P-IBEX: Acelerador Posit en plataforma basada en RISC-V
(P-IBEX: Posit Accelerator based on RISC-V platform)

Jaime del Rey García y Ángel Cruz Alonso

GRADO EN INGENIERÍA INFORMÁTICA.
FACULTAD DE INFORMÁTICA.
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado del Grado en Ingeniería Informática

2019/2020

Tutores:

Alberto Antonio del Barrio García
Guillermo Botella Juan

Resumen

Las arquitecturas hardware se abren, progresivamente, al mundo Open Source. En este trabajo se introducirán las bases fundamentales de una arquitectura computacional RISC, profundizando en la ISA abierta RISC-V. Se expondrán los antecedentes de la arquitectura, su estructura y se comentarán algunos de los principales cores, hoy en día disponibles, junto con las herramientas para el desarrollo de RISC-V.

Una vez introducidos los conceptos básicos, el núcleo del trabajo consistirá en el uso de la plataforma Ibex de LowRisc para el prototipado y la implementación de dos aceleradores hardware que calculen operaciones aritméticas con el formato numérico Posit introducido por John Gustafson. Se discutirán pruebas de concepto realizadas y a continuación la implementación de los módulos y los resultados obtenidos.

Como parte final, las conclusiones del trabajo realzan la gran capacidad de la arquitectura RISC-V, así como sus claras ventajas frente a sus competidores.

Palabras clave

RISC-V, computación, ISA, Posit, SIMD.

Abstract

Hardware architecture is slowly opening to Open Source world. In this BSc. thesis the fundamental bases of RISC and a detailed description into the open ISA RISC-V will be introduced. The architecture background, his structure, some available cores and tools for RISC-V development will be described.

Once the basic concepts are introduced, the core of the thesis consists in the use of LowRisc's Ibex project to prototipe and implement two hardware accelerators capable of computing John Gustarfson's Posit arithmetic operations. Proofs of concept, the implementation of the modules and results will be discussed.

Finally, the results will show the great capabilities and advantages that RISC-V has in contrast with his competitors.

Keywords

RISC-V, computation, ISA, Posit, SIMD.

Agradecimientos

A nuestros tutores y a todas las personas que nos han aguantado durante el proceso de realización de este proyecto.

Jaime y Ángel.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

Cada día más, la utilización de tecnologías libres lidera el mercado y proveen a desarrolladores software de todo tipo de herramientas de gran calidad. Sin embargo, el terreno del hardware, en su mayoría desarrollado con tecnologías propietarias ha supuesto que el acceso de la comunidad a aportaciones sea reducido.

En un mundo en auge con nuevos avances en inteligencia artificial, cloud computing y computación de alto rendimiento, RISC-V propone un nuevo paradigma, sencillo, modular y eficiente para el desarrollo del hardware mundial, que quiere corregir los errores aprendidos durante años en el diseño de ISAs.

Hasta 2010 la presencia escasa y de poca importancia de las ISAs libres, originó el nacimiento de RISC-V, que surgió para dar respuesta a la poca oferta libre en entornos de computación hardware.

Desde entonces RISC-V International, comunidad de desarrolladores hardware que trabajan en la ISA, ha ido creciendo exponencialmente, contando con miembros de alto nivel como NVIDIA, Google, NXP y la startup SiFive, centrada exclusivamente en diseños RISC-V.

La forma modular de la ISA permite desarrollar cores de muy pequeño tamaño y escaso consumo de energía. En concreto, algunas de las nuevas extensiones, como la extensión vectorial V, son observadas muy de cerca por las grandes compañías, con la intención de desarrollar posibles implementaciones centradas en el diseño de cores potentes para aplicaciones de inteligencia artificial.

1.2. Motivation

Nowadays, the use of Open Source technologies provide developers with multiple good quality tools. However, the hardware landscape is quite different. Hardware technologies are developed mostly using proprietary technologies. Due to this fact the community access to this technologies are not common.

In a world dominated by innovation in artificial intelligence, cloud computing and high performance computing, RISC-V propose a fresh, easy, modular and efficient paradigm to global hardware development, which looks for learn from past errors.

Until 2010 free ISAs are quite uncommon. This situation originated the beginning of RISC-V which borns as a response to low hardware Open Source ISAs offer.

Since then, RISC-V International, hardware development community which work on the ISA's design, has been growing exponentially. RISC-V International is supported by top-level companies as NVIDIA, Google, NXP and SIFive, highly RISC-V oriented.

ISA's modularity allows high efficient and small embedded designs. Furthermore, some of the available extensions, like vector extension, V, supposed a great opportunity for top-level companies to develop cutting edge microchips for fields like AI.

1.3. Objetivos

El objetivo de este trabajo es familiarizarse con la arquitectura RISC-V y obtener un entorno de trabajo adecuado para el desarrollo de un acelerador hardware capaz de realizar operaciones aritméticas utilizando el novedoso formato Posit, introducido por John Gustafson en 2017.

1.4. Objectives

To get with RISC-V and obtain a working environment to develop a hardware accelerator capable of compute arithmetic operations using the new Posit numbers introduced by John Gustafson in 2017.

1.5. Plan de trabajo

Este trabajo, como puede verse en la figura 1.1, se puede dividir en 5 fases, de las cuales daremos una descripción breve de estás para entender el contexto en el que se ha realizado el proyecto.

- **Documentación**: punto fundamental del trabajo, donde se realizó la recopilación de información sobre el trabajo realizado.
- **Reuniones**: las reuniones con los tutores han servido como puntos de referencia para valorar y conseguir un feedback del trabajo que se iba realizando.
- **Preparación del entorno**: se analizaron y se probaron las distintas posibilidades que teníamos para poder desarrollar los aceleradores propuestos.
- **Diseño**: una vez el entorno listo, se comienza el diseño y la implementación de los módulos hardware.
- **Memoria**: el trabajo en la parte teórica ha sido continuo, unido siempre a la documentación que se iba recopilando.
- **Pruebas**: Una vez finalizado el diseño de los módulos, se pasaron pruebas de validación para comprobar la fiabilidad de los diseños.

1.6. Work Plan

The work, as is shown in figure 1.1, can be split into five sections. We will provide a brief description to understand the context where the project has been developed.

- **Documentation**: This part has been fundamental. The core of our work has been developed looking at multiple information from different sources.
- **Meetings**: Meetings have been a checkpoint for our work and an opportunity to show the work to professors and get feedback.
- **Setting the Environment**: Analysis and testings related to the development environment have been done in this stage.
- **Accelerator Design**: After some initial stages, the Verilog modules have been designed and implemented.
- **Documentation**: The text of this document have been written during the research and the design of the modules.
- **Testing**: Testing and design checking using custom tests to validate modules' integrity have been done.

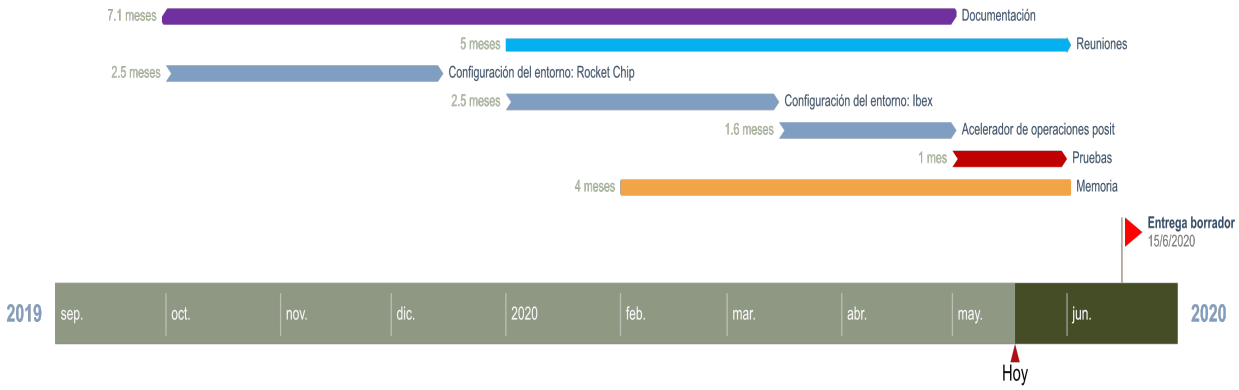

Figura 1.1: Flujo de trabajo.

1.7. Estructura

El proyecto ha sido dividido en ocho capítulos y un apéndice. Junto con el capítulo uno de introducción, se muestra un pequeño resumen del contenido del resto de ellos.

- **Capítulo 2**: En este capítulo se discuten conceptos relacionados con las ISAs disponibles actualmente, así como varias arquitecturas existentes.
- **Capítulo 3**: Un análisis en profundidad de RISC-V. Se comentan detalles de la historia de la ISA, algunos de los cores disponibles y las herramientas de desarrollo.
- **Capítulo 4**: Una pequeña introducción a Posit donde se comenta la historia, el trabajo relacionado y el estado del arte.
- **Capítulo 5**: Descripción de las herramientas utilizadas en este proyecto. Así como las pequeñas pruebas de concepto que se realizaron en el proceso inicial de desarrollo.
- **Capítulo 6**: El capítulo principal del trabajo. En él se detalla la arquitectura diseñada para los aceleradores hardware presentados.
- **Capítulo 7**: Presentación y discusión de resultados.
- **Capítulo 8**: Conclusiones del proyecto, así como posibles vías de investigación para un trabajo futuro.
- **Anexo I**: Anexo donde se incluyen ejemplos de ejecución y la explicación del uso de un script personalizado para probar los diseños hardware.

1.8. Structure

The project has eight chapters and one appendix. We will introduce each chapter with a brief explanation:

- **Chapter 2**: In this chapter some basic concepts related to the available ISAs and different architectures are covered.
- **Chapter 3**: A detailed description into RISC-V and the available tools.
- **Chapter 4**: A brief introduction to Posit, his context, work related and the state of the art.
- **Chapter 5**: RISC-V tools used in this project and explanations related to our research in this field.
- **Chapter 6**: The main core of the project. In this chapter scalar and SIMD coprocessors' designs and implementations are discussed.
- **Chapter 7**: Presentation and discussion of the results.
- **Chapter 8**: Finally we cover some conclusions looking at the results and propose some future lines of work.
- **Appendix I**: Execution examples and an explanation for the use of a custom script for test the hardware designs are described in the appendix.

Capítulo 2

Contexto ISA

En este capítulo se presentarán los antecedentes a RISC-V comparando arquitecturas RISC con arquitecturas CISC.

2.1. Clasificación

Para poder tener una comparativa clara de RISC tenemos que compararlo con otra ISA, en este caso se ha elegido CISC (Complex Instruction Set Computer).

En la tabla 2.1 podemos observar sus principales características [7]. Sobre esta tabla se comenzará el análisis antes de entrar a comparaciones enfocadas a rendimiento.

En el punto 1 tenemos una de las principales diferencias en el enfoque que tiene cada arquitectura. Por una parte CISC hará hincapié en la adaptación del hardware, mientras que RISC se centrará en el software, ya que la estrategia que seguirá irá relacionada con la adaptación de las instrucciones sencillas del programa.

En el punto 2 y 4 nos encontramos otra clave del enfoque relacionado con la reducción de tiempo por programa. RISC tendrá como objetivo reducir el número de ciclos por instrucción, estableciendo por ello instrucciones lo más sencillas posibles. Por otro lado, CISC se enfoca en la reducción de instrucciones por programa, llegando a reducir el tamaño de código pero no los ciclos por instrucción.

En el punto 3, como se mostrará más adelante, las instrucciones *Load* y *Store* están incorporadas dentro de la instrucción de CISC, mientras que en RISC están separadas como hemos señalado antes, porque el objetivo es establecer unas instrucciones lo más rápidas posible en relación con el CPI.

Para poder contrastar estas dos arquitecturas se pondrá el ejemplo de la multiplicación de dos números desde los dos distintos enfoques, haciendo uso del modelo representado en

	RISC	CISC
1.	Centrado en la parte software	Centrado en la parte hardware
2.	Single-clock	Multi-clock
3.	*Load* y *Store* son instrucciones independientes	El *Load* y *Store* ya vienen incorporadas en las instrucciones
4.	Bajo CPI pero mucho tamaño de código	Poco tamaño de código pero alto CPI

Tabla 2.1: Comparativa: Características de RISC y CISC.

la figura 2.1. En él se encuentran: una memoria principal con una serie de filas y columnas, una unidad de ejecución responsable de todos los resultados que se calculen y nueve registros representados (A, B, C, D, E, F, G, H ,I). A modo de ejemplo se desarrollara la operación multiplicación entre el valor que está en la fila 2 - columna 3 y el valor que está en la fila 5 - columna 2 para posteriormente guardarlo en la fila 2 - columna 3.

2.1.1. Enfoque de CISC

El enfoque de una arquitectura CISC es finalizar una tarea en el menor número de instrucciones por programa [1]. Esto se consigue gracias a la construcción de un hardware que sea capaz de entender y ejecutar operaciones particulares con múltiples cometidos por realizar.

Para una acción concreta podemos tener una instrucción específica para esa tarea. En el ejemplo de multiplicación expuesto se ejecuta la instrucción $Mult$, que carga los dos valores en registros separados, se multiplican los operandos en la unidad de ejecuciones y se guardan los resultados en otro registro, como se puede observarse en la figura 2.1. Es por eso, que esta tarea de multiplicación de dos números se puede representar de la siguiente forma: $Mult$ 2:3, 5:2 [2].

Con esta operación es posible manipular directamente los bancos de memoria principal sin necesidad de llamar explícitamente a operaciones *Load* o *Store*. Esto se asemeja a operaciones en un lenguaje de alto nivel.

[1] La ecuación del rendimiento es: $\frac{TIEMPO}{PROGRAMA} = \frac{TIEMPO}{CICLOS} * \frac{CICLOS}{INSTRUCCIN} * \frac{INSTRUCCIN}{PROGRAMA}$

[2] Esta notación hace referencia a la figura 2.1, donde se cogerá el valor de la columna 2 - fila 3 y el valor de la columna 5 - fila 2 de la memoria principal

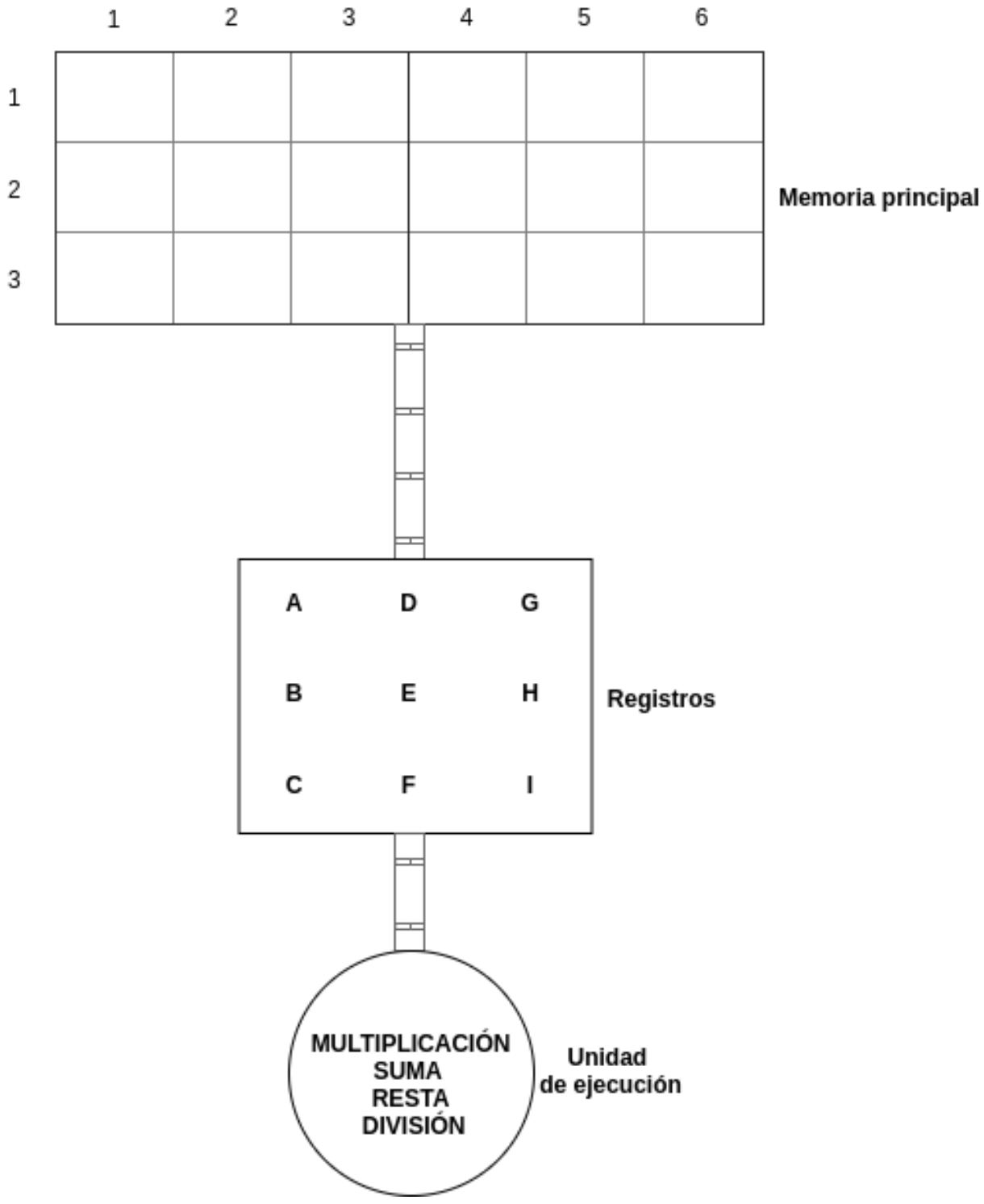

Figura 2.1: Modelo de máquina.

2.1.2. Enfoque de RISC

El enfoque para resolver el problema con una arquitectura RISC es bastante distinto. La operación sería descrita por 3 instrucciones: *Load*, *Store* y *Mult*, quedando un programa de la siguiente forma:

Load A, 2:3
Load B, 5:2
Mult A, B
Store 2:3, A

Como se puede ver, la estrategia que sigue RISC es totalmente distinta. Posee más líneas de código y, por lo tanto, necesitará más memoria RAM. Sin embargo, por otro lado, RISC requiere menos transistores que CISC debido a que todas las instrucciones ejecutadas se harán en un tiempo uniforme.

2.1.3. Comparación de rendimiento entre RISC y CISC

A continuación se describirán comparaciones entre RISC y CISC en relación a los ciclos por instrucción y fallos de lectura de cache [6].

Para poder hacer la equiparación adecuadamente se establecen dos computadoras en las cuales están implementadas RISC y CISC. La MIPS M/2000 y la VAX 8700, respectivamente.

En las tablas 2.2, 2.3 y 2.4 se pueden ver reflejadas en la columna Benchmark las distintas pruebas realizadas, pero por el momento se centrará la atención en la tabla 2.2.

Estos resultados, están relacionados con la velocidad de la computadora para ejecutar una serie de instrucciones. Se establece una media de instrucciones por cada benchmark y posteriormente se obtienen los ciclos tanto para la máquina MIPS como para la máquina VAX. Para obtener una imagen más clara de la diferencia entre estas dos se establece la proporción de ambos en la columna Ratio.

Además se extrae otra característica importante, el $RISC\ Factor$. Esta es la proporción del número de ciclos por programa y el número de instrucciones para la máquina VAX correspondido con el valor obtenido en la maquina MIPS. El factor es calculado con la siguiente fórmula: $RISC\ Factor = \frac{Ratio\ CPI}{Ratio\ Instructions}$. Los benchmarks irán ordenados de menor a mayor $RISC\ Factor$.

Una vez explicada la tabla, podemos comenzar a comentar los resultados obtenidos.

Nótese una gran diferencia de resultados entre la maquina MIPS y la maquina VAX. Las

pruebas basadas en el punto flotante obtienen en la VAX un CPI alto, debido a un hardware más complejo[3] en comparación con la MIPS.

Los tres mayores valores de *RISC Factor* se corresponden con pruebas relacionadas con números enteros, que poseen un bajo número de instrucciones de media pero un alto CPI. Por otro lado, los tres menores provienen de benchmarks donde se tiene un alto número de instrucciones medio pero un bajo CPI.

En la tabla 2.3 se analizarán los tiempos de ejecución de instrucciones de punto flotante en cada máquina. En general, el número de etapas de cada instrucción de la máquina MIPS será inferior la máquina VAX, ya que la MIPS necesitará utilizar instrucciones *Load* y *Store* donde la VAX utilizará operandos específicos. Además, la tabla muestra el número de instrucciones *Load* y *Store* por instrucción junto con el índice de cada operación en la MIPS en relación con la VAX.

Se puede apreciar que la máquina VAX casi siempre realiza más referencias de memoria. Uno de los motivos por el que esto ocurre es el escaso número de registros y la falta de ellos para punto flotante. Como regla general, los benchmarks de punto flotante hacen más instrucciones *Load* y *Store* que basado puramente en enteros. Debido a un gran número de instrucciones *Load* y *Store* por instrucción, en esta tabla no existe una correlación tan clara con el *RISC Factor*.

La tabla 2.4 tiene como objetivo el análisis del comportamiento de la cache en la dos máquinas. Los resultados obtenidos en esta tabla, por parte de la máquina MIPS, provienen de una simulación de la cache [18]. Las 3 caches tienen un tamaño de 64KB, están mapeadas y tienen una política de escritura write-through, excepto la I-cache de la máquina MIPS.

Se puede apreciar una clara relación entre el *RISC Factor* y la media de fallos en D-stream, principalmente en la VAX, donde cuanto más alto es el número de fallos promedio, más bajo es el *RISC Factor*. Es por eso que los tres benchmarks con mayor número de fallos, en las dos máquinas, son los que tienen el menor *RISC Factor*.

El I-stream es mucho menos relevante que el D-stream para casi todos los benchmarks de ambas máquinas. En concreto, el comportamiento de la cache I-stream en la máquina MIPS es excelente. Se podría llegar a considerar que la media de fallos en D-stream en la MIPS es más baja que la VAX, debido a la D-cache separada. Pero podemos observar que la MIPS en cuatro benchmarks obtiene unos resultados peores frente a la máquina VAX. Esto sucede porque el número de instrucciones *Load*, recordemos que tiene menos registros, de la VAX supera al de la MIPS, es por eso que las instrucciones *Load* adicionales son, a menudo, referencias a datos que la máquina MIPS mantiene en registros.

[3]Tanto VAX 8700 como VAX 4000/300 realizan operaciones de punto flotante en una media de 11 a 15 ciclos, el MIPS en comparación en 2 a 5 ciclos

Benchmark	**Instruc. Ratio**	**CPI**			**RISC Factor**
		MIPS	VAX	Ratio	
spice2g6	2.48	1.80	8.02	4.44	1.79
matrix300	2.37	3.06	13.81	4.51	1.90
nasa7	2.10	3.01	14.95	4.97	2.37
fpppp	3.88	1.45	15.16	10.45	2.70
tomcatv	2.86	2.13	17.45	8.18	2.86
doduc	2.65	1.67	13.16	7.85	2.96
espresso	1.70	1.06	5.40	5.09	2.99
eqntott	1.08	1.25	4.38	3.51	3.25
li	1.62	1.10	6.53	5.97	3.69
Media geométrica	2.17	1.71	9.87	5.77	2.66

Tabla 2.2: Comparativa: CPI.

Benchmark	Operaciones de punto flotante			LOADS			STORES			RISC Factor
	MIPS	VAX	Comp. MIPS (siendo VAX=1)	MIPS	VAX	Comp. MIPS (siendo VAX=1)	MIPS	VAX	Comp. MIPS (siendo VAX=1)	
spice2g6	0.034	0.083	1.02	0.09	0.94	0.25	0.04	0.14	0.65	1.79
matrix300	0.156	0.370	1.00	0.31	1.44	0.52	0.16	0.40	0.93	1.90
nasa7	0.216	0.440	1.03	0.34	1.59	0.45	0.13	0.52	0.53	2.37
fpppp	0.228	0.879	1.01	0.43	2.04	0.81	0.11	0.36	1.24	2.70
tomcatv	0.267	0.724	1.05	0.40	1.82	0.63	0.12	0.62	0.56	2.86
doduc	0.240	0.525	1.21	0.28	1.03	0.72	0.09	0.37	0.64	2.96
espresso	0	0	0	0.18	0.52	0.58	0.02	0.14	0.24	2.99
eqntott	0	0	0	0.16	0.32	0.55	0.01	0.07	0.13	3.25
li	0	0	0	0.22	0.85	0.42	0.12	0.51	0.38	3.69

Tabla 2.3: Comparativa: Operaciones.

Este efecto se hace notar cuando el I-stream de la máquina VAX no sea un factor clave en el rendimiento de la cache. En la tabla se aprecia que el pequeño número de fallos en el I-stream de la VAX proviene de pruebas donde el índice de fallos en el D-stream en la MIPS es superior o muy cercano al de la máquina VAX.

Benchmark	Fallos en lectura de cache (D-stream)					Fallos cache (I-stream)			RISC Factor
	Ratio de fallo (%)		Por instrucción		Comp. MIPS (siendo VAX=1)	Por instrucción		Comp. MIPS (siendo VAX=1)	
	MIPS	VAX	MIPS	VAX		MIPS	VAX		
spice2g6	26.9	9.1	0.0250	0.0856	0.72	0.0001	0.0089	0.03	1.79
matrix300	12.7	10.8	0.0400	0.1550	0.61	0.0000	0.0055	0.00	1.90
nasa7	12.3	8.7	0.0424	0.1390	0.64	0.0000	0.0035	0.00	2.37
fpppp	0.2	2.4	0.0007	0.0496	0.06	0.0024	0.0588	0.16	2.70
tomcatv	5.7	5.4	0.0228	0.0982	0.66	0.0000	0.0040	0.00	2.86
doduc	0.9	2.7	0.0026	0.0275	0.25	0.0031	0.0336	0.24	2.96
espresso	0.7	4.0	0.0012	0.0208	0.10	0.0002	0.0026	0.13	2.99
eqntott	3.3	4.0	0.0055	0.0128	0.46	0.0000	0.0021	0.00	3.25
li	0.6	1.8	0.0013	0.0158	0.13	0.0002	0.0103	0.03	3.69

Tabla 2.4: Comparativa: Cache.

Capítulo 3

Introducción a RISC-V

En la actualidad el desarrollo hardware es complejo debido a un oligopolio en el actual mercado, donde prima lo propietario a lo libre. Es por eso que al tener este hándicap nos obliga, para evitar el pago de royalties, a llevar el desarrollo de todo nuestro proyecto en una arquitectura de conjunto de instrucciones basado en hardware libre, RISC-V.

A continuación se realizarán una serie de comparaciones [3] relacionadas con las distintas ISA RISC no propietarias.

En la tabla 3.1 se establecen diferentes ISA: SPARC V8, OpenRISC y RISC-V, junto con ellas, se muestran una serie de características propias de cada una. Observando detenidamente se puede apreciar como que RISC-V es la más completa, ya que soporta direcciones de 128 bits. Además, también cuenta con distintas bases y extensiones que la permiten modularidad.

3.1. Historia de RISC-V

RISC-V [2, 25] fue un proyecto que comenzó en 2010 en la Universidad de Berkeley, pero la gran mayoría de los colaboradores actuales de este proyecto no pertenecen a la propia universidad. Además, existen distintas implementaciones software que son bastante útiles para el desarrollo con RISC-V.

Después de la creación de RISC-V muchos proyectos hicieron uso de ella, entre ellos se en-

	Base+Ext	Compact Code	Quad FP	Direcciones			Software			
				32 bits	64 bits	128 bits	GCC	LLVM	Linux	QEMU
SPARC V8			**X**	**X**			**X**	**X**	**X**	**X**
OpenRISC				**X**	**X**		**X**	**X**	**X**	**X**
RISC-V	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Tabla 3.1: Comparativa: Open ISA basadas en RISC.

Nombre	Descripción	Estado	Nº Instrucciones
RV32I	ISA 32 bits	En pausa	49
RV32E	Embedded - ISA 32 bits, 16 registros	Abierto	49
RV64I	ISA 64 bits	En pausa	14
RV128I	ISA 128 bits	Abierto	14

Tabla 3.2: Bases RISC-V.

Nombre	Descripción	Estado	Nº Instrucciones
M	Extensión para multiplicación y división de entero	En pausa	8
A	Extensión para instrucciones atómicas	En pausa	11
F	Extensión para (precisión simple) punto flotante	En pausa	25
D	Extensión para (precisión doble) punto flotante	En pausa	25
G	Abreviatura para la base y extensiones anteriores	Abierto	N/A
Q	Extensión para (precisión cuádruple) punto flotante	En pausa	27
L	Extensión para punto flotante decimal	Abierto	Sin definir
C	Extensión para instrucciones comprimidas	En pausa	36
B	Extensión estándar para manipulación de bits	Abierto	42
J	Extensión estándar para lenguajes traducidos dinámicamente	Abierto	Sin definir
T	Extensión estándar para Memoria Transaccional	Abierto	Sin definir
P	Extensión estándar para Empaquetado-SIMD Instrucciones	Abierto	Sin definir
V	Extensión estándar para Operaciones de Vector	Abierto	186
N	Extensión estándar para interrupciones de nivel de usuario	Abierto	3
H	Extensión estándar para hipervisor	Abierto	2
S	Extensión estándar para instrucciones de supervisor	Abierto	7

Tabla 3.3: Extensiones RISC-V.

cuentran programas de investigación financiados por DARPA (Defense Advanced Research Project Agency). El concepto Open Source hace posible que se consigan muchos beneficios en el campo de la investigación, ya que reduce el coste de desarrollo de sistemas militares y permite al gobierno construir sus propias implementaciones a bajo coste.

3.2. Bases y extensiones

Una particularidad de RISC-V, expuesta antes, son las diferentes bases y extensiones que se pueden llegar a tener. Las distintas bases y extensiones se encuentran representadas en las tablas 3.2 y 3.3. En ambas se puede ver el número de instrucciones que corresponden con esa base o extensión y el estado en el que se encuentran. La mayoría se encuentran aún en desarrollo.

3.3. Herramientas para desarrollar

3.3.1. Rocket Chip

Uno de los conjuntos de herramientas más beneficiosos al progreso de RISC-V es el repositorio de Rocket Chip [1]. Este repositorio es una colección de herramientas para la generación y la validación de SoCs y de herramientas de descripción RTL.

Entre ellas, Rocket Chip Generator es una herramienta que permite la generación de SoCs personalizados haciendo uso de Chisel, un lenguaje de descripción hardware basado en Scala.

La estructura y división del repositorio es la siguiente:

- **Chisel**: Rocket Chip Generator lo utiliza para describir RTLs.
- **Firrtl**: Representación intermedia del RTL generado con Chisel.
- **Hardfloat**: Módulo utilizado para generar unidades de punto flotante parametrizadas con el estandar IEE 754-2008. Estas unidades se utilizarán para diversas operaciones.
- **Rocket Tools**: Se comentarán más adelante.
- **Torture**: Módulo encargado de generar y ejecutar instrucciones aleatorias para hacer pruebas de stress en cores y en módulos no relacionados con el ellos.

Rocket Tools es un repositorio utilizado en Rocket Chip Generator. Este meta-repositorio contiene una serie de herramientas compatibles para el funcionamiento del Rocket Chip Generator.

- **Spike**: Simulador de la ISA RISC-V
- **RISCV-Tests**: Batería de pruebas para la simulación de la ISA
- **RISCV-OPCodes**: Lista de opcodes posibles que pueden ejecutar en el simulador
- **RISCV-PK**: Contiene un bootloader y un proxy kernel.

3.3.2. Ibex

Arquitectura

Las implementaciones realizadas para incorporar los aceleradores a un core RISC-V, han sido realizados en Ibex [20], un eficiente core in-order de 32 bits. Dicho core esta formado por 2 etapas de pipeline, como se puede ver en la figura 3.1).

Fue desarrollado en 2015, bajo el nombre “Zero-riscy” como parte de la plataforma PULP. La gran mayoría del código fue desarrollado para simplificar el RV32 CPU Core “RI5CY” y

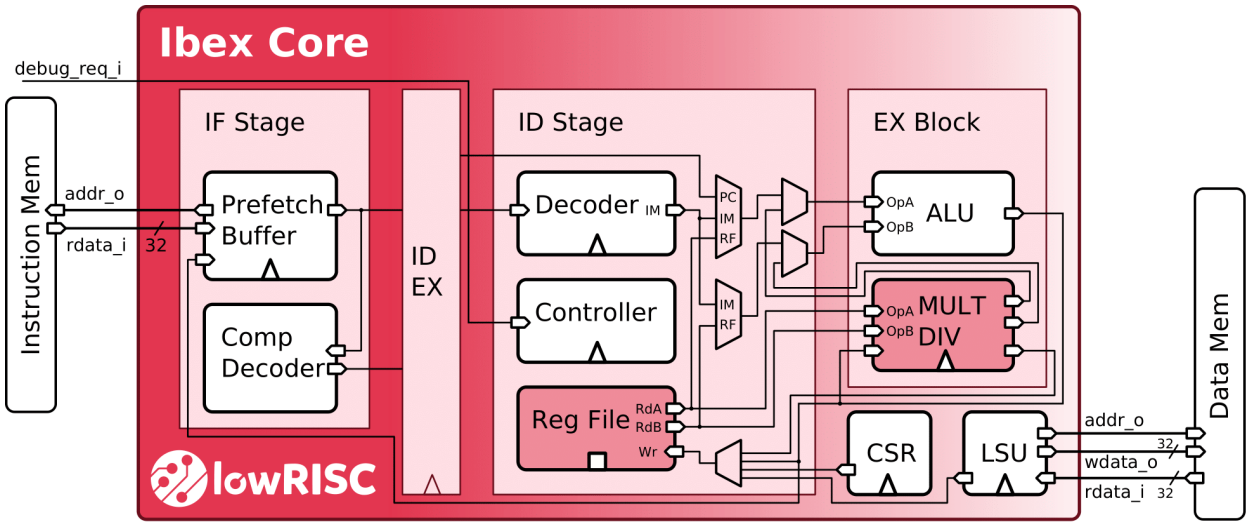

Figura 3.1: Diagrama de Ibex.

demostrar cuánto se podría reducir el tamaño del hardware. Posteriormente con la intención de reducirlo más, se dio soporte a la extensión "E" que fue añadida a un core bajo el nombre de "Micro-Riscy". En el ecosistema de PULP este core es usado como módulo de control para PULP, PULPino y PULPissimo, cores PULP, más complejos, con diferentes especificaciones según el campo al que van destinados.

En diciembre del 2018 la organización sin ánimo de lucro LowRISC, fundada en Cambridge, recogió el testigo del desarrollo de "Zero-Riscy" y cambió el nombre a "Ibex".

Volviendo a la figura 3.1 observamos la sencillez con el que esta diseñado. Se pueden observar las etapas más básicas como Fetch, Decode y Execution, sin embargo, Ibex carece de etapas de acceso a memoria y de Write Back. Incluye, además, un CSR (Control and Status Register Block) y un LSU (Load-Store Unit).

Organización del Proyecto

El directorio de Ibex se divide en múltiples carpetas, de las cuales se comentarán las más destacadas:

- **examples**: Contiene los distintos ejemplos base que se tienen para poder ejecutar Ibex, entre ellos está simple_system.
- **rtl**: Se encuentran todos los módulos en SystemVerilog para poder crear Ibex.
- **shared**: Se encuentran los ficheros en System Verilog comunes relacionados con las implementaciones para pruebas y otros escenarios.

- **syn**: Contiene scripts para usar Sv2v junto con Yosys.

Capítulo 4

Introducción a Posit

4.1. Antecedentes

El uso de la computación en múltiples aspectos de nuestra vida y el incremento del uso de las técnicas de inteligencia artificial en nuestro día a día, requiere, cada vez más, no sólo de cálculos intensivos, sino también de exactitud en los mismos.

Dicha exactitud está íntimamente ligada a la propia estructura numérica con la que se opera y que desde inicios del siglo XX lidera el punto flotante. Sin embargo, John L. Gustafson et al. [11, 12, 13], han definido durante estos últimos años formatos numéricos modernos que pretenden aportar mayor precisión que el punto flotante.

4.2. Unum Type I

Algunas debilidades del punto flotante son: el desbordamiento en infinito o en cero, permitir valores indeterminados llamados NaN, Not a Number, y el uso ineficiente, en ciertas situaciones, de los bits de la representación.

Estos hechos pueden suponer un gran problema en escenarios de gran nivel de precisión. Por ello, en 2013 John L. Gustafson [12] idea un bit que extenderá el formato de punto flotante IEEE 754.

Dicho bit es llamado Ubit, abreviatura de Universal Number, y se encarga de definir si la representación de un número es la exacta o si por el contrario, es una aproximación.

Además, Gustafson introduce dos campos más a la extensión que permiten que la representación sea dinámica: El campo $esize$, que indica la longitud en bits del exponente y el campo $fsize$, que indica la longitud en bits de la fracción.

La dinamización del formato junto con el bit "Ubit", permiten una representación más preci-

sa con menos bits, lo que permite ahorrar memoria. Sin embargo, dicha estructura es costosa de implementar y requiere un nivel mayor de complejidad hardware.

4.3. Unum Type II

En 2016 John L. Gustafson [11] decide abandonar su extensión "Type I" del punto flotante IEEE 754 y se centra en un diseño completamente nuevo, que le permita desarrollar las mismas ideas en una estructura menos costosa. Llama a su nuevo formato, Unum Type II.

El formato está principalmente basado en la recta proyectiva para la representación de los números reales. Dicha representación es similar a una recta pero con la peculiaridad de que el infinito, tanto negativo como positivo, convergen en un mismo punto, como se puede observar en la figura 4.1.

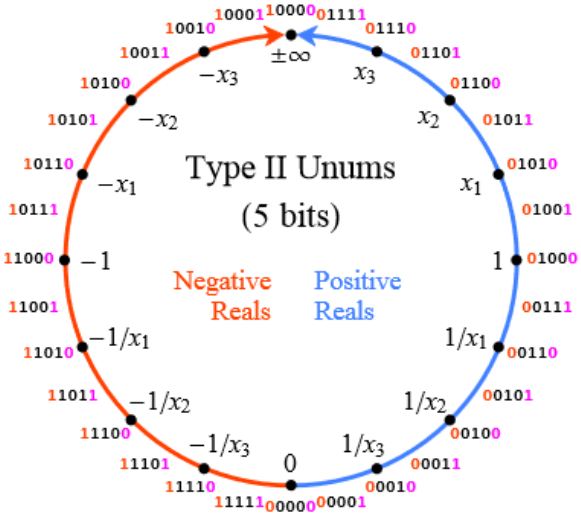

Figura 4.1: Recta Proyectiva [13].

Esta segunda aproximación de los número universales, permite un cambio drástico en el diseño hardware dedicado al cálculo numérico así como propiedades matemáticas ideales que permiten un ahorro de memoria. Sin embargo, el hardware dedicado, requiere de tablas para la mayoría de las operaciones, haciéndolo lento y costoso de implementar.

4.4. Posit o Unum Type III

Los problemas que suponen los números universales a la implementación hardware y a la velocidad de procesamiento, motivan al diseño de una estructura numérica con las virtudes

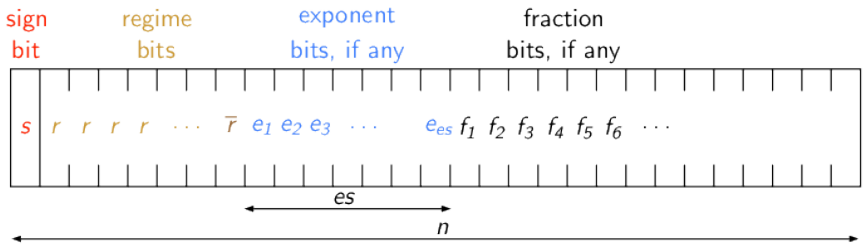

Figura 4.2: Distribución de campos Posit [13].

de los Unum II y con la facilidad de implementación hardware del punto flotante. De la unión de dichas características, nacen los Posit, aka Unum III [13].

Los números Posit se definen con dos parámetros clave: n (longitud total) y es (longitud del exponente), la notación que se utiliza para representar una configuración concreta es Posit<n, es>. La composición interna está formada por una estructura de cuatro campos: el bit de signo, los bits de régimen, los bits de exponente y los bits de fracción. En la figura 4.2 se puede ver la distribución de estos campos. Además, la ecuación 4.1 determina el valor numérico para los Posits [13, 22, 23].

$$x = \begin{cases} 0, & p = 0 \\ \pm\infty, & p = -2^{n-1}, \\ sign(p) * useed^k * 2^e * (1+f), & all\ other\ p. \end{cases} \tag{4.1}$$

Si procedemos a desglosarla, $useed = 2^{2^{es}}$ es calculado con el campo de régimen, que sigue una codificación "Run-Length". Con la ayuda de la figura 4.3 y 4.4 podemos ver los distintos valores de $useed$ y k; el término f hace referencia al campo $fraction$ y el término e al valor del campo $exponent$.

Para calcular el término k se tiene que hacer a través de m, que es la longitud de bits iguales consecutivos hasta el cambio de bit:
Si régimen es $\mathbf{0} \cdots \mathbf{0}1$, entonces $k = -m$
Si régimen es $\mathbf{1} \cdots \mathbf{1}0$, entonces $k = m - 1$

Para comprender mejor el desglose y la estructura que sigue Posit se propone el siguiente ejemplo: siendo el número Posit<16,2> **0x3333** (en binario: 0011 0011 0011 0011), el número Posit<16,2> **0x4444** (en binario: 0100 0100 0100 0100) y el número Posit<16,2> **0x4711** (en binario: 0100 0111 0001 0001) (en binario: 0011 0011 0011 0011); los tres con un $useed = 2^{2^{es}} = 16$ entonces:

0x3333

$sign = 0$

$regimen = 01$ (k = -1)

$e = 10$

$f = 01100110011$ (en decimal: 819)

$$Res = (-1)^0 * 16^{-1} * 2^2 * (1 + (819/2048)) = 0.34997558593$$

0x4444

$sign = 0$

$regimen = 10 (k = 0)$

$e = 0$

$f = 10001000100$ (en decimal: 1092)

$$Res = (-1)^0 * 16^0 * 2^0 * (1 + (1092/2048)) = 1.533203125$$

0x4711

$sign = 0$

$regimen = 10 (k = 0)$

$e = 0$

$f = 11100010001$ (en decimal: 1809)

$$Res = (-1)^0 * 16^0 * 2^0 * (1 + (1553/2048)) = 1.88330078125$$

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, *k*	−4	−3	−2	−1	0	1	2	3

Figura 4.3: Significado de k en el régimen [13].

es	0	1	2	3	4
useed	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

Figura 4.4: *useed* como función de *es* [13].

Las ventajas principales de los números Posits son:

- Un mayor rango dinámico que el punto flotante.
- Precisión cónica, por lo que son más precisos en el entorno de cero que el punto flotante. Por ello, los Posits son muy interesantes en DNNs, ya que los pesos suelen seguir una distribución gaussiana.

- La comparación entre Posits se realiza como la comparación de enteros, lo cual es una gran ventaja frente al punto flotante.
- Aproximaciones rápidas de funciones no elementales, como por ejemplo la sigmoide.

Los números Posit por su estructura matemática y su gran facilidad de implementación son idóneos para su uso en campos como la inteligencia artificial o la computación de alto rendimiento.

Capítulo 5

Estudio de Recursos Disponibles

El objetivo de nuestro proyecto plantea diferentes posibilidades a la hora de seleccionar la mejor plataforma para diseñar y desarrollarlo.

5.1. Berkeley's Rocket Chip

Las primeras pruebas de concepto, cuyo objetivo era el uso y la familiarización con RISC-V, se realizan con Rocket Chip Generator [4]. Para ello se utilizó una máquina en Amazon AWS, con el objetivo de facilitar el proceso de compilación que Berkeley tiene descrito para la construcción de la herramienta. Además se hace necesario, inicialmente, el uso de Synopsys VCS en un servidor (bisaurin) que nos proporcionó el grupo ArTeCS [10] para la simulación del hardware obtenido mediante la herramienta. La simulación con VCS es sustituida finalmente por la herramienta Verilator de software libre. Sin embargo, aunque se consigue la generación de un core personalizado y existe una interfaz especialmente dedicada a la integración de módulos externos, la documentación escasa y poco actualizada del proyecto, hace complicada la tarea y nos hace decantarnos finalmente por la búsqueda de otras herramientas o cores más definidos.

5.2. Ibex: Primera aproximación

Posterior análisis de los cores disponibles actualmente y de la documentación que proporcionan, se decide utilizar el core Ibex de LowRISC [21] para las primeras pruebas de concepto. Aunque la documentación es más extensa que en el caso del proyecto Rocket Chip de Berkeley, la curva de aprendizaje es bastante abrupta. Esto nos obliga a ponernos en contacto con la comunidad de desarrolladores que nos indican las mejores opciones para la realización de nuestro objetivo.

La interfaz que utiliza Ibex para el soporte de módulos externos esta basada en periféricos de entrada/salida mapeados en memoria. Estos periféricos mapean la dirección de sus registros directamente a direcciones de memoria que forman parte del core.

Para la primera experiencia desarrollando un periférico, se implementa un pequeño módulo que realiza la operación Or entre dos operandos, mapeando los registros de los operandos y la salida del circuito combinacional a la memoria del procesador. Para ello, se utiliza System Verilog en el desarrollo y diferentes ejemplos proporcionados por los desarrolladores del core.

Los resultados obtenidos en esta prueba de concepto nos permiten decantarnos por la plataforma Ibex para construir algo más complejo.

5.3. PaCoGen

Con el objetivo de generar las unidades de cálculo para Posits, se utiliza el proyecto PaCoGen [15, 16, 17]. PaCoGen es un generador de módulos hardware Posit creado por Manish Kumar Jaiswal, y Hayden K.-H. So que soporta operaciones suma, resta, multiplicación y división. Con el generador se pueden configurar n, anchura, y es, número de bits del exponente.

5.4. Julia y SoftPosit

Para la familiarización con Posit y sus operaciones disponibles, para la posterior creación de pruebas y para la exposición de resultados ha sido indispensable el uso de simuladores software Posit. Entre ellos se destaca el uso de SoftPosit [19], librería de referencia, escrita en C y el lenguaje de programación Julia [5] como herramienta de comprobación de resultados.

Capítulo 6

Módulos Aceleradores

El core Ibex, como se ha comentado antes, permite la creación de periféricos de entrada-salida de memoria mapeada, lo que permite la extensión del sistema con el diseño de aceleradores complejos.

6.1. Acelerador Escalar

Como primera aproximación se decide implementar un pequeño prototipo de acelerador, como puede observarse en la figura 6.1, utilizando solamente Ibex y PaCoGen. La arquitectura del módulo escalar consiste en tres módulos PaCoGen, véase figura 6.2, para cada operación disponible y dos registros entrada-salida para los dos operandos. Dichos registros están mapeados en memoria, lo que permite acceder a ellos usando las instrucciones *Load* y *Store* del procesador principal. En cada ciclo de reloj el modulo calcula, independientemente de los operandos, las tres operaciones disponibles y se destina a cada operación una dirección mapeada en el módulo para acceder al resultado por medio de instrucciones *Store* del procesador principal.

Figura 6.1: SoC Ibex y módulo Posit.

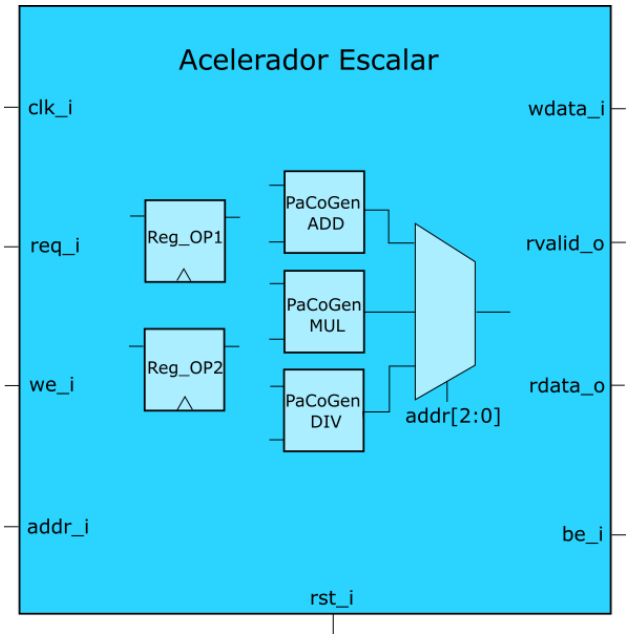

Figura 6.2: RTL del módulo escalar.

6.2. Acelerador SIMD

Nuestra última aproximación, motivada por la posibilidad de acelerar las operaciones Posit de forma paralela, es el prototipado de una arquitectura vectorial. Las necesidades de la industria y las últimas tendencias nos indican la importancia de las arquitecturas paralelas, sobre todo SIMD, Single Instruction Multiple Data.

6.2.1. Arquitectura

El acelerador diseñado es SIMD in-order y non-pipelined, que usa el formato Posit<16,2>, el cual permite la ejecución de una instrucción en, como máximo, cuatro datos de forma simultanea, utilizando el concepto de warp de las arquitecturas NVIDIA [8, 9], como grupos de threads. Su arquitectura esta formada por una memoria global quad-port de 16-bits de anchura de datos con un almacenamiento total de 64 bytes; cuatro unidades aritmético-lógicas, muy similares al módulo escalar comentado anteriormente, formadas internamente por cuatro módulos PaCoGen, uno para cada operación soportada: suma, resta, multiplicación y división; y cuatro bancos de registros formados cada uno por un total de 16 registros. Los 16 registros de cada banco están divididos en 4 grupos de 4 registros correspondiendo un grupo por warp en ejecución.

Se utiliza el concepto de warp para ejecutar grupos de threads de forma paralela. Resultando, por ejemplo, que una ejecución de un programa que sume dos arrays de 8 posiciones, dividirá dichas 8 posiciones en 2 grupos, warps, de 4 threads que se ejecutaran de forma paralela.

El módulo de control del procesador recibe las instrucciones, organiza su ejecución y habilita o deshabilita los recursos del procesador. Posee, a su vez, un módulo de warps encargado de distribuir las operaciones Posit en hasta 4 warps de 4 threads para su posterior ejecución paralela, además de mandar a los diferentes módulos información acerca del warp ejecutado en ese momento. El módulo de warps depende del registro de Dimensión de Bloque, que es el encargado de almacenar el número total de operaciones Posit a realizar y cuyo máximo es 16. El lanzamiento a ejecución de los warp depende de un contador ascendente módulo 4.

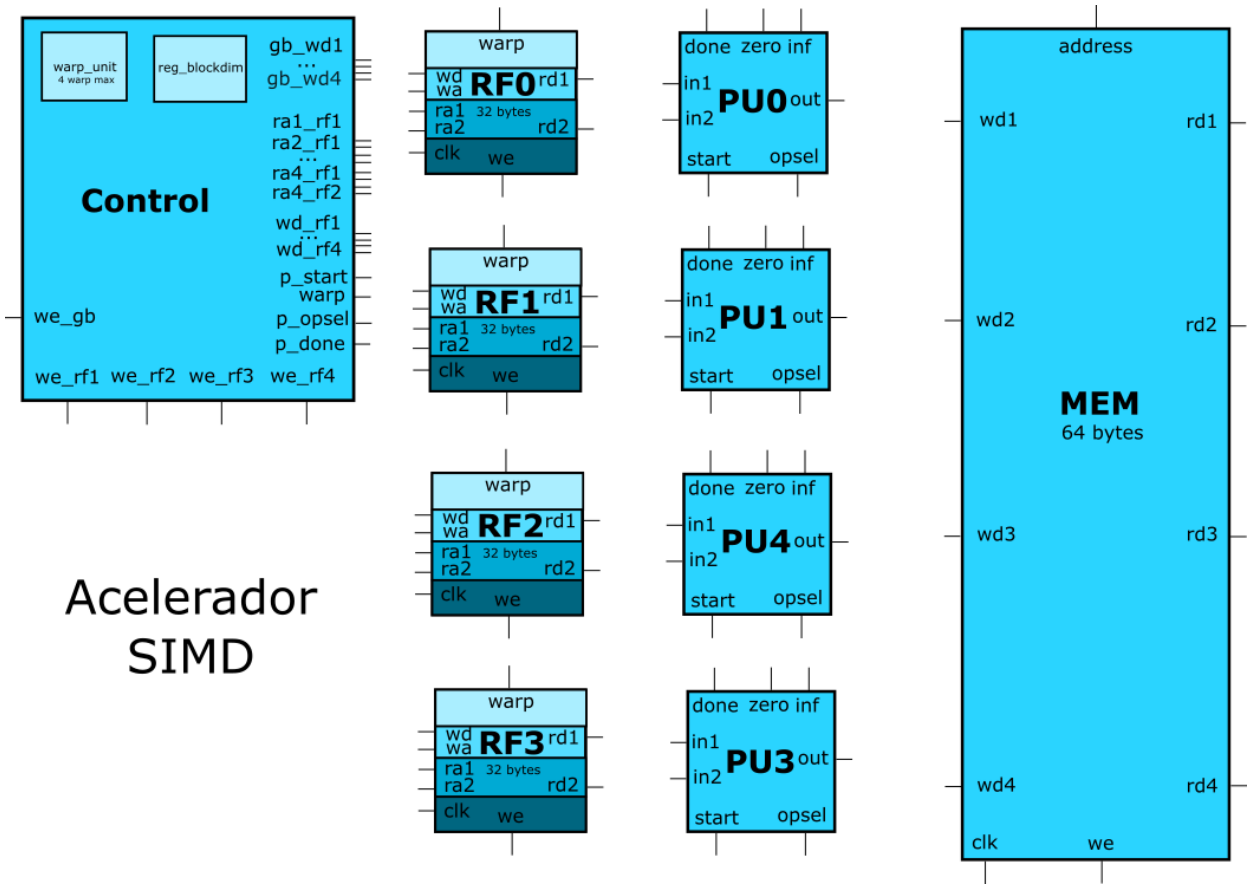

Figura 6.3: Arquitectura del módulo SIMD.

6.2.2. ISA dedicada

El acelerador diseñado tiene una ISA de 32 bits dedicada y definida por los integrantes de este trabajo. Está formada por 8 instrucciones básicas para el control del procesador. Todas las instrucciones tienen un campo único básico de 4 bits que las identifican unívocamente, opcode.

La instrucción *Load* con $opcode = 0x1$, carga en los registros la dirección de memoria

seleccionada y sus posiciones sucesivas teniendo en cuenta la dimensión del bloque. Está compuesta de un primer campo [27:20] para seleccionar el registro destino y de un segundo campo [5:0] para seleccionar la dirección de memoria global origen.

La instrucción *Store* con *opcode* = $0x2$, carga la información de los registros en posiciones de memoria sucesivas, teniendo en cuenta nuevamente la dimensión del bloque. Las instrucciones *Add* con *opcode* = $0x3$, $Mult$ con *opcode* = $0x4$ y Div con *opcode* = $0x5$ realizan las correspondientes operaciones aritméticas entre Posits. Estas instrucciones poseen tres campos para indicar los registros que contienen los operandos y el registro de escritura del resultado.

La instrucción $SetBlockDim$ con *opcode* = $0x6$ es la encargada de dar valor al registro Dimensión de Bloque. Idealmente es la primera instrucción que recibirá el acelerador.

Debido a la naturaleza del acelerador en forma de periférico de entrada/salida, son necesarias instrucciones para la escritura y la lectura en memoria global. Para ello la ISA dedicada contiene dos instrucciones más: la instrucción $MemWrite$ con *opcode* = $0x7$ se encarga de almacenar los datos recibidos en memoria. Contiene un primer campo de 8 bits para la dirección de memoria destino y un campo de 16 bits para los datos. Como se puede observar, la carga en memoria de los datos se hace de uno en uno, lo que ocasiona no poderse beneficiar de una arquitectura paralela para la carga. La instrucción $MemRead$ con *opcode* = $0x8$ se encarga de situar los datos de una dirección de memoria global en la salida del acelerador para su posterior lectura por medio de instrucciones *Store* del procesador principal.

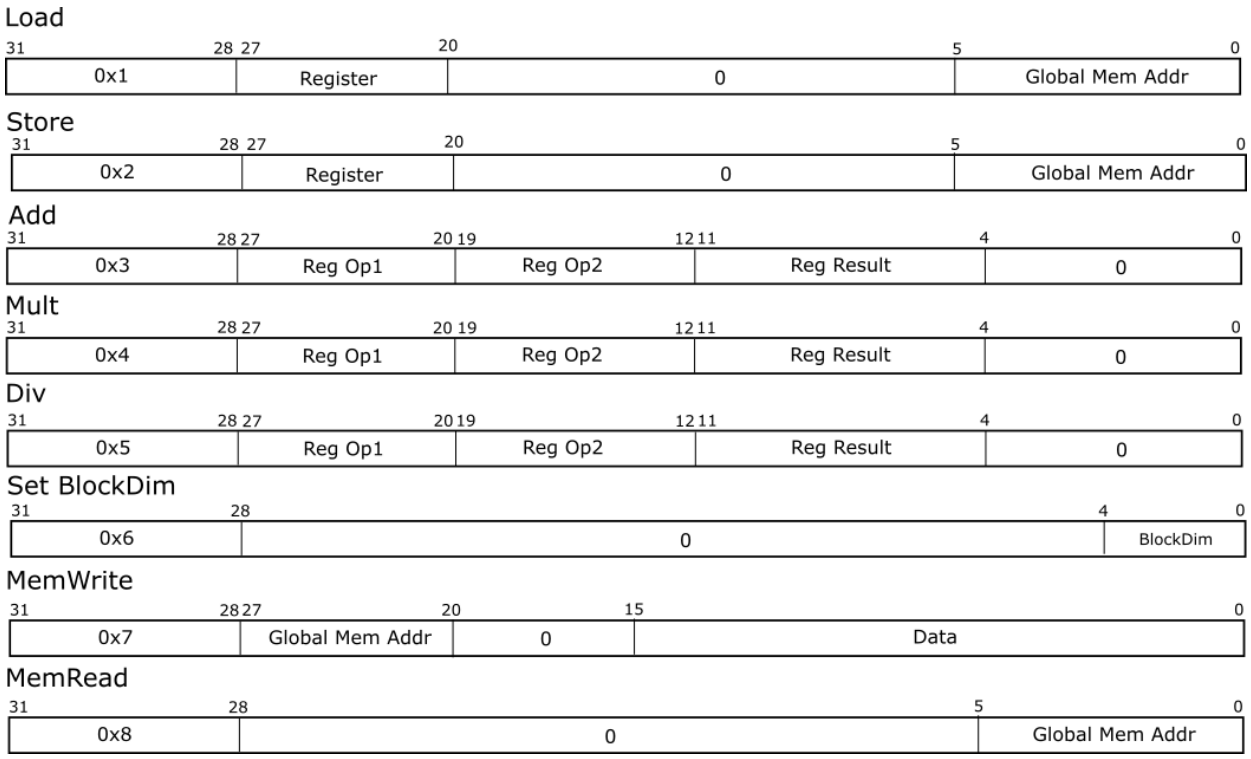

Figura 6.4: ISA personalizada.

Capítulo 7

Resultados

A continuación, presentamos los resultados obtenidos y las herramientas utilizadas para el desarrollo y prueba del diseño RTL de nuestros dos aceleradores. Nos centramos en mostrar las simulaciones gráficas obtenidas y los valores en términos de ciclos de reloj teóricos conseguidos.

7.1. Herramientas

Para el completo desarrollo de nuestros aceleradores se decide utilizar el lenguaje HDL Verilog por su sencillez y gran documentación. Además, ha sido de gran ayuda el libro de John L. Hennessy y David A. Patterson [24] y el de David Money Harris y Sarah L. Harris [14]. Se ha utilizado también System Verilog en las primeras etapas de diseño RTL.

Para la simulación de Verilog se han utilizado Icarus Verilog y Verilator. En una primera fase de diseño inicial, se decidió utilizar Icarus Verilog por su rápida configuración para proyectos pequeños. Más tarde, se utilizó Verilator ya que es el elegido por LowRISC para la compilación del proyecto Ibex.

La administración de los diferentes módulos del SoC se ha realizado con FuseSoC, una herramienta escrita en Python que permite la modularidad y el manejo de configuraciones personalizadas, así como la integración con lenguajes HDL.

La simulación gráfica ha sido posible gracias a la herramienta GTKWave, la cual se ha utilizado para depurar el comportamiento de nuestros diseños y para mostrar gráficamente resultados.

Para la realización de las pruebas se han utilizado el lenguaje C y la herramienta Make para crear los diferentes entornos. La compilación del código C se ha realizado con el compilador GCC exclusivo para RISC-V que se comenta en capítulos anteriores.

Finalmente, comentar que todo el proceso se ha realizado en una maquina virtual Debian, construida exclusivamente con las herramientas necesarias de este proyecto.

7.2. Visualización con GTKWave

Inicialmente se van a comentar los resultados obtenidos en el módulo escalar para posteriormente continuar el análisis con el módulo SIMD.

Para las visualizaciones del acelerador escalar, se ha decidido utilizar como ejemplo la suma en Posit<16,2>, entre el valor 0x3333 (0.3499...) y 0x4444 (1.5332...) cuyo resultado es 0x4711 (1.8833...). El desglose de estos números Posit se comentó en capítulos anteriores.

En la figura 7.1 se muestra el comportamiento obtenido para el módulo escalar. Las direcciones de memoria del periférico están definidas como 0x50000 para el registro del operador A, 0x50008 para el registro del operador B y 0x50010 para la obtención del resultado. Analizando la figura, en un primer instante de tiempo las señales *conv_addr_i* y *conv_wdata_i* contienen, respectivamente, la dirección de memoria del dato a manipular y los datos a escribir. Las señales *a_d*, *a_b*, *b_d* y *b_q*, entradas y salidas de los registros, conservan los operadores para su posterior utilización en el módulo PaCoGen. Cerca del instante $t = 204ps$, se realiza la operación Posit suma, calculándose y sirviéndose en la entrada del registro de salida, *rdata_d*, en el mismo ciclo de reloj. Finalmente, cuando la dirección del registro de salida es recibida, el módulo redirige la salida del registro *out* hacia el procesador Ibex para servir el resultado.

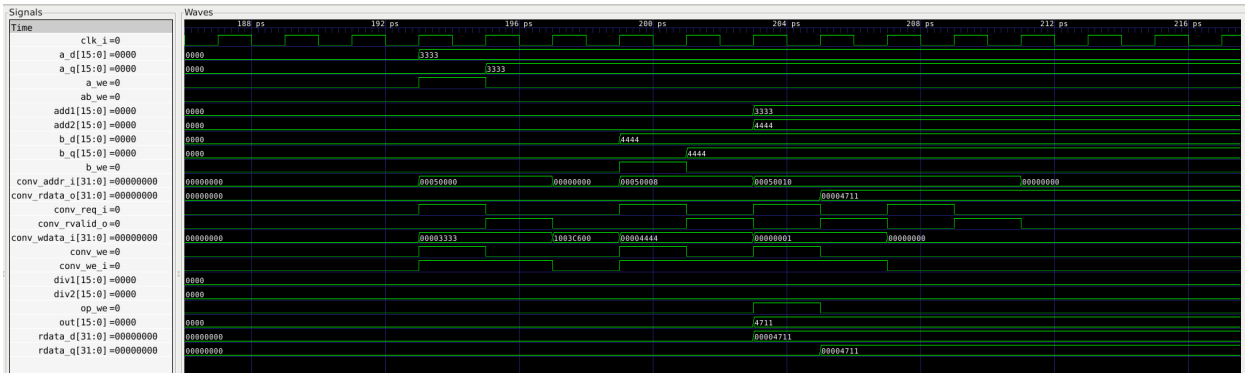

Figura 7.1: Módulo escalar en GTKWave.

Para las visualizaciones SIMD se han utilizado dos arrays de ocho posiciones, uno de ellos compuesto por ocho números Posit 0x3333, y el otro por ocho 0x4444. Se espera como resultado un array de ocho posiciones formado por Posits con valor 0x4711. La elección de los

dos valores numéricos de prueba es arbitraria.

Para el análisis de la simulación SIMD, se ha probado el comportamiento de los diferentes comandos. El test de prueba está constituido por una instrucción *SetBlockDim*; dieciséis instrucciones *MemWrite* que definen los dos arrays de 8 componentes almacenándolos de forma consecutiva en la memoria global del acelerador; dos instrucciones *Load* que asignan los valores de memoria de ambos arrays respectivamente a los registros $r0$ y $r1$; una instrucción *Add* que almacena su resultado en los registros $r2$; una instrucción *Store* para almacenar el resultado en los registros $r2$ en posiciones consecutivas de memoria y ocho instrucciones *MemRead* que se encargan de redirigir el resultado a la salida del módulo.

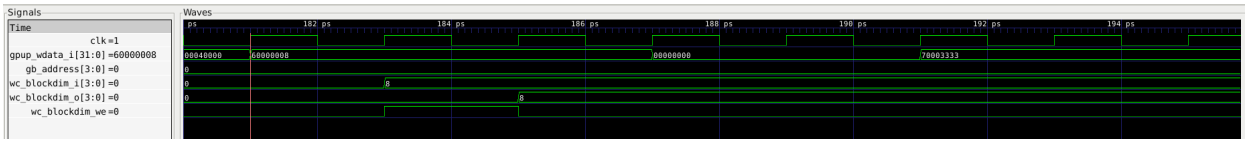

Figura 7.2: Comportamiento de la instrucción *SetBlockDim*.

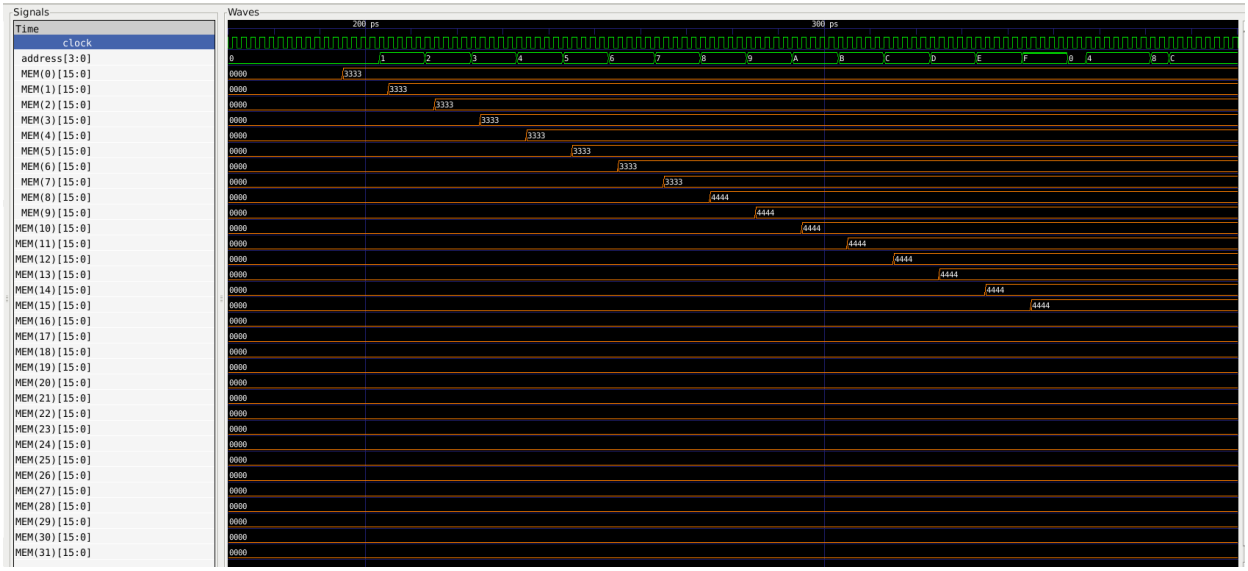

Figura 7.3: Comportamiento de la instrucción *WriteMem*.

En las siguientes figuras, se puede observar el comportamiento en simulación de algunas de las instrucciones mencionadas anteriormente. La figura 7.2 muestra la instrucción *SetBlockDim*. Del mismo modo que en la simulación realizada anteriormente para el módulo escalar, la señal *gpup_wdata_i* contiene los datos recibidos por el periférico. En el caso del acelerador SIMD, la señal de datos contiene la instrucción a ejecutar, siendo el

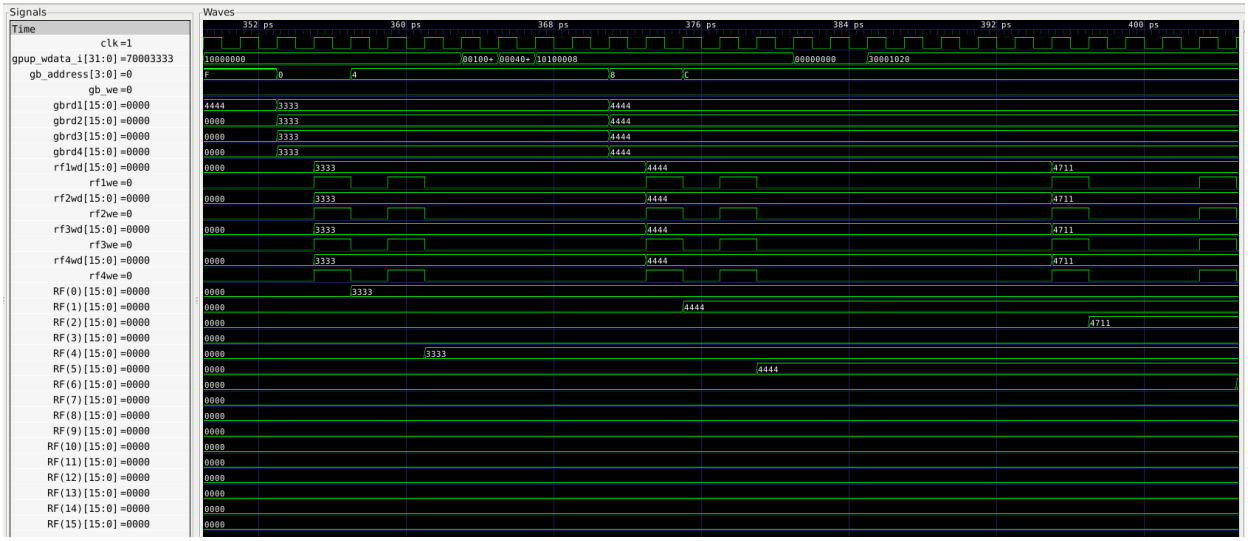

Figura 7.4: Comportamiento de la instrucción *Load*.

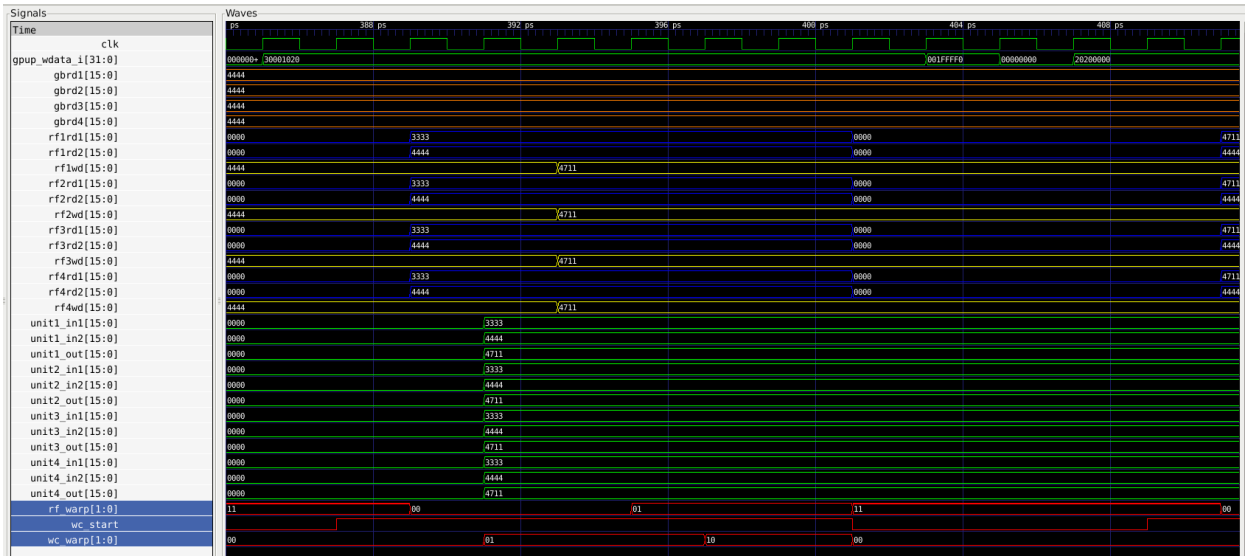

Figura 7.5: Comportamiento de la instrucción *Add*.

rango de memoria mapeada de 0x40000 a 0x4FFFF. Nótese que no hay rangos de memoria dentro del acelerador que refieran a ningún registro concreto. Volviendo al ejemplo de la figura 7.2, se observa cómo en un primer ciclo de reloj, la instrucción es analizada y posteriormente ejecutada, guardando el valor Dimensión de Bloque en el registro *wc_blockdim*.

La siguiente figura 7.3 muestra la simulación de la instrucción $WriteMem$. En la figura se pueden observar las diferentes posiciones de la memoria global, etiquetadas como MEM, así como la señal $Write\text{-}Enable$, *gb_we* y las entradas de dirección y de datos. El dato es obtenido de la instrucción e introducido como señal de escritura, *gbwd*, en ella. Finalmente

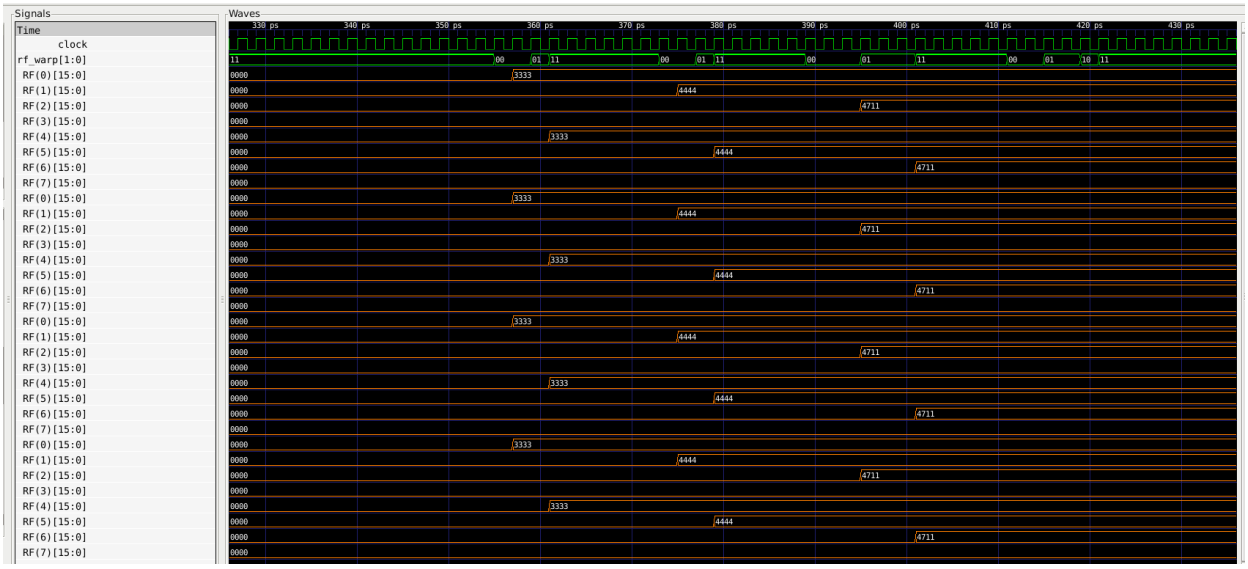

Figura 7.6: Resultado de los registros tras la instrucción *Add*.

en el siguiente ciclo de reloj el dato es almacenado en la posición correspondiente.

En las figuras posteriores, 7.4, 7.5 y 7.6, se ilustran las instrucciones *Load* y *Add* así como el resultado final en los bancos de registros. La instrucción Load mostrada en la figura 7.4, permite apreciar de primera mano el paralelismo de carga de los operandos. Se recuerda que los operandos son tratados en grupos de cuatro threads máximo. Así pues, se puede observar como de forma simultánea la memoria sirve cuatro operandos a la vez desde sus salidas *gbrd* a los registros, en concreto a las entradas de cada banco de registros. Cabe destacar que esa entrada de datos en los registros se realiza de forma simultánea y que por tanto la escritura también. La instrucción Load de un grupo de threads finaliza su ejecución en cuatro ciclos de reloj.

La instrucción *Add* ilustrada en la figura 7.5, involucra gran cantidad de señales en simulación. El control de grupos de thread es crucial para la correcta ejecución de las instrucciones aritméticas, además de las instrucciones *Load* y *Store*. La ejecución de las instrucciones aritméticas se encuentra dividida en cuatro fases: una primera fase de resolución de direcciones, una segunda fase de lectura de operandos, la tercera fase de ejecución, y una cuarta fase de almacenamiento de resultados. Se pueden observar en la figura tres señales interesantes, wc_start, wc_warp y rf_warp, encargadas, respectivamente, de activar al módulo "Warp Unit" que comenzara a utilizar el contador, de llevar la cuenta de los grupos de threads ejecutados y de indicarles a los bancos de registros en que grupo de registros van a trabajar.

Continuando con la ejecución de la operación *Add*, se puede observar cerca del instante de tiempo $t = 392ps$ cómo las señales $unit1\_in1$ y $unit1\_in2$, entradas en uno de los cuatro módulos PaCoGen, reciben las entradas correspondientes y de forma síncrona muestran

el resultado en la señal $unit1\_out$. En el siguiente ciclo de reloj es almacenado en el registro correspondiente.

En la figura 7.6, se pueden observar siete posiciones de cada banco de registros, en concreto de las posiciones 0, 1 y 2, que contienen los dos operandos y el resultado final de la operación. Destaca en la figura cómo los valores provenientes de las unidades Posit son almacenadas de forma paralela y cómo la señal rf_warp interviene en la selección del grupo de registros apropiado en cada banco. En el primer grupo de resultados, antes del instante de tiempo $t = 480ps$, rf_warp indica que el warp ejecutado es el cero, lo que permite seleccionar el grupo 0 de los bancos de registros. Por lo que los resultados correspondientes se guardaran en el registro 2, como indica la instrucción *Add*, del grupo 0 del banco de registros. Sin embargo, si se observa el segundo grupo de resultados, se aprecia que el warp es el 1, lo que desplaza el almacenamiento del resultado hasta el registro 2 del grupo 1 del banco de registros.

Las instrucciones *Store* y $ReadMem$, así como las operaciones aritméticas $Mult$ y Div, son similares a las mostradas en las anteriores figuras y, por simplicidad en el documento, no se muestran visualmente. Destacar cómo la instrucción $ReadMem$ es más lenta con respecto a $WriteMem$ debido a operaciones adicionales que realiza Ibex.

7.3. Análisis del rendimiento en simulación

El propio Ibex y sus pruebas de ejecución están dotadas de un pequeño analizador de ciclos de reloj de simulación que es de gran ayuda para medir la eficacia del acelerador.

Para el análisis del rendimiento en simulación se han utilizado tres pruebas diferentes: una primera prueba, escrita con la librería de referencia SoftPosit, que opera los dos mismos arrays que en el test SIMD realizado con GTKWave; una segunda prueba para el módulo escalar, muy similar al utilizado para la simulación con GTKWave, que incorpora 7 operaciones iguales adicionales para la mejor comparación con el acelerador SIMD; finalmente una tercera prueba destinada a la plataforma SIMD, igual a la usada en la simulación con GTKWave.

Los ciclos mostrados no solo forman parte de la ejecución en los correspondientes aceleradores sino que son producto de la completa simulación junto con el core RISC-V.

Se observa cómo la diferencia entre los dos aceleradores y la versión software SoftPosit es extremadamente alta, lo que expone unos muy buenos resultados en cuanto a la aceleración del cálculo se refiere, incrementando su rendimiento más del doble.

Sin embargo, la figura también muestra deficiencias en el diseño del acelerador SIMD, que no consigue alcanzar un rendimiento mayor que el acelerador escalar. Este resultado, tras

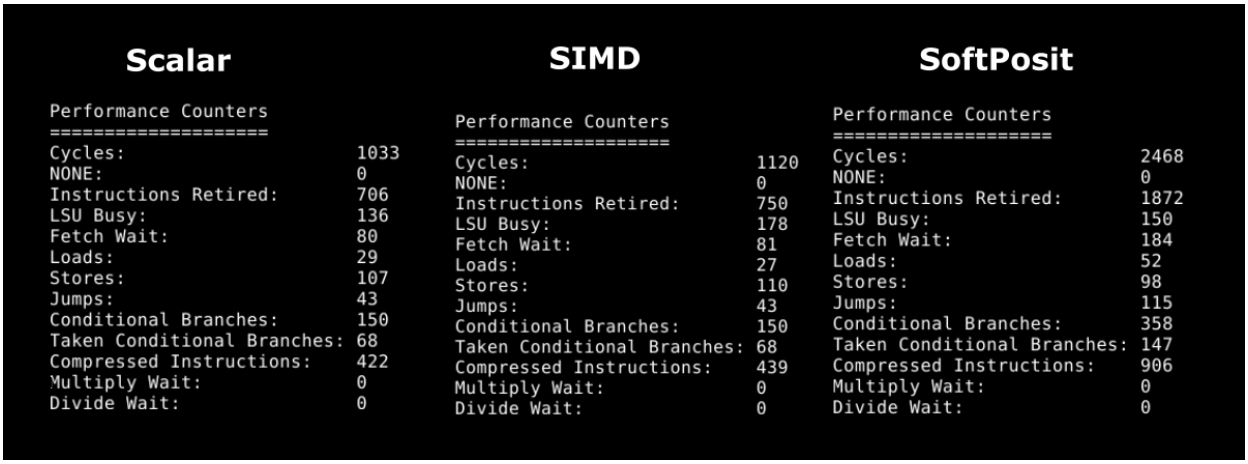

Figura 7.7: Análisis del rendimiento en simulación con la herramienta de Ibex.

analizar a fondo la solución, es provocado por el consumo de ciclos de reloj procedente tanto de la interpretación de diferentes instrucciones como de las diferentes fases de ejecución utilizadas para la carga de registros o resolución de direcciones. Tras los resultados, creemos que agregar pipelines al diseño es una firme solución para incrementar con creces el rendimiento del módulo SIMD. Esto permitiría dividir la ejecución en diferentes fases, pudiendo ejecutar varias instrucciones de forma segmentada. Para ello será importante controlar las señales de Ibex a la hora de mandar instrucciones al acelerador.

Capítulo 8

Conclusiones

Desde el inicio del desarrollo hardware la industria ha estado dominada por arquitecturas cerradas sin posible acceso al público corriente. RISC-V y su comunidad hacen posible el desarrollo de nuevo hardware y el diseño de SoCs dedicados, usando cores de libre uso que de otra forma serían imposible obtener.

The industry has been dominated by closed architectures without any possible access for common people. RISC-V and his community works in a new hardware paradigm which make possible the design of dedicated SoCs using open cores already developed.

8.1. Discusión de Resultados

En este trabajo se ha analizado la ISA RISC-V, desde sus inicios hasta lo que hoy en día supone para la comunidad de desarrolladores. Se han analizado algunas de las diferentes opciones hardware que existen para la implementación de la ISA y se ha hecho un análisis de dichas plataformas con el objetivo de seleccionar una de ellas para la implementación de un acelerador personalizado que permitiese el cálculo con números Posit. A la vista de los resultados se puede concluir que la plataforma RISC-V es un gran candidato para el desarrollo de hardware, para el aprendizaje en detalle de la arquitectura de microprocesadores libres disponibles, así como una excelente elección para el desarrollo de SoCs para plataformas específicas que necesiten de módulos o aceleradores personalizados.

Los aceleradores hardware diseñados en este trabajo han mejorado, en términos de rendimiento, a la implementación software para cálculos con números Posit, SoftPosit. Sin embargo, hemos observado que el diseño del acelerador SIMD no consigue ser más eficiente que la versión escalar. No obstante, confiamos que aplicando segmentación el rendimiento del acelerador mejore exponencialmente.

8.2. Results and Discussion

In this final project the ISA of RISC-V is exposed. Some different hardware options which implement RISC-V have been discussed, as well as a platform selection for implement a coprocessor capable of doing Posit parallel computations.

Looking at the results, we can conclude that RISC-V is a good candidate for hardware development, for computer architecture learning, as well as an excellent option for SoC's design.

Furthermore, the designed hardware accelerators have improved the main software implementation for Posit calculations, SoftPosit. However, we have observed that the design of the SIMD accelerator fails to be more efficient than the scalar version. Nevertheless, we are confident that by applying segmentation the performance will improve dramatically.

However, a poor design in the SIMD accelerator makes this parallel architecture inefficient and in need of new features and a redesign.

8.3. Trabajo futuro

Finalmente, comentamos algunas de las posibles líneas de investigación futuras así como la mejora del diseño de los aceleradores. Nos hemos centrado, sobre todo, en la implementación física de los diseños realizados para poder obtener una visión más certera de RISC-V como firme candidato.

A la vista de los resultados, es necesario un rediseño de la arquitectura SIMD mostrada, con el objetivo de eliminar posibles retardos y hacerla más eficiente. Agregar etapas pipeline al acelerador es claramente una de las metas, así como la implementación de una ISA más formal como la extensión V de RISC-V.

Además la finalización del ciclo de desarrollo hardware en una FPGA puede ser una posible línea de trabajo. Las etapas de síntesis, enrutado y emplazamiento, nos darían un comportamiento más aproximado a la realidad en términos de rendimiento, junto con una idea más cercana de cómo el acelerador ayudaría en proyectos que requieran cálculos Posit.

8.4. Future Work

Finally, we want to discuss some of the possible work lines for the future, looking at improvements in our accelerators. Also we will discuss some future work for the design on an FPGA.

Looking at the results, the SIMD architecture should be restructured to remove delays and improve performance. Adding pipeline stages could be a good starting point, as well as

the use of a more formal ISA like RISC-V's V extension.

Furthermore, using an FPGA to finish the hardware development process could be a possible line of work. Synthesis and Place and Routing stages would be a good opportunity to measure performance and get a better idea of how our design will help in Posit relating projects.

Contribución

Para poder explicar las distintas contribuciones generadas por los colaboradores del Trabajo de Fin de Grado se comentará cada capítulo de la memoria y se indicará cuál ha sido la contribución tanto en la parte práctica como teórica.

La introducción del trabajo y su resumen fue realizado en conjunto por los dos integrantes. La motivación fue escrita por Jaime del Rey y los objetivos y el plan de Trabajo desarrollado por Ángel Cruz. Además la estructura del trabajo ha sido desglosada por ambos.

El contexto ISA presentado, dividido en dos partes: comentarios acerca de los dos enfoques actuales y las comparaciones entre ambos fue realizado respectivamente por Jaime del Rey y Ángel Cruz. El estudio de los contextos fue estudiado conjuntamente para obtener una buena base de las arquitecturas a tratar, leyendo juntos las especificaciones.

En referente a la historia y los cores disponibles de RISC-V ha sido una tarea conjunta de investigación durante meses leyendo documentos y libros actuales. Refiriéndonos a la historia, Ángel Cruz ha desarrollado en la memoria su contenido y las extensiones disponibles. En cuando a Jaime del Rey, su contribución ha estado más cercana a las diferentes herramientas y cores utilizados en el trabajo.

Tras varias explicaciones por parte de los tutores de este trabajo y ciertas lecturas recomendadas, ambos integrantes obtienen conocimiento acerca de los números Posit y sus antecedentes. En el capítulo de la memoria referente a Posit, Jaime del Rey ha escrito sobre los antecedentes previos y sobre el Unum Type I. Ángel Cruz ha concretado la información estudiada referente a los números Unum Type II y Type III.

Habiendo ya estructurado la introducción de Posit, entramos en el capítulo donde se explicarán los recursos utilizados. En esta parte se ha trabajado colectivamente con el objetivo de conseguir el máximo conocimiento en relación a la herramientas utilizadas.

Para Berkeley's Rocket Chip como se ha comentado, se ha montado un entorno en Amazon Web Services donde Jaime del Rey ha sido el encargado de la configuración previa y Ángel Cruz se ha encargado de la instalación de dependencias necesarias para el desarrollo en RISC-V y Rocket Chip. Tras la construcción de la máquina virtual, se ha utilizado

conjuntamente a la hora de la investigación relativa a Rocket Chip. Se tomó la decisión conjuntamente de abandonar el desarrollo con la plataforma de Berkeley por la ausencia de documentación. La descripción del trabajo en la memoria ha sido escrita por Jaime del Rey.

Con Ibex se ha seguido la misma línea de desarrollo, donde se ha investigado paralelamente el propio core. Gracias a la ayuda de la comunidad de Ibex, se ha podido continuar con la labor de desarrollo de RISC-V. Cada integrante desarrolló individualmente, al unísono, un pequeño prototipo para una prueba de concepto en el core Ibex. Finalmente, fueron expuestos los dos y se seleccionó el más adecuado. La descripción de Ibex en la memoria ha sido escrita por Ángel Cruz.

Por último, las herramientas software utilizadas, PACoGen, Julia y SoftPosit fueron analizadas y descritas en la memoria colectivamente.

Concretando la contribución en los módulos aceleradores, el diseño integro de ambos aceleradores ha sido realizado por ambas partes del proyecto, de forma simultanea y conjunta, después de la previa lectura y recopilación de información de documentos explicativos de diferentes arquitecturas. La implementación del código Verilog para el módulo escalar, ha sido realizado por Ángel Cruz. En cuanto al acelerador SIMD, fue programado por Jaime del Rey. Posteriormente, para una mayor calidad de testeo, las pruebas se realizaron por los integrantes que no habían participado en la implementación, respectivamente Jaime del Rey probó el módulo escalar y Ángel Cruz el módulo SIMD.

Finalmente, el anexo y las conclusiones han sido realizadas y plasmadas en la memoria por ambos integrantes con el fin de poder representar al máximo, el conocimiento adquirido a lo largo del trabajo como grupo.

Anexos

Apéndice A

Anexo I: Script de Ejecución

Para facilitar las pruebas de las operaciones Posit y para poder obtener de una forma más sencilla los datos, se ha creado un script llamado *tb.sh* que servirá como punto de inicio para las personas desconocedoras del código del acelerador y su implementación en Ibex. El script está escrito en Bash y en Python y permite ejecutar test para los diferentes módulos aceleradores, así como código implementado con SoftPosit en Ibex.

Internamente se ejecuta la compilación del core y de las pruebas, su ejecución y un script en Python que extraerá la información relevante y la guardará en un archivo llamado $speed\_file.csv$. Por otro lado, se obtiene también el archivo $.fst$ para poder utilizarlo con GTKWave.

Antes de ejecutarlo hay que descargar el core Ibex disponible en el repositorio de LowRisc, las dependencias que se indican en él y añadir al Path del sistema la version del GCC para RISC-V. [1].

En las figuras A.1 y A.2 se pueden observar distintas ejecuciones del módulo escalar con el script. En el ejemplo de la figura A.1 se ejecuta en el acelerador escalar, con una configuración de Posit<32, 2>, la suma de 16.1111 (0x60071c43) con 16.1111 (0x60071c43), cuyo resultado es 32.2222 (0x64071c43). En el ejemplo de la figura A.2 se ejecuta, con una configuración de Posit<32, 2>, la multiplicación de 16.1111 (0x60071c43) con 16.1111 (0x60071c43), cuyo resultado será 259.56754302978... (0x70072295).

Las figuras A.3 y A.4 muestran un ejemplo de uso del script con la ejecución de la prueba para el módulo SIMD realizada en el capítulo 6.2 y su código correspondiente. Como puede observarse, se mostrará la información relacionada con los tiempos de ejecución y otros datos relevantes.

[1] `export PATH=`$PATH : /opt/riscv/bin/|$

```
RECUERDE: Escriba la ruta donde se encuentra riscv32-unknown-elf-gcc en PATH
export PATH='$'PATH:[RUTA_RISCGCC]
Introduzca el Path de Ibex
/home/jaime/ibex

[1]Scalar [2]SIMD [3]SoftPosit
-----------
1
$: Scalar Coprocessor
Seleccione una N (32, 16 o 8)
$ (INT): 32
Ahora seleccione un num. exponencial (0, 1 o 2)
$ (INT): 2
Introduzca el valor 1 POSIT
$ (HEX): 0x60071c43
Introduzca el valor 2 POSIT
$ (HEX): 0x60071c43
¿Qué operación quiere hacer? (0x1=ADD, 0x2=MUL, 0x3=DIV)
$ (HEX): 0x1
Compilando Scalar...
Ejecutando Scalar...
Extrayendo información...
OUTPUT:
64071C43
```

Figura A.1: Ejecución ejemplo 1.

```
RECUERDE: Escriba la ruta donde se encuentra riscv32-unknown-elf-gcc en PATH
export PATH='$'PATH:[RUTA_RISCGCC]
Introduzca el Path de Ibex
/home/jaime/ibex

[1]Scalar [2]SIMD [3]SoftPosit
-----------
1
$: Scalar Coprocessor
Seleccione una N (32, 16 o 8)
$ (INT): 32
Ahora seleccione un num. exponencial (0, 1 o 2)
$ (INT): 2
Introduzca el valor 1 POSIT
$ (HEX): 0x60071c43
Introduzca el valor 2 POSIT
$ (HEX): 0x60071c43
¿Qué operación quiere hacer? (0x1=ADD, 0x2=MUL, 0x3=DIV)
$ (HEX): 0x2
Compilando Scalar...
Ejecutando Scalar...
Extrayendo información...
OUTPUT:
70072295
```

Figura A.2: Ejecución ejemplo 2.

```
//WRITE BLOCKDIM = 8
DEV_WRITE(POSIT_COOPROC_BASE, 0x60000008);

//WRITE IN MEMORY
DEV_WRITE(POSIT_COOPROC_BASE, 0x7000FFFF);
DEV_WRITE(POSIT_COOPROC_BASE, 0x7010FFFF);
DEV_WRITE(POSIT_COOPROC_BASE, 0x7020FFFF);
DEV_WRITE(POSIT_COOPROC_BASE, 0x7030FFFF);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70402222);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70502222);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70602222);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70702222);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70803333);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70903333);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70A03333);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70B03333);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70C04444);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70D04444);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70E04444);
DEV_WRITE(POSIT_COOPROC_BASE, 0x70F04444);

//LOAD
DEV_WRITE(POSIT_COOPROC_BASE, 0x10000000);
DEV_WRITE(POSIT_COOPROC_BASE, 0x10100008);

//POSIT OP = 3 (ADD)
//4 (MUL) 5 (DIV)
DEV_WRITE(POSIT_COOPROC_BASE, 0x30001020);

//STORE
DEV_WRITE(POSIT_COOPROC_BASE, 0x20200000);

//READ FROM MEMORY
DEV_WRITE(POSIT_COOPROC_BASE, 0x80000000);
puthex(DEV_READ(POSIT_COOPROC_BASE, 0));
puts("\n");
DEV_WRITE(POSIT_COOPROC_BASE, 0x80000001);
puthex(DEV_READ(POSIT_COOPROC_BASE, 0));
puts("\n");
DEV_WRITE(POSIT_COOPROC_BASE, 0x80000002);
puthex(DEV_READ(POSIT_COOPROC_BASE, 0));
puts("\n");
DEV_WRITE(POSIT_COOPROC_BASE, 0x80000003);
puthex(DEV_READ(POSIT_COOPROC_BASE, 0));
puts("\n");
DEV_WRITE(POSIT_COOPROC_BASE, 0x80000004);
puthex(DEV_READ(POSIT_COOPROC_BASE, 0));
```

Figura A.3: Código para el acelerador SIMD.

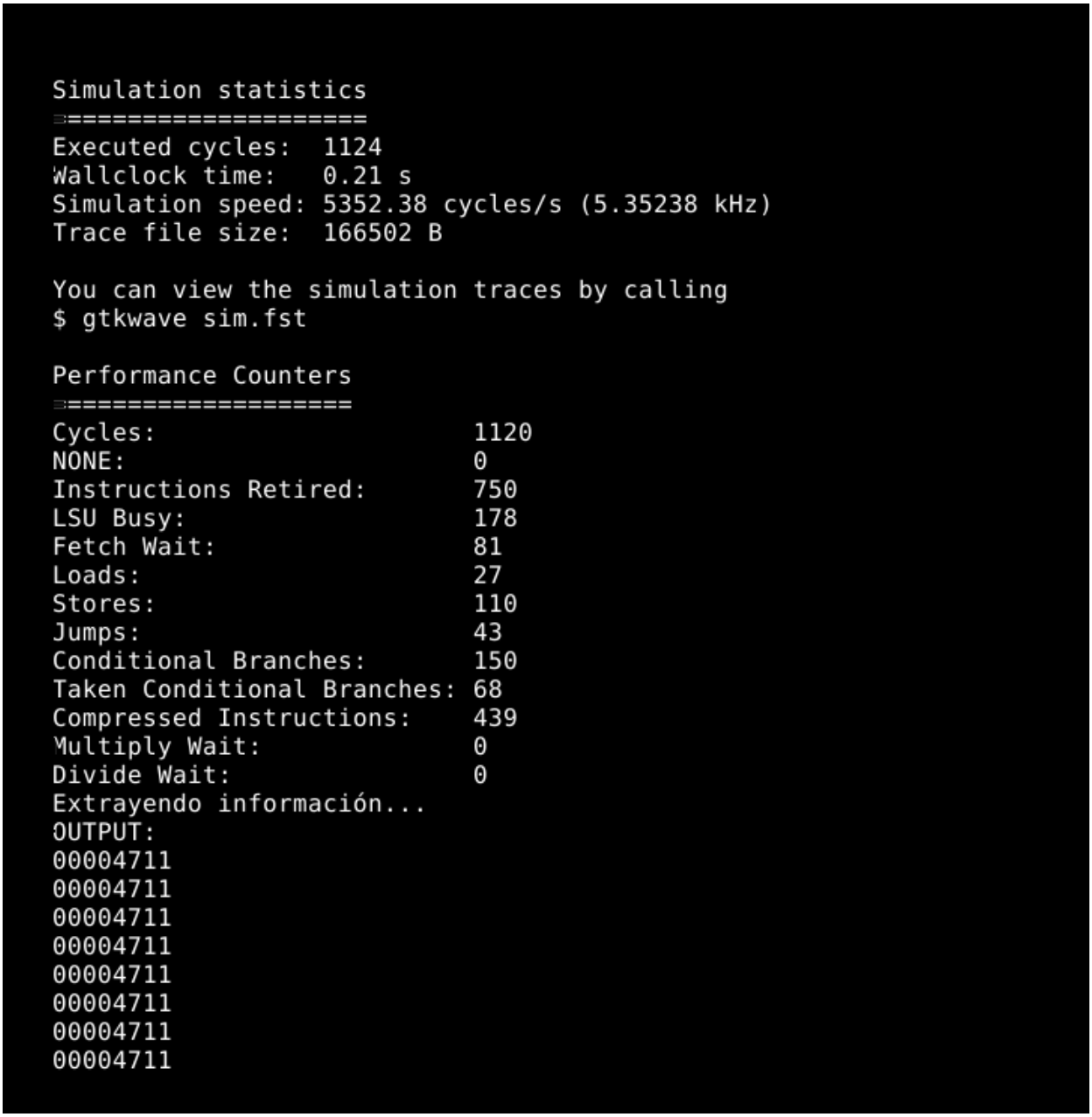

Figura A.4: Ejecución de prueba para acelerador SIMD

Bibliografía

[1] CHIPS Alliance. *Rocket Chip*. URL: https://github.com/chipsalliance/rocket-chip.

[2] David Patterson y Andrew Waterman. *Guia Practica de RISC-V*. Strawberry Canyon, 2018.

[3] Krste Asanović y David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Inf. téc. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[4] Krste Asanović y col. *The Rocket Chip Generator*. Inf. téc. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[5] Jeff Bezanson y col. "Julia: A fast dynamic language for technical computing". En: *arXiv preprint arXiv:1209.5145* (2012).

[6] Dileep Bhandarkar y Douglas W. Clark. "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization". En: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, 310–319. ISBN: 0897913809. DOI: 10.1145/106972.107003. URL: https://doi.org/10.1145/106972.107003.

[7] Greg Novick Crystal Chen y Kirk Shimano. *RISC Arquitecture*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[8] *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[9] Peter N. Glaskowsky. "NVIDIA ' s Fermi : The First Complete GPU Computing Architecture". En: 2009.

[10] *Grupo ArTeCS*. URL: https://artecs.dacya.ucm.es/.

[11] John Gustafson. "A radical approach to computation with real numbers". En: *Supercomputing Frontiers and Innovations* 3 (ene. de 2016), págs. 38-53. DOI: 10.14529/jsfi160203.

[12] John Gustafson. *The End of Error: Unum Computing*. Feb. de 2015. ISBN: 1482239868.

[13] John Gustafson e I. Yonemoto. "Beating floating point at its own game: Posit arithmetic". En: *Supercomputing Frontiers and Innovations* 4 (ene. de 2017), págs. 71-86. DOI: 10.14529/jsfi170206.

[14] David Harris y Sarah Harris. *Digital Design and Computer Architecture, Second Edition*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 0123944244.

[15] M. K. Jaiswal y H. K. . So. "PACoGen: A Hardware Posit Arithmetic Core Generator". En: *IEEE Access* 7 (2019), págs. 74586-74601.

[16] Manish Jaiswal y Hayden So. "Architecture Generator for Type-3 Unum Posit Adder/Subtractor". En: mayo de 2018, págs. 1-5. DOI: 10.1109/ISCAS.2018.8351142.

[17] Manish Jaiswal y Hayden So. "Universal number posit arithmetic generator on FPGA". En: mar. de 2018, págs. 1159-1162. DOI: 10.23919/DATE.2018.8342187.

[18] Earl Killian. "MIPS cache simulation results". En: *Personal communication* (1990).

[19] Cerlane Leong. *SoftPosit C Library*. URL: https://gitlab.com/cerlane/SoftPosit.

[20] lowRISC. *Ibex Core*. URL: https://github.com/lowRISC/ibex.

[21] *lowRISC*. URL: https://www.lowrisc.org/.

[22] Raul Murillo, Alberto A. [Del Barrio] y Guillermo Botella. "Deep PeNSieve: A deep learning framework based on the posit number system". En: *Digital Signal Processing* 102 (2020), pág. 102762. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2020.102762. URL: http://www.sciencedirect.com/science/article/pii/S105120042030107X.

[23] Raul Murillo, Alberto A. Del Barrio y Guillermo Botella. "Customized Posit Adders and Multipliers using the FloPoCo Core Generator". En: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. to appear. 2020.

[24] David A. Patterson y John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269.

[25] *RISC-V History*. URL: https://riscv.org/risc-v-history/.