cuales pueden ser utilizados mediante las librerías PSP/BSP de Western Digital, siguiendo las instrucciones de la práctica 11. Los resultados devueltos son los siguientes: *Cycles SwEmul*, que representa los ciclos de ejecución requeridos para la emulación por software de las instrucciones de punto flotante mediante código en C, y *Cycles HwImpl*, que indica los ciclos necesarios para la ejecución utilizando la implementación hardware mediante código ensamblador. Este último, es el que tendremos en cuenta a la hora de analizar resultados en este capítulo. En la figura 5.1 se muestra el resultado del método en la implementación original de la práctica 18 (usando la unidad *fpu* en el *pipe* de la división).

A la hora de llevar a cabo el método en las dos implementaciones que se han hecho en este trabajo (*fpu* en un nuevo *pipe* y la de la unidad FPnew), surgió una dificultad que, al principio, hizo que no funcionase correctamente el algoritmo. Era debida a que el orden de los operandos de entrada en la versión de la práctica 18 es diferente al desarrollado en este TFG. Esto se ha solucionado modificando el orden de los registros de entrada en las instrucciones de punto flotante del método de bisección.

Una vez hecho lo anterior, se procede a ejecutar en ambas versiones obteniendo los siguientes resultados:

- Con la unidad *fpu* y el nuevo *pipe* correspondiente, se obtiene una cantidad de 2016 ciclos en la ejecución en placa como se muestra en la figura 5.2

Figura 5.1: Resultado del método de bisección en la implementación de la práctica 18. (Elaboración propia)

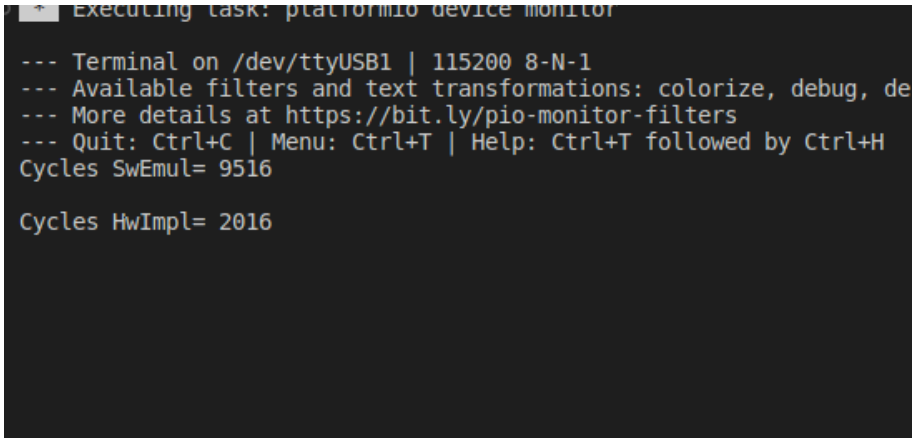

Figura 5.2: Resultado del método de bisección en la implementación con la *fpu* y su respectivo *pipe*. (Elaboración propia)

- Usando la unidad FPnew, en cambio, se consiguen 712 ciclos (ver la figura 5.3). Este resultado está condicionado por la reducción de frecuencia de reloj a la mitad que se ha explicado anteriormente, por tanto la ganancia en tiempo no es tan alta. Sin embargo, es muy probable que dedicando más tiempo a la integración de la nueva unidad se pudiera ajustar mucho mejor la frecuencia.

  Cabe destacar que, fue necesario reducir la tasa de baudios para que funcionase correctamente en el monitor de PlatformIO.

En la tabla 5.1 se muestran los resultados de las tres implementaciones probadas.

Implementación	**Ciclos hardware**
fpu usando el *pipe* de la división	1922
fpu usando su propio *pipe*	2016
FPnew	712

Tabla 5.1: Tabla con los resultados del método de bisección.

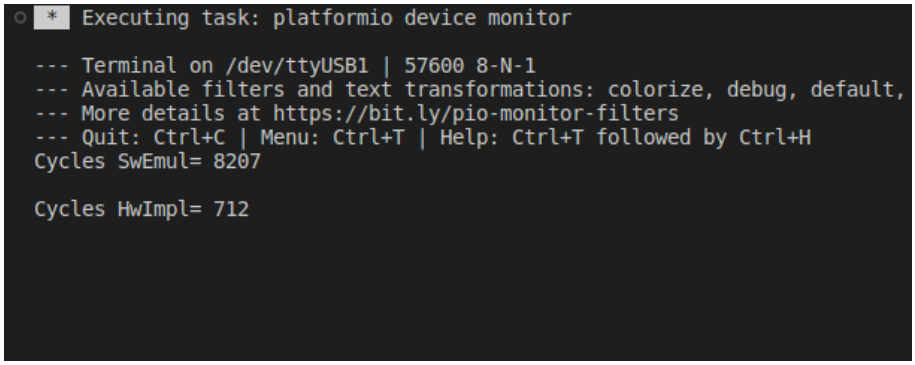

Figura 5.3: Resultado del método de bisección en la implementación con la unidad FPnew. (Elaboración propia)

Analizando los resultados anteriormente expuestos, se puede ver claramente lo eficiente que la unidad FPnew es, comparado con la otra unidad integrada al principio del trabajo. En el capítulo 5, la figura 4.8 ilustra, como se ha explicado en ese capítulo, que la unidad FPnew hace la suma en punto flotante en el mismo ciclo en el que la señal *in_valid* se establece a uno. Tanto la resta como la multiplicación comparten el mismo comportamiento. En la figura 5.4 se muestra las formas de onda de la instrucción *fmul*.

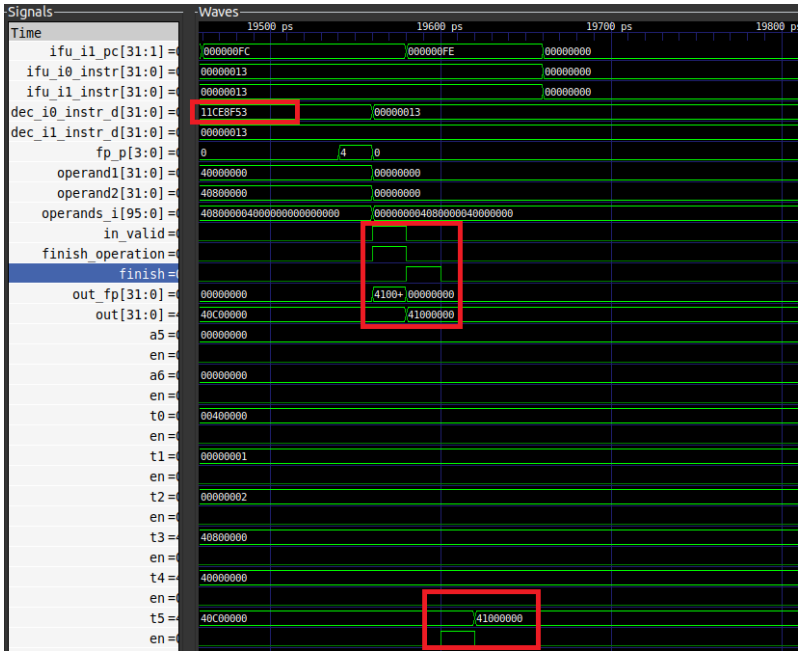

Figura 5.4: Formas de onda de la instrucción *fmul* en GTKWave. (Elaboración propia)

En cambio, la instrucción *fdiv* necesita de más ciclos, en concreto de diez. En la figura 5.5 se muestran las formas de onda, en ella se pueden apreciar los ciclos que tarda en generar la salida en *out_fp*.

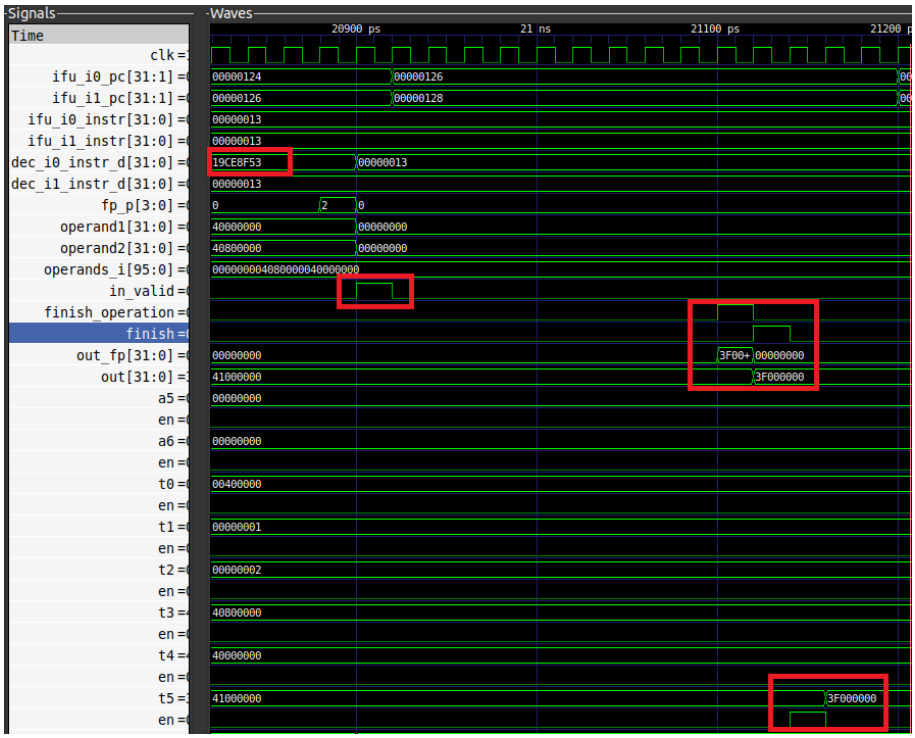

Figura 5.5: Formas de onda de la instrucción *fdiv* en GTKWave. (Elaboración propia)

Capítulo 6

Conclusiones y Trabajo Futuro

Los objetivos que fueron planteados al comienzo de este TFG han sido alcanzados. Como se ha ido explicando en esta memoria, se han seguido varias fases para la consecución de dichos objetivos.

La meta principal de soportar números en punto flotante en el procesador SweRV EH1, ha sido cumplida. Aunque la idea inicial consistía en dar soporte a las extensiones F y D, este objetivo fue establecido antes de iniciar el trabajo, cuando aún no se conocía la necesidad de incorporar un banco de registros de punto flotante. Gracias al estudio realizado para este trabajo, se ha conseguido encontrar una extensión compatible (*Zfinx*) con las necesidades del procesador y que cumple el objetivo de dar soporte para instrucciones en punto flotante. Además, se ha conseguido adaptar una nueva versión de los test oficiales de RISC-V en el SoC SweRVolf. Este logro puede tener un gran impacto en la comunidad al permitir probar las nuevas actualizaciones de los test, que incluyen tanto nuevas extensiones como una mayor cantidad de test en las extensiones ya soportadas anteriormente. Esta adaptación ha sido probada con todas las extensiones compatibles con SweRV EH1, obteniendo resultados exitosos en todas ellas. Esto reafirma la correcta implementación de la adaptación para dar soporte a los test oficiales y también confirma que se ha logrado cumplir el objetivo de no alterar el correcto funcionamiento del SoC.
Como al principio del trabajo se integró otra unidad menos eficiente, se realiza también una comparativa de rendimiento para reflejar los ciclos requeridos por cada una de las integraciones hechas, y así aportar al trabajo una evaluación sobre los rendimientos de ambas versiones.
Respecto al último objetivo restante, se ha logrado que ambas integraciones funcionen satisfactoriamente en la placa Nexys 4 DDR.

Gracias al uso de herramientas *open source* como la arquitectura RISC-V y el procesador SweRV EH1, se consigue que este trabajo pueda servir para contribuir al desarrollo de ambas comunidades, así como poder tener una gran utilidad en el ámbito académico a la hora de estudiar el funcionamiento de un procesador con una unidad de punto flotante. Un ejemplo de esto último, puede ser el curso RVfpga[9]

que ha servido como ayuda para desarrollar este TFG.

Respecto al trabajo futuro, en esta memoria se ha comentado el problema que puede conllevar bajar la frecuencia del procesador, por eso, puede ser interesante estudiar qué se podría hacer para cumplir con los requisitos de tiempo de los caminos críticos, conservando la frecuencia original.
También hay instrucciones que no han sido implementadas, por lo tanto, se podrían añadir esas instrucciones en un trabajo futuro. Así como la extensión *Zdinx*, equivalente a la extensión D.
Además, de cara a que este trabajo tenga un mayor impacto, los directores propusieron colgar el trabajo en el repositorio oficial de SweRV EH1 para que estuviese disponible para la comunidad de RISC-V y SweRV. Para lograrlo, habría que adaptar la implementación a las últimas versiones del procesador y redactar la documentación necesaria en inglés.

Pese a las posibles mejoras que se podrían hacer, el trabajo ha conseguido unos resultados que son más que suficientes para demostrar la viabilidad y el éxito de la integración. Además, este trabajo proporciona unas bases sólidas para la integración de nuevas unidades utilizando sus propios *pipes*, así como para su posterior prueba utilizando los test adaptados.

Chapter 7

Introduction

In this chapter, the background that has motivated the development of this project, the set objectives, and the work plan to achieve those objectives will be presented.

7.1. Background

In 1971, the Intel 4004, the first commercial microprocessor in history, was released. In the following years, Intel introduced a series of processors, the 8008 and 8080, which laid the foundation for the first generation of personal computers.

Until then, all processors used the Complex Instruction Set Computer (CISC) architecture or architectures based on it, such as the Intel 8086 and 8088 (x86 architecture), Motorola 68000, among others. However, as these processors improved in terms of performance and computing power, they also became more complex and less energy-efficient. This led to the emergence of the Reduced Instruction Set Computing (RISC) architecture, which focused on eliminating complex instructions to create simpler and more efficient processors. Some of the processors that emerged using RISC architecture include the MIPS R2000, IBM POWER1, DEC Alpha, among others.

The lack of an open-source architecture free from patents (all architectures up to that point were subject to expensive patents) led to the creation of RISC-V in 2010.

Another problem that arose with the first processors was the need to be able to represent very large or small real numbers in an easy and efficient way. The first solution devised was in 1914 by Torres y Quevedo, who described the first concept of floating-point representation, but it wasn't until 1938 that the first machine that included it, the Z1, was created. In the years that followed, all machines had real

number operations performed at the software level or with additional hardware. It was not until the 1980s that floating point units began to be included in processors for home use. This led in 1985 to the Institute of Electrical Engineers establishing a standard called IEEE 754 that was largely based on the Intel 8087 created by Kahan et al. in 1980. The purpose of this standard was to establish a set of rules on how to represent floating-point numbers, since each company had its own guidelines for defining these numbers, which led to incompatibilities and problems.

Regarding the processor used in this project, the SweRV EH1 is an open-source (RISC-V) processor developed by Western Digital in 2019. Although it is a powerful and energy-efficient processor, it lacks support for floating-point extensions, which limits its ability to perform precise and complex calculations.

Currently, there is a wide variety of open-source floating-point units developed by the community that can be implemented in RISC-V processors. In this work, we chose to implement the FPnew unit [7] by Stefan Mach on the SweRV EH1 processor.

7.2. Objectives

The main objective of this work is to add support for floating-point numbers using the D and F extensions to the SweRV EH1 processor. For this purpose, we use the SwervolfX SoC developed by the Imagination University Programme, which is an adaptation of the original Swervolf SoC from Chips Alliance. To gain a better understanding of the processor, we utilize the RVfpga course provided by the Imagination University Programme.

The objectives for implementing the floating-point unit in the processor are as follows:

- Do not alter the proper functioning of the SoC when integrating the unit.
- Expand the functionality of the processor. By supporting basic floating-point operations (addition, subtraction, multiplication, and division), the processor improves performance in tasks that require significant computational capability.
- The SoC should operate on the Nexys 4 DDR board.

7.3. Work Plan

After the initial meeting with the directors, the bases for the project's development are established. The project is divided into several phases:

- Preparation: Access to the RVfpga course provided by the Imagination University Programme is requested. Initial documentation is read, and recommended development environments are set up.
- Study of the SwervolfX SoC: The course consists of twenty practices focused on learning about the SoC and the SweRV EH1 processor. Some of the practices are completed to gain the necessary knowledge for making the required changes. One of the highlights is practice 18, which tells how to implement a basic floating point unit using the divisor pipe.
- Integration of the basic unit: The unit provided in the course is implemented by creating a new dedicated pipe for the floating-point unit.
- Study of the FPnew unit: The documentation is read to understand its operation and communication protocols.
- Integration of the FPnew unit: The previous integration is used as a base, and the new unit is adapted accordingly.

- Debugging the integration: Basic tests are conducted to detect and correct errors.
- Testing on the Nexys 4 DDR board: The functionality is verified to ensure it matches the results obtained in simulation.
- Verification: Official RISC-V Foundation tests are executed to ensure correct functionality in all possible scenarios.
- Evaluation of the implementation by means of the Bisection Method for finding the roots of a function.
- Report Writing: This report is written.

Chapter 8

Conclusions and Future Work

The objectives that were set at the beginning of this dissertation have been achieved. As explained in this report, several phases have been followed to achieve these objectives.

The main goal of supporting floating point numbers in the SweRV EH1 processor has been achieved. Although the initial idea was to support the F and D extensions, this objective was established before starting the project, when the need to incorporate a floating point register bank was not yet known. Thanks to the study carried out for this project, it has been possible to find an extension compatible (*Zfinx*) with the needs of the processor and that fulfills the objective of providing support for floating-point instructions. In addition, a new version of the official RISC-V tests has been adapted to the SweRVolf SoC. This achievement can have a great impact on the community by allowing to test the new test updates, which include both new extensions and a larger number of tests on previously supported extensions. This adaptation has been tested with all extensions supported by SweRV EH1, obtaining successful results in all of them. This reaffirms the correct implementation of the adaptation to support the official tests and also confirms that the objective of not altering the correct functioning of the SoC has been achieved.
As another less efficient unit was integrated at the beginning of the project, a performance comparison is also made to reflect the cycles required by each of the integrations made, and thus provide the project with an evaluation of the performance of both versions.
Regarding the last remaining objective, both integrations have been made to work satisfactorily on the Nexys 4 DDR board.

Thanks to the use of tools such as the RISC-V architecture and the SweRV EH1 processor, this work can contribute to the development of both communities, as well as being very useful in the academic field when studying the operation of a processor with a floating point unit. An example of the latter can be the RVfpga[9] course that has helped to develop this Final Degree Project.

Regarding future work, the problem of lowering the processor frequency has been discussed in this report, so it may be interesting to study what could be done to meet the timing requirements of the critical paths while preserving the original frequency.
There are also instructions that have not been implemented, therefore, those instructions could be added in a future work. As well as the Zdinx extension, equivalent to the D extension.
In addition, in order to make this project more impactful, the editors proposed to post the work in the official SweRV EH1 repository to make it available to the RISC-V and SweRV community. To achieve this, the implementation would have to be adapted to the latest versions of the processor and the necessary documentation would have to be written in English.

Despite the possible improvements that could be made, the work has achieved results that are more than sufficient to demonstrate the feasibility and success of the integration. In addition, this project provides a solid basis for the integration of new units using their own pipes, as well as for their subsequent testing using the adapted tests.

Bibliografía

[1] The risc-v instruction set manual. volume i. Disponible en https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[2] Risc-v zfinx repository. Disponible en https://github.com/riscv/riscv-zfinx.

[3] Alliance, C. Swervolf. Disponible en https://github.com/chipsalliance/Cores-SweRVolf.

[4] Dawson, J. P. Fpu. Disponible en https://github.com/dawsonjon/fpu.

[5] Gala, N. y Karasek, M. Risc-v architecture test sig. Disponible en https://github.com/riscv-non-isa/riscv-arch-test.

[6] Gustafson, J. L. Rbeating floating point at its own game: Posit arithmetic. Disponible en https://www.superfri.org/index.php/superfri/article/view/137.

[7] Mach, S., Schuiki, F., Zaruba, F. y Benini, L. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29(4), páginas 774–787, Disponible en https://github.com/openhwgroup/cvfpu.

[8] Mallasén, D., Murillo, R., Del Barrio, A. A., Botella, G., Piñuel, L. y Prieto-Matias, M. Percival: Open-source posit risc-v core with quire capability. *IEEE Transactions on Emerging Topics in Computing*, vol. 10(3), páginas 1241–1252, 2022. Disponible en https://github.com/artecs-group/PERCIVAL.

[9] Programme, I. U. Rvfpga: Understanding computer architecture. Disponible en https://university.imgtec.com/teaching-download/#rvfpga.

[10] Semiconductors, I. Riscof. Disponible en https://riscof.readthedocs.io/en/stable/.

[11] Society, I. C. Ieee standard for floating-point arithmetic. 2008.

[12] Waterman, A. Design of the risc-v instruction set architecture. Disponible en `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.pdf`.