GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Extensión Zve32x para RISC-V

Autor: Moisés Martínez Hernantes

Director: Fermín Zabalegui Sanz

Co-Director: José Daniel Muñoz Frías

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Extensión Zve32x para RISC-V

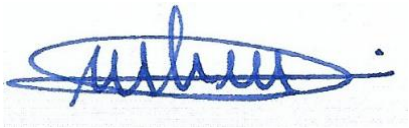
en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2021/22 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.



Fdo.: Moisés Martínez Hernantes Fecha:20/07/2022

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Fermín Zabalegui Sanz Fecha: 20/07/2022

Fdo.: José Daniel Muñoz Frías Fecha: 20/07/2022

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Extensión Zve32x para RISC-V

Autor: Moisés Martínez Hernantes

Director: Fermín Zabalegui Sanz

Co-Director: José Daniel Muñoz Frías

Madrid

Agradecimientos

A mi familia por todo su apoyo.

EXTENSIÓN ZVE32X PARA RISC-V

Autor: Martínez Hernantes, Moisés.
Director: Zabalegui Sanz, Fermín.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

1. Introducción

Mediante este Trabajo Fin de Grado se estudiará, diseñará, simulará e implementará la extensión de procesamiento vectorial, Zve32x para la arquitectura RISC-V. Para ello, se partirá de una CPU ya creada, NeoRV32, y se utilizará el lenguaje VHDL para su descripción.

Para la consecución de que el núcleo sea capaz de realizar instrucciones vectoriales en un primer momento se estudiará el NoeRV32, con el fin de poder comprender su funcionamiento y las señales de comunicación que usa. Para ello se realizarán una serie de testbench de distintos componentes similares a los que se van a implementar, tales como la ALU.

Seguidamente se estudiará el funcionamiento de la extensión, informándose por distintos medios, como videos de expertos o informes descriptivos. Una vez se tiene una idea clara de su funcionamiento se comienza con el diseño de los circuitos digitales en VHDL.

2. Definición del Proyecto

En este Proyecto se desarrollará la extensión de procesamiento vectorial Zve32x, que permitirá al núcleo NeoRV32 realizar instrucciones vectoriales. La principal diferencia que aporta esta extensión es que permite trabajar al núcleo con vectores de una manera diferente y que aporta un mayor rendimiento.

De esta manera se podrá trabajar con vectores y realizar operaciones aritméticas en un menor tiempo, ya que el núcleo será capaz de procesar todos los elementos del vector al mismo tiempo.

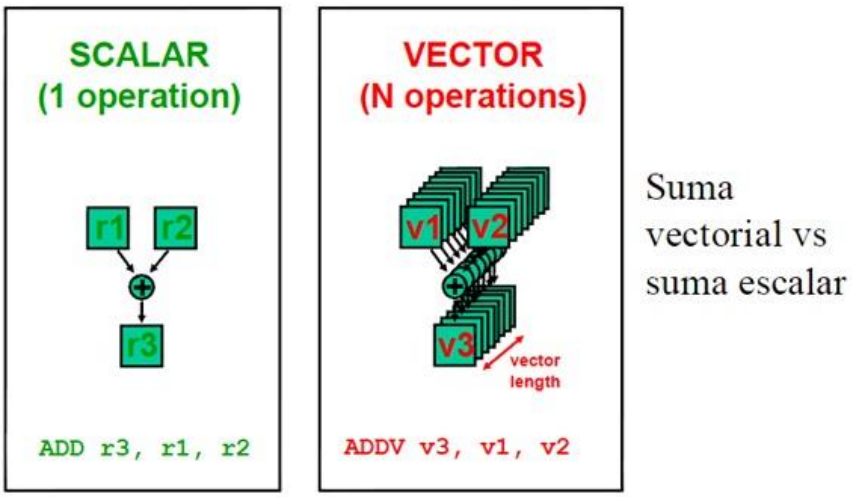


Figura 1. Imagen comparativa entre la suma escalar y la suma vectorial. [1]

3. Descripción del modelo/sistema/herramienta

Para lograr que el núcleo sea capaz de trabajar con todos los elementos del vector será necesario implementar una serie de componentes nuevos o actualizar algunos de los existentes.

De esta manera lo primero en implementarse serán los nuevos CSR(Configuration and Status Registers), teniendo en cuenta que unos son de lectura y escritura y otros de solo lectura.

Seguidamente se implementarán las instrucciones vsetvl, vsetvli y vsetivli, con las cuales se conseguirá modificar los datos contenidos en los CSR vtype y vl, ya que estos no se escriben como un CSR normal y además tienen una serie de condiciones que cumplir.

Se tendrá que crear un banco de registros completamente nuevo en el que se almacenen todos los vectores con los que se trabajará. Concretamente se crearán 32 nuevos registros de una longitud fija denominada VLEN.

Para poder cargar datos en este registro se implementará la instrucción vlex.v, con la que se podrá leer datos desde la memoria RAM y almacenarlos en el banco de registros de vectores para su posterior procesamiento.

Por último, se creará una nueva ALU que sea capaz de realizar las operaciones aritméticas vectoriales. Para indicar qué tipo de operación se realiza se implementará la instrucción, que en el caso de la suma sería vadd.vv. Con esta instrucción se podrá cargar, sumar y guardar todos los elementos de un vector al mismo tiempo, sin tener que cargar los dos elementos a sumar del registro, sumarlos, volver a guardarlos y así con todos los elementos.

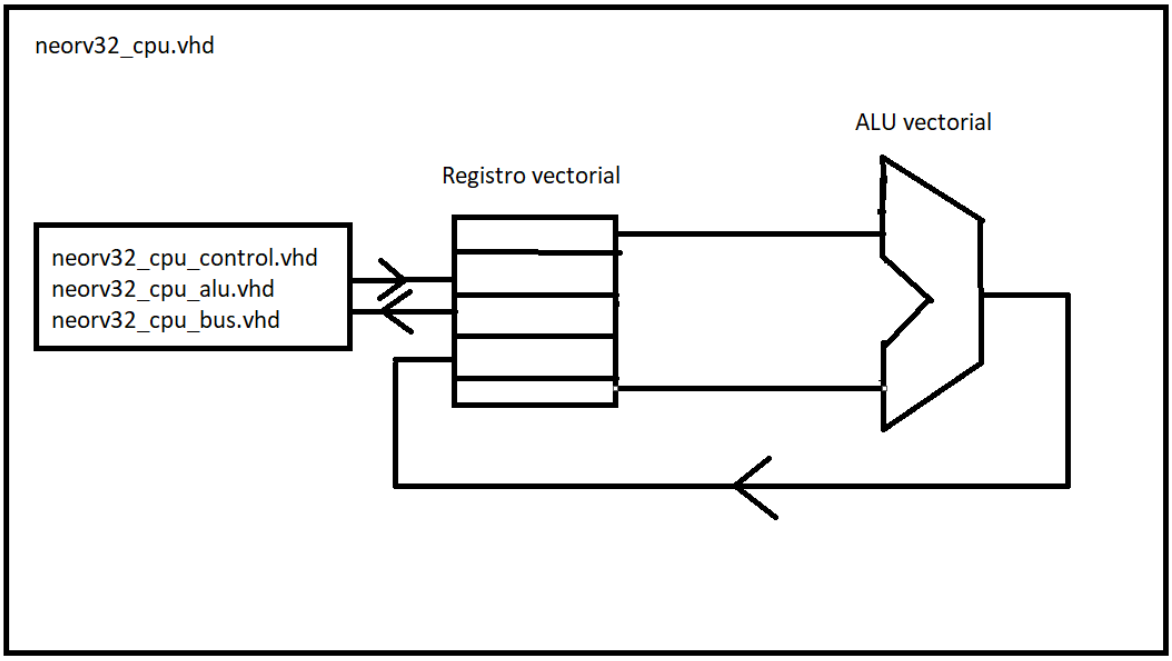


Figura 2. Diagrama de bloque de los circuitos digitales diseñados.

4. Resultados

Al termino de este proyecto se consiguió implementar todo lo mencionado anteriormente, y para comprobar que todo funcionaba correctamente se realizaron pruebas en cada uno de los componentes nuevos que se iban creando. Para mostrar el funcionamiento de todo lo implementado se adjuntan la Figura 3 en la que se puede observar cómo se cargan todos los CSR, incluidos vtype y vl con su instrucción correspondiente y la Figura 4 en la que se ve como se carga datos desde la memoria en los dos registros vectoriales y como se realiza la suma y la resta de los mismos almacenándose el resultado en otro registro vectorial, ejecutándose de este modo vle8.v, vadd.vv y vsub.vv.

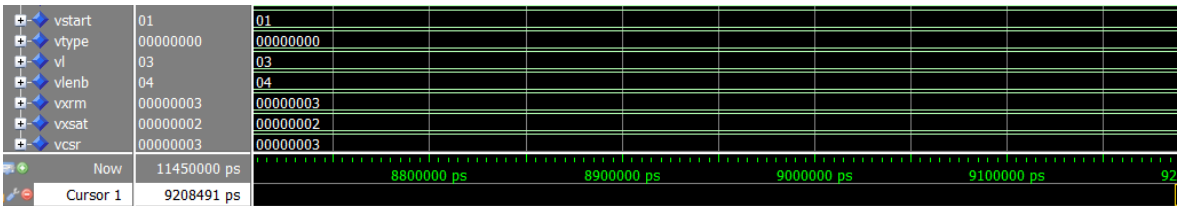


Figura 3. Simulación en la que se cargan valores en todos los CSR.

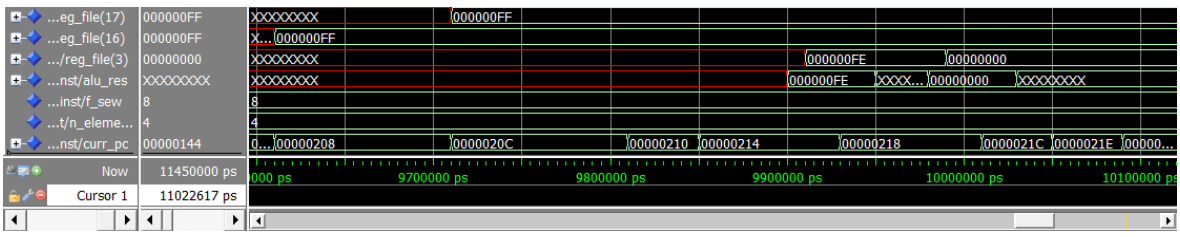


Figura 4. Simulación en la que se cargan datos vectoriales y se procesan en la ALU vectorial.

5. Conclusiones

Con este Trabajo Fin de Grado se ha conseguido completar todos los objetivos marcados, consiguiendo implementar una nueva funcionalidad para el núcleo NeoRV32. Si bien es cierto que se le pueden ampliar las instrucciones para que tenga otras maneras de cargar datos o realizar operaciones que no sean solo aritméticas, sí que es verdad que se ha establecido unas bases muy sólidas desde las que se puede empezar, ya que la mayoría del trabajo de estudio del funcionamiento está realizado y la mayoría de las señales necesarias del núcleo ya están conectadas a la extensión.

6. Referencias

[1] «Electropediadigital,» 22 abril 2013. [En línea]. Available: http://electropediadigital.blogspot.com/2013/04/risc-vs-procesadores-vectoriales.html. [Último acceso: 2022 junio 19].

EXTENSION ZVE32X FOR RISC-V

Author: Martínez Hernantes, Moisés.
Supervisor: Zabalegui Sanz, Fermín.
Collaborating Entity: ICAI – Universidad Pontificia Comillas.

ABSTRACT

1. Introduction

With this Bachelor Thesis will be studied, designed, simulated and implemented the extension of vector processing, the extension Zve32x, for the architecture RISC-V. For it, will split from a CPU already created, the NeorRV32 CPU, and the language VHDL will be used to describe the digital circuits.

For the achievement of the core will be capable of processing vector instructions, in the first instance the NeoRV32 will be studied, with the final purpose of being able to understand how it works and the communication signals that use. For it a number of testbench will be done in different components similar to the ones that will be implemented, such as the ALU.

Next the performance of the extension will be studied, getting informed by different means, such us experts' videos or papers. Once you have a clear idea of the functioning the design of the digital circuits will be start.

2. Definition of the project

In this project the vector processing extension Zve32x will be developed, allowing the core NeoRV32 able to process vector instructions. The main difference that contributes this extension es that allows the core to work with vector in a different way and with a better performance.

In this way it will be possible working with vectors and perform arithmetic operations in less time, because the core is able to access to all the vector elements at the same time.

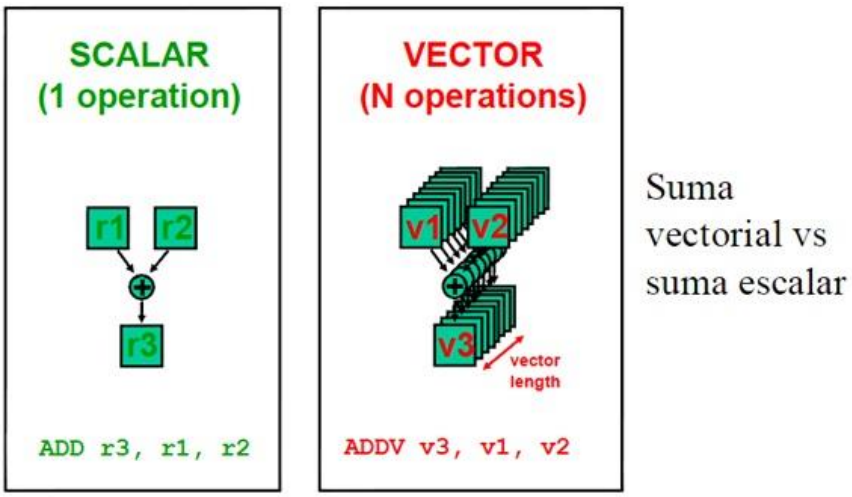


Figura 5. Comparative image between scalar add and vector add. [1]

3. Model description

A To achieve that the core is able to work with all the vector elements at the same time it will be necessary to implement a series of new components or update some of the existing ones.

In this way the first thing to implement is the new CSR, taking into account that some of them have writing and reading access and other only reading access.

Next the instructions vsetvl, vsetvli, vsetivli will be implemented, with it the data contained in the CSR vtype and vl will be modified, because this CSR don`t be write like a normal CSR and moreover have some additional conditions to comply.

A completely new vector register must be created, where all the vectors that will be used will be stored. Specifically, 32 new registers will be created with a fixed length call VLEN.

To be able to load data in these registers a new instruction call vlex.v will be implemented, with which data from the RAM memory will be read and load in the vector register in order to be process.

Finally, a new ALU capable of doing arithmetic vector operations will be created. For indicate which type of operation will be done the instruction vxxx.v will be implement, for example, in the add case the instruction will be vadd.v. With this instruction all the vector elements could be load, add and store at the same time, without loading two vector elements to add, add them, store them again in the vector register and so with all the vector elements.

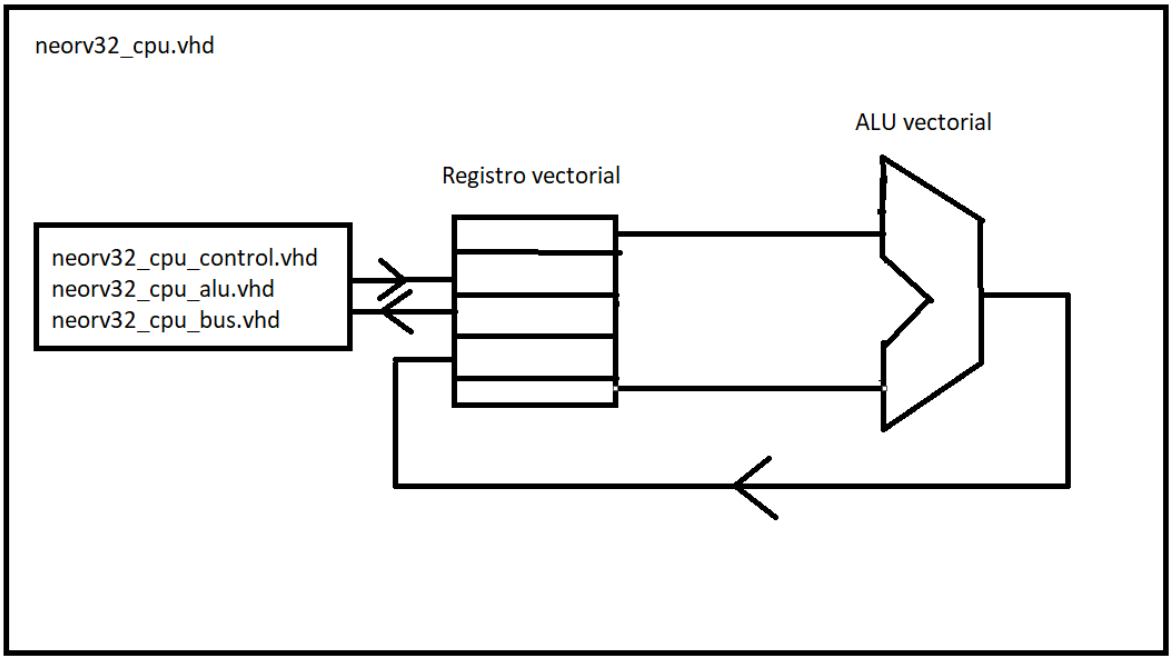


Figura 6. Block diagram of the designed digital circuits.

4. Results

At the end of this project, it was achieved all of the above, and in order to check everything was working properly, many test was realized for each of the new components that were being created. To show functioning of everything thar was implemented are attached the Figura 7 in which you can see how all CSR are loaded, including vtype and vl with the corresponding instruction and the Figura 8 in which it is seen how vector registers are loaded from the RAM memory and how the add and the sub of those vector registers are done, storing it in other vector register.

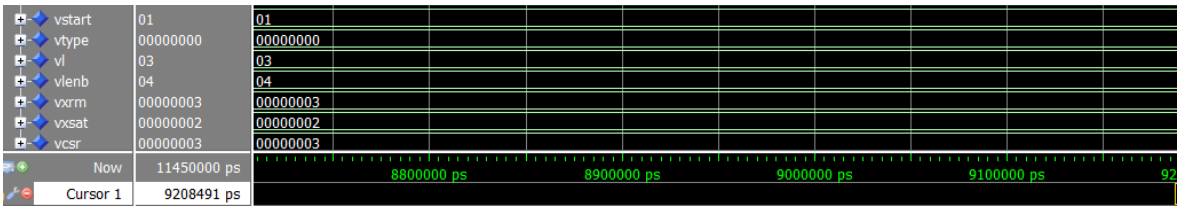


Figura 7. Simulation in which all the CSR are loaded with a value.

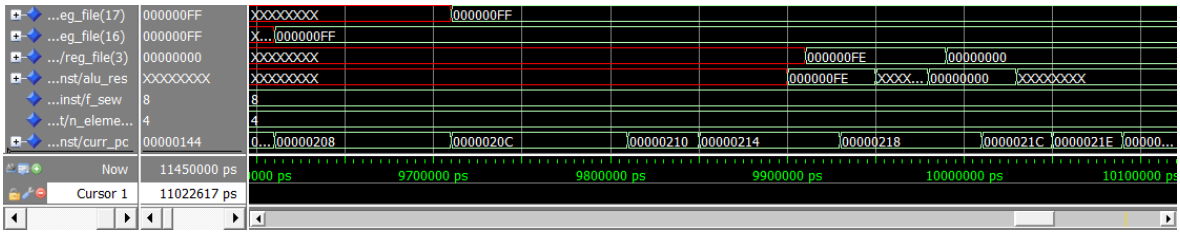


Figura 8. Simulation in which data vector is loaded and is process in the vector ALU.

5. Conclusions

With this Bachelor Thesis all the objectives have been completed, achieving the implementation of a new functionality for the NeoRV32. While it is true that some instructions can be added in order to have other ways to load data or doing different operations, not only arithmetic, is true that solid foundation have been laid from where other project can start, because most of the functioning study of the core have been done and the most of the needed signals of the core have been already connected to the extension.

7. References

[1] «Electropediadigital,» 22 abril 2013. [En línea]. Available: http://electropediadigital.blogspot.com/2013/04/risc-vs-procesadores-vectoriales.html. [Último acceso: 2022 junio 19].

Índice de la memoria

Índice de figuras

Índice de tablas

Índice de códigos

Capítulo 1. INTRODUCCIÓN

1.1 INTRODUCCIÓN DEL PROYECTO

En este proyecto se desarrollará la extensión Zve32x para el núcleo NeoRV32, basado en la arquitectura RISC-V, que consistirá en dar la posibilidad de que el núcleo sea capaz de realizar operaciones vectoriales.

Es un proyecto que se basa completamente en el uso de hardware libre, es decir que no se tiene que pagar nada por el uso de componentes que han sido desarrollados por otros desarrolladores. De esta manera se partirá del núcleo NeoRV32, que será al que se le implementará la extensión.

Lo primero será familiarizarse con el funcionamiento de los microprocesadores, a nivel de diseño de hardware, y en concreto con el NeoRV32, ya que se usarán las señales internas de comunicación del núcleo para la extensión. Para ello se seguirán los siguientes pasos:

1. Se genera un proyecto en Quartus II y se sigue la guía del núcleo para conseguir configurarlo y que nos compile.
2. Una vez ya configurado se comienza con el estudio de los archivos que contiene el núcleo, con el objetivo de comprender la jerarquía y sus señales internas, para encontrar donde implementar la extensión.
3. Para confirmar que se tiene un buen conocimiento de núcleo se realizara un testbench de una pequeña parte de núcleo, la extensión M que se encarga de hacer la multiplicación y la división de números enteros. El siguiente paso será subir en el nivel de jerarquía, que seria la ALU.

El siguiente paso será diseñar los circuitos digitales para posteriormente describirlos en VHDL. También se realizarán a estos nuevos componentes simulaciones mediante el uso de testbench, para comprobar que cumplen con su cometido.

Una vez desarrollados estos componentes nuevos se realizarán programas en ensamblador con el fin de poder comprobar su buen funcionamiento. Para ello también se utilizará una toolchain que permita compilar los programas, y en ocasiones tocará ensamblar a mano alguna de las instrucciones.

Por último, se conectarán todos los componentes debidamente con el núcleo para que pase a ser una unidad funcional del mismo.

1.2 MOTIVACIÓN DEL PROYECTO

Todo comienza con la simple curiosidad de conocer como es el diseño de un microprocesador, por lo que se empieza por buscar información acerca, descubriendo que existe una arquitectura de código abierto, RISC-V, dando la posibilidad de desarrollar parte de ella. De esta manera nace la idea de crear una nueva extensión para un núcleo, el NeoRV32, basado en dicha arquitectura.

Tras estudiar varias opciones, se llega a la conclusión de desarrollar la extensión de procesamiento vectorial para números de 32 bits en coma fija, la extensión Zve32x, con la que se conseguiría que ciertos algoritmos que requieren mucha potencia de cálculo aumentasen su rendimiento, como podría ser el caso de los algoritmos de inteligencia artificial.

Principalmente se escogió dicha extensión por el hecho de que el procesamiento vectorial no es muy conocido y aplica métodos que permiten, en algunos casos, al procesador realizar operaciones con vectores enteros en la duración de un ciclo de reloj, permitiendo un mayor rendimiento que si se aplicasen procesos escalares.

Una gran motivación es que nadie ha diseñado esta extensión con anterioridad para este núcleo, y tras una búsqueda exhaustiva tan solo se ha encontrado otro estudio que la implementase, aunque diseñada para otro tipo de núcleo y que no está escrita en VHDL. Esto hace que sea la primera persona en diseñarla, lo que es gratificante al mismo tiempo

que desafiante, ya que permite la posibilidad de que cualquier persona pueda usar mi proyecto para usarlo o continuarlo.

Finalmente, la principal razón por la que se desarrolla este proyecto es para llenar la necesidad de conocimiento sobre el diseño de los microprocesadores, ya que gracias a este trabajo se ha podido comprender la gran mayoría de sus componentes, de cómo se comunican entre sí, de las etapas que se siguen, desde la idea básica a implementar el resultado. En resumidas cuentas, se ha cumplido con las expectativas con las que se empezó la búsqueda del trabajo fin de grado.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

Durante el desarrollo del proyecto se va a utilizar distintos tipos de software, lenguajes de descripción de hardware y arquitecturas, de las que se puede tener conocimiento, pero para asegurar el perfecto entendimiento de los mismos, se describirán a continuación.

2.1 VHDL

El lenguaje VHDL, tal y como lo define el IEEE (Institute of Electrical and Electronic Engineers), "*es un lenguaje utilizado para describir circuitos digitales y para la automatización de diseño electrónico, a estos lenguajes se les suele llamar lenguajes de descripción de hardware"* [2]. Se utilizará para describir cada uno de los componentes de la nueva extensión a crear, todo desde un entorno muy parecido al de los compiladores de programación en C, python, etc. Si no se conoce mucho acerca del tema del diseño de circuitos digitales, puede surgir la duda de cómo funciona este tipo de lenguajes. Para esclarecerlo se retrocederá hasta los comienzos del diseño de microprocesadores cuando no se contaba con esta tecnología, y el método que se usaba consistía en dibujar sobre una lámina de grandes dimensiones el hardware y posteriormente usar el Breadboarding, que simplemente consistía en unir mediante cables todos los circuitos electrónicos que se habían creado, en la Figura 9 aparece un microprocesador creado mediante esta filosofía. Es ahí donde este tipo de lenguajes aparecen, permitiendo realizar y unir todos los circuitos electrónicos de una manera más sencilla y rápida, mediante del uso de código.

Figura 9. Ejemplo de Breadboarding. [3]

2.2 RISC-V

"Es una arquitectura de conjunto de instrucciones (ISA) de hardware libre basado en un diseño de tipo RISC (conjunto de instrucciones reducido)" [4]. Es el tipo de arquitectura en la que se basará la extensión y que se implementará en un núcleo ya creado por otros desarrolladores, que al ser de hardware libre se puede usar siempre que no se use con fines comerciales entre otras condiciones. Es una arquitectura tan válida como las que usan empresas como Intel o AMD, con la diferencia de que estas empresas cobrarían tarifas de licencia por el uso de sus patentes.

Figura 10. Logo de RISC-V international. [5]

2.3 NEORV32

Es el núcleo del que se partirá para crear la extensión, que es de código libre y basado en la arquitectura RISC-V. Es un diseño de procesador ya creado por otros desarrolladores, por lo que la finalidad de este trabajo es añadir una nueva extensión que no posee e implementarla. Es un núcleo que está muy bien documentado, facilitando la comprensión de su funcionamiento y extremadamente configurable, lo que le hace perfecto para este TFG.

2.4 MODELSIM

Es la herramienta que se utilizará para poder simular todos los circuitos que se diseñen con el fin de comprobar su buen funcionamiento. La simulación es un proceso de gran importancia, ya que nos permite ver todas las señales y su comportamiento en función de las entradas. Es necesario crear un archivo VHDL denominado testbench con el que se elegirán tanto los componentes como los valores de las entradas, por lo que para tener una buena simulación es necesario crear un testbench que pruebe todos los escenarios posibles. En la Figura 11 se puede ver un ejemplo de salida que nos proporciona este programa.

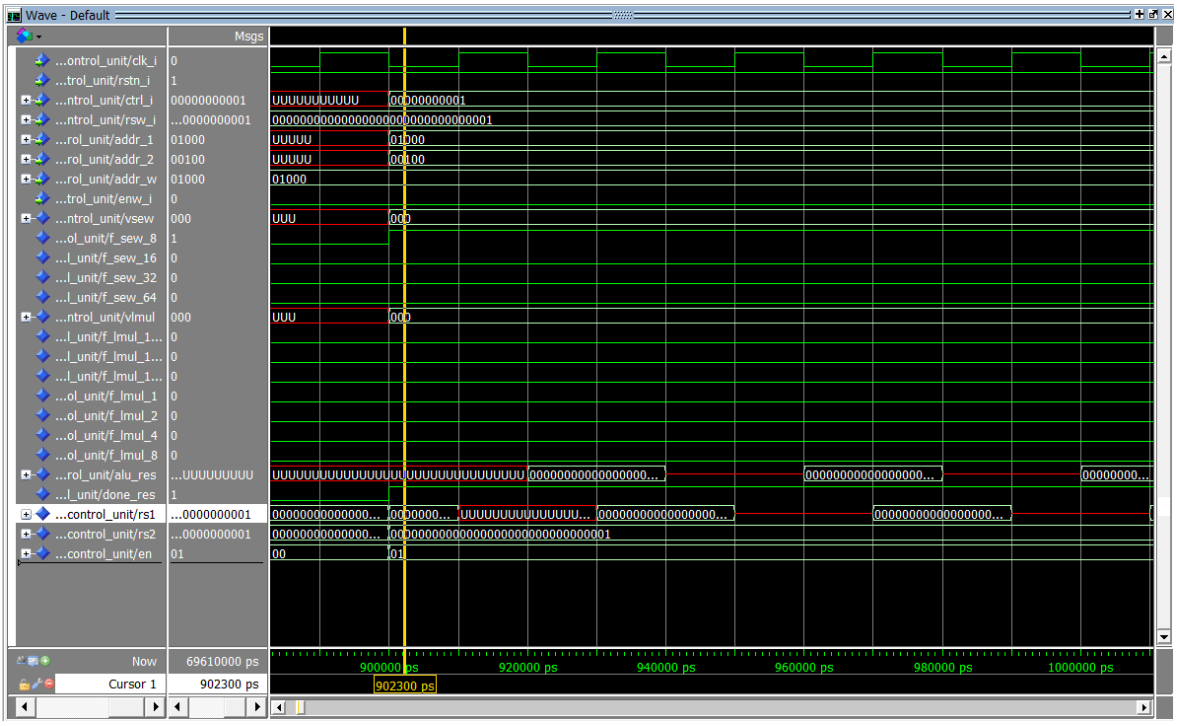


Figura 11. Salida de un testbench de un componente de la extensión creada.

2.5 QUARTUS II

Es el programa que se utilizará para sintetizar todos los circuitos digitales, además también se utilizará para describir los mismos.

2.6 CSR

Sus siglas en inglés son Control and Status Registers, de las que se deduce que son registros para el control e informan sobre algún estado. En otras palabras, "*estos registros permiten desde conocer ciertos aspectos de la ejecución del procesador, como el número de ciclos de reloj; como controlar ciertos aspectos de su funcionamiento, como la habilitación de interrupciones"* [6]. Estos tipos de registros pueden tener tres niveles de prioridad, usuario (U), supervisor (S) y maquina (M), en nuestro sistema al ser empotrado bastará con el nivel máquina, que es el más prioritario de todos. En cuanto al modo se tienen dos, de lectura y escritura (RW) y de solo lectura (RO), ambos se usarán.

Dirección	Modo	Nombre	Descripción
0x003	RW	`fcsr`	Registro de estado y control de coma flotante
0xC00	RO	`cycle`	Contador de ciclos
0xC01	RO	`time`	Hora
0xC02	RO	`instret`	Instrucciones retiradas
0xC03	RO	`hpmcounter3`	*Hardware Performance-monitoring counter 3*
0xC04	RO	`hpmcounter4`	*Hardware Performance-monitoring counter 4*
		...	
0xC1F	RO	`hpmcounter31`	*Hardware Performance-monitoring counter 31*
0xC80	RO	`cycleh`	Parte alta de `cycle` (solo RV32)
0xC81	RO	`timeh`	Parte alta de `time` (solo RV32)
0xC82	RO	`instreth`	Parte alta de `instret` (solo RV32)
0xC83	RO	`hpmcounter3h`	Parte alta de `hpmcounter3` (solo RV32)
0xC84	RO	`hpmcounter4h`	Parte alta de `hpmcounter4` (solo RV32)
		...	
0xC9F	RO	`hpmcounter31h`	Parte alta de `hpmcounter31` (solo RV32)

Figura 12. Algunos de los CSR accesibles en modo usuario. [7]

Capítulo 3. ESTADO DE LA CUESTIÓN

La idea de realizar este trabajo comienza con el deseo y la curiosidad de un estudiante en conocer cómo se diseñan los microprocesadores, por ello tras investigar sobre el tema se encuentra que existe un tipo de arquitectura que actualmente es usada y que es de hardware libre, RISC-V, por lo que se podría partir de otros proyectos e implementar nuevas funcionalidades. Junto a mis tutores del TFG se llegó a la conclusión de que una buena idea sería implementar una nueva extensión a un núcleo que ya había sido desarrollado por otros desarrolladores. De esta manera se decide que se utilizará el núcleo NeoRV32, ya que está muy bien documentado, facilitando el aprendizaje de su funcionamiento. Se decidió que la extensión que se añadiría seria la Zve32x, la cual permite hacer cálculos vectoriales con números de 32 bits en coma fija.

Una de las principales razones por las que se implementa esta extensión es porque hasta la fecha de escritura no se ha encontrado ningún trabajo de código abierto que describa dicha extensión en VHDL, por lo que es algo innovador. En el mercado existen determinadas empresas que ofrecen microprocesadores con este tipo de extensiones implementadas, pero son empresas como Intel o AMD, que siempre te van a vender el producto, pero no te van a dar el código, o en caso de que lo hagan te cobrarían derechos de autor por su uso.

Este tipo de extensiones permiten agilizar determinados algoritmos, como son por ejemplo los de la inteligencia artificial, pero también son muy usadas en los superordenadores del área de la computación científica. Otro ámbito en el que son muy usadas es el de las tarjetas gráficas o GPU y las videoconsolas.

A la vista de los expuesto, se entiende que la extensión que se ha diseñado tiene un amplio mercado, que se ve copado por empresas que cobran por sus servicios, mientras que esta extensión es de código libre y puede ser usada por cualquier persona sin ningún tipo de problema burocrático ni regalías.

3.1 EXTENSIÓN MMX DE INTEL

La extensión MMX es la extensión de procesamiento vectorial diseñada por Intel. Esta fue introducida en sus procesadores en el año 1997, en la gama de procesadores Pentium, y desde entonces la han soportado la gran mayoría de las compañías fabricantes de microprocesadores basados en la arquitectura x86.

El principal objetivo con el que se creó fue el de aumentar el rendimiento en tareas multimedia, tales como, el procesado de imágenes o la compresión y descompresión de audio y video.

3.2 EXTENSIÓN 3D NOW!

La extensión 3D now! nació de la mano de AMD con el objetivo de aumentar el rendimiento de los procesos multimedia de sus procesadores. De esta manera se tiene una nueva extensión vectorial, que permite al microprocesador realizar operaciones vectoriales. El microprocesador elegido para estrenar la extensión fue el AMD K6-2 en 1998. Claramente nació para competir con la anteriormente menciona extensión MMX de Intel, por lo que en ella se implementaban mejoras, entre las que destaca el poder utilizar números en coma flotante y no solo enteros.

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

El desarrollo de este proyecto se ve justificado con la necesidad de crear una extensión de código abierto que permita el procesamiento vectorial de datos, ya que hay una disponibilidad muy baja y permite optimizar varios procesos, como los algoritmos de inteligencia artificial, haciendo que no se necesite un hardware muy potente para la realización de dichos procesos. La justificación se basará en los siguientes puntos:

4.1.1 PROCESAMIENTO VECTORIAL

Lo primero es entender en que se diferencian los procesadores vectoriales de los escalares. En la actualidad domina el uso de procesadores escalares, con los que se ha conseguido unas frecuencias de trabajo tan altas que nos permiten realizar procesos en una fracción de tiempo infinitesimal, sin embargo, que sean muy rápidos no los hace perfectos para todos los procesos. Este tipo de procesadores solo son capaces de manejar un dato en cada ciclo de reloj, aunque su frecuencia de trabajo es muy alta, por lo que a cada segundo son capaces de procesar muchos datos, a pesar de que sea de uno en uno.

La gran mayoría de los dispositivos electrónicos que cuentan con un microprocesador actualmente son escalares, ya que son muy eficientes y rápidos para la mayoría de los procesos de uso cotidiano.

En contraposición se tiene los procesadores vectoriales, que son un concepto totalmente distinto, ya que se basan en actuar sobre múltiples datos en un ciclo de reloj. Actualmente su uso está restringido básicamente a la supercomputación, ámbito en el que todavía se mantienen como la mejor opción, ya que realizan de manera mucho más eficiente los diferentes cálculos matemáticos. Es decir, este tipo de procesadores son mejores que los escalares en la realización de unas pocas instrucciones.

Es por ello que actualmente la mejor opción es tener un procesador escalar, pero que este cuente con componentes que puedan realizar procesamiento vectorial. En la Figura 13 se puede ver un modelo muy simplificado de la estructura descrita. De esta manera actualmente se encuentran procesadores que incluyen dichos componentes los cuales son capaces de trabajar con instrucciones vectoriales. Estas instrucciones se conocen como instrucciones SIMD (Single Instruction, Multiple Data), que por sus siglas en ingles se deduce su funcionamiento, que consiste que mediante una sola instrucción se pueda acceder a múltiples datos. En la Figura 14 se puede observar más claramente cómo funcionan las instrucciones SIMD.

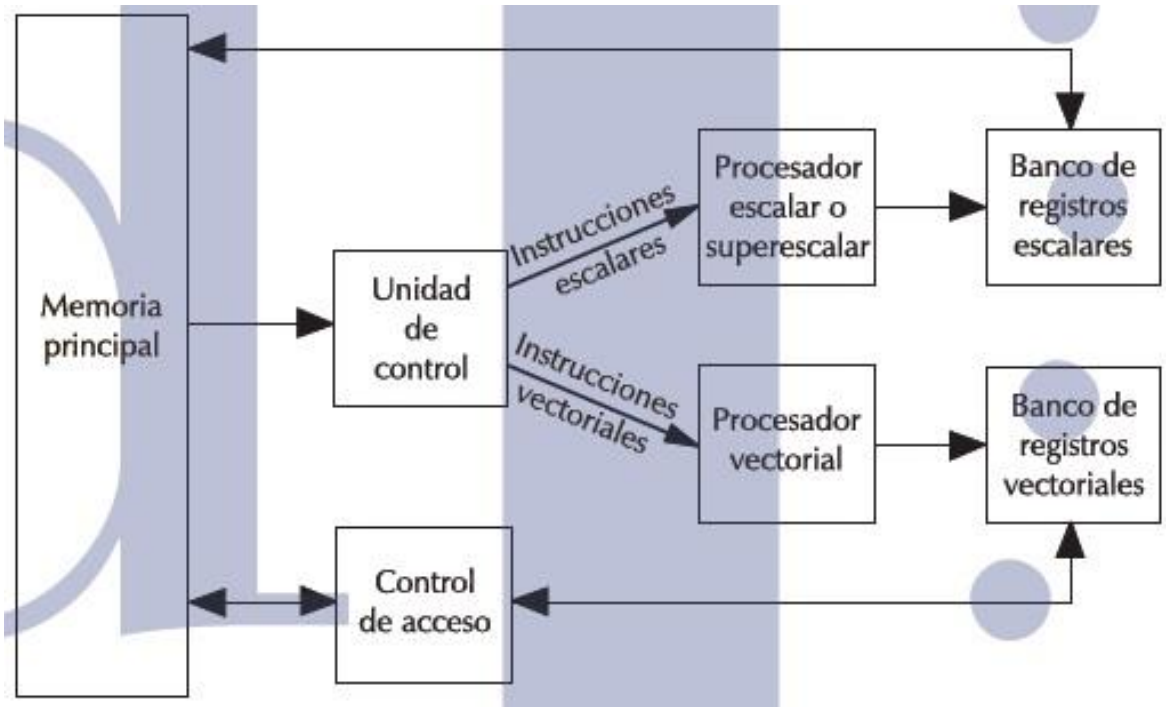


Figura 13. Estructura simplificada de un procesador con funciones escalares y vectoriales [8].

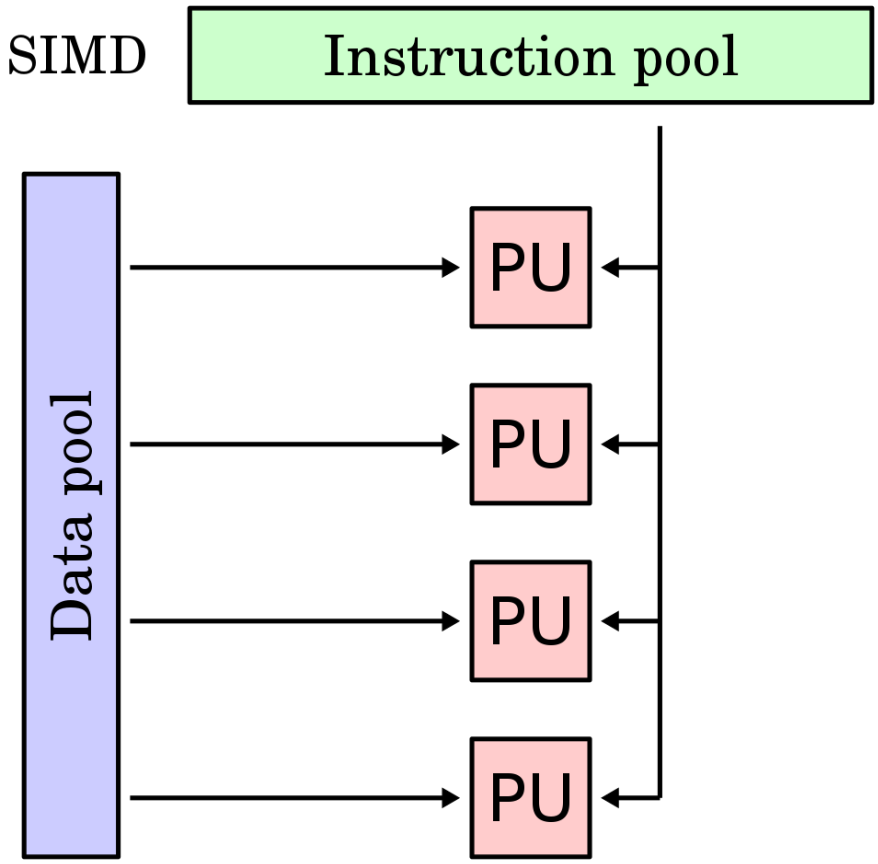


Figura 14. Esquema del funcionamiento de una instrucción SIMD. [9]

4.1.2 ARQUITECTURA RISC-V

Todo el núcleo está diseñado siguiendo la arquitectura de conjunto de instrucciones RISC-V, por lo que la extensión también tendrá esta arquitectura. RISC-V tiene una característica muy importante, y es que RISC-V es una arquitectura de hardware libre, por lo que puede usarse por cualquier persona sin tener que pagar por su uso. Si bien es cierto que no es la única arquitectura libre, tiene una gran ventaja sobre el resto de sus competidores, ya que fue diseñada para poder ser usada en múltiples y diversos chips, ofreciendo una gran compatibilidad con muchos y variados dispositivos electrónicos.

Otra gran ventaja que nos ofrece que sea de hardware libre reside en que no se tiene burocracia, ya que, si se usase otra arquitectura patentada, como la ARM muy utiliza en dispositivos móviles y tablets, cuando se quisiese publicar las características y ventajas de nuestro proyecto se debería tener en cuenta que no se podría publicar todo ya que normalmente se firman acuerdos de no divulgación.

Por último, RISC-V se está teniendo muy en cuenta para ser la arquitectura de los próximos superordenadores europeos, ya que hasta la fecha estos dependen de arquitecturas como la x86-64 de Intel o ARM de AMD, por lo que sería una manera de independizarse de estas grandes compañías, y por lo tanto de la hegemonía que ostenta Estados Unidos. Aunque es un cambio muy grande, es muy posible que termine aconteciendo, como ya paso con los sistemas operativos de los superordenadores, que usaban Unix de código cerrado, y desde 2018 el 100% de los superordenadores más potentes del mundo usan Linux, de código abierto [10]. En la Figura 15 se puede observar los 10 superordenadores más potentes y el sistema operativo que usan, todas variaciones de Linux.

A la vista de lo expuesto se puede sacar una clara conclusión, RISC-V es una arquitectura con mucho futuro y muy prometedora, por lo que su uso y entendimiento es la mejor opción para este proyecto.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
7	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
8	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
9	**SuperMUC-NG** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
10	**Lassen** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100 , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	

Figura 15. Imagen de las características de los 10 superordenadores más potentes del mundo. [10]

4.1.3 INTELIGENCIA ARTIFICIAL (IA)

Una de las principales funciones que puede desarrollar la extensión es acelerar algoritmos de inteligencia artificial. Esto quiere decir que con la misma potencia de hardware se obtendrá un rendimiento mayor. Es bien conocido que últimamente la inteligencia artificial ha irrumpido muy fuertemente en nuestra sociedad, para tener una idea más clara hasta que nivel ha llegado cabe decir que numerosas organizaciones internacionales han creado una normativa con el fin de poder controlar como se usa y emplea. Dentro de la inteligencia artificial, aunque no es unánime, se distinguen dos tipos principales:

- Centrados en humanos
    - Sistemas que piensan como los humanos
    - Sistemas que actúan como los humanos
- Centrados a la racionalidad
    - Sistemas que piensan racionalmente
    - Sistemas que actúan racionalmente

Ahora que ya se tiene un contexto sobre la inteligencia artificial, se tiene que ver porque realmente es tan importante, y por lo tanto cualquier proceso que mejore su funcionamiento, como la extensión desarrollada, tiene un mercado muy amplio y lucrativo. Entre los múltiples campos en los que se aplica, se destacaran dos:

- Uso en coches autónomos. Actualmente la industria automovilística apuesta muy fuerte por esta tecnología, que usa una de las ramas más complicadas de la inteligencia artificial, el Deep Learning.
- Predicciones en base a conductas. Mediante el uso de inteligencia artificial actualmente se es capaz de poder predecir los gustos de cada persona, mediante un entrenamiento previo. Un ejemplo muy claro son las famosísimas cookies que encontramos en la gran mayoría de las páginas web.

Aunque probablemente estos dos campos sean donde más repercusión e influencia pueda llegar a tener la inteligencia artificial, no son los únicos y es por ello que en la Figura 16 se puede ver otros campos en los que cobra relevancia.

Tal y como se ha expuesto, la inteligencia artificial es un campo con un gran interés internacional, por lo que poder disponer de una extensión de hardware libre para un núcleo libre y de arquitectura también libre que mejora el rendimiento de los algoritmos parece ser un proyecto de gran interés para ser utilizado.

Figura 16. Infografía sobre las principales aplicaciones de la inteligencia artificial. [11]

4.2 OBJETIVOS

Se tienen varios objetivos con el desarrollo de este trabajo, los cuales se intentarán abordar de la mejor manera posible y son los siguientes:

- Conocer cómo se diseñan los microprocesadores.
- Ser capaz de implementar un nuevo componente al microprocesador.
- Trabajar en un tema del que no se tiene ningún conocimiento.
- Aprender sobre la arquitectura RISC-V.
- Aprender sobre el núcleo NeoRV32.
- Tener un conocimiento más amplio sobre el lenguaje VHDL.
- Mejorar en el uso de herramientas para la simulación de circuitos digitales.

El primero consiste en conocer cómo funciona a nivel de diseño de hardware un microprocesador, por lo que se partirá de un núcleo ya desarrollado. De esta manera se podrá estudiar cómo se comunican las distintas partes de este, para luego poder aplicar todos los conocimientos en desarrollar nuestra parte del microprocesador.

El objetivo más importante que se persigue es el de ser capaz de implementar una extensión al microprocesador, segundo objetivo de la lista. En concreto se va a implementar la extensión Zve32x, que permite al procesador operar con números de 32 bits en coma fija de manera mucho más eficiente que si se usase la extensión escalar equivalente.

En cuanto al tercer objetivo consiste en adentrarse en un tema del que no se tenía ningún conocimiento antes de empezar el TFG. En este caso es el haber elegido la extensión de procesamiento vectorial, ya que no se sabría ni de su existencia si no hubiese sido por este trabajo.

Si se pone la vista en el último objetivo, este consiste en poder desarrollar simulaciones propias, si bien es cierto que en otras asignaturas se han realizado simulaciones de este tipo en dichas asignaturas los programas para simular nuestros circuitos ya nos los daban.

Con la consecución de sendos objetivos se tendrá un conocimiento mucho más amplio de cómo se diseñan los microprocesadores, que es el último gran objetivo que se persigue y fue la razón principal por la que se decidió realizar este Trabajo Fin de Grado.

4.3 METODOLOGÍA

La metodología que se usará para desarrollar el proyecto consiste en el estudio y conocimiento previo de cada una de las partes que engloba este proyecto. Con ello se hace referencia al estudio del funcionamiento del núcleo, de la arquitectura RISC-V y de la extensión Zve32x. Una vez se tiene una noción básica de su funcionamiento se procederá a realizar simulaciones y pruebas para comprobar dichos conocimientos. El siguiente paso será el diseño de los circuitos digitales, para posteriormente ser descrito en VHDL. Terminados estos se volverán a simular y probar con el objetivo de comprobar que realizan todas las funcionalidades que se les exige, corrigiendo los posibles errores que se detecten. En la Figura 17 podemos ver un diagrama de flujo que expresa de manera clara lo expuesto.

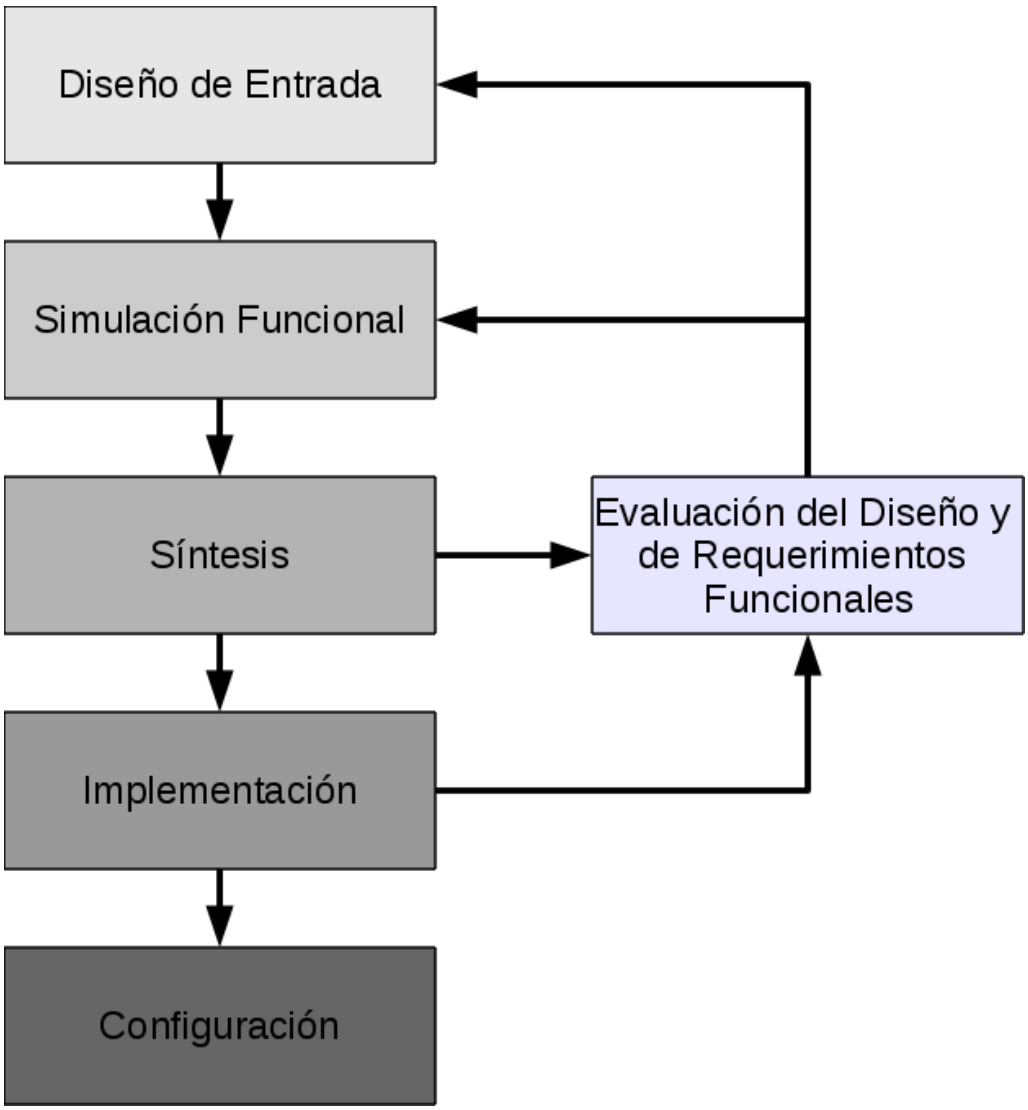


Figura 17. Diagrama de flujo básico sobre el diseño e implementación de circuitos digitales. [12]

4.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

Mediante la Tabla 1 se representará de manera aproximada la planificación para desarrollar la extensión.

Tarea	*SEP*	*OCT*	*NOV*	*DIC*	*ENE*	*FEB*	*MAR*	*ABR*	*MAY*	*JUN*	*JUL*
Elección de TFG e información											
Trabajo previo											
Documentación sobre microprocesadores											
Estudio del núcleo NeoRV32											
Diseño de los circuitos y descripción en VHDL											
Implementación en una FPGA											
Redacción de la memoria											

Tabla 1. Planificación del TFG.

Para poder realizar una estimación económica lo más precisa posible se ha decidido desglosar el número de horas dedicado a cada una de las tareas, de manera que se obtiene el total, que al multiplicarlo por el sueldo medio de un Ingeniero Técnico Industrial nos dará el gasto en mano de obra. A este gasto hay que añadir el coste del software, los componentes, materiales y herramientas. Por último, a este total hay que añadirle el porcentaje de gastos generales y beneficio industrial.

Tareas realizadas	*Tiempo invertido en horas*
Estudio y documentación necesaria	50
Diseño de los circuitos digitales	30
Descripción de los circuitos en VHDL	50
Pruebas y simulaciones de los circuitos	50
Desarrollo de la memoria	30
Total	**210**

Tabla 2. Tabla de número de horas destinadas a cada tarea.

Teniendo en cuenta que el sueldo medio de un Ingeniero Técnico Industrial en España es de 14,01€/hora [13].

$$E.\ 1 \quad \textbf{\textit{Sueldo Ingeniero Técnico Industrial}} = 210\ h\ \cdot {}^{14.01€}/_{h} = \mathbf{2942.1}\ €$$

Para la amortización de los equipos para tratamiento de la información y sistemas y programas informáticos se usará un coeficiente lineal del 20%, por lo que se tiene un periodo de amortización de 5 años [14]. Para el cálculo de los costes atribuidos al proyecto se asume que durante un mes se trabajan 173,33 horas [15], por lo que se ha trabajado el equivalente a 1,21 meses.

Elementos Patrimoniales	*Coste del producto*	*Amortización*	*Costes atribuidos al proyecto*
Ordenador portátil	1100 €	5 años	22.18 €
ModelSim-Altera	0 €	5 años	0 €
Quartus II	0 €	5 años	0 €
FPGA	320.24 €	5 años	6.46 €
Office Hogar y Estudiantes 2021	149 €	5 años	3 €
Windows 10 Pro	259 €	5 años	5.22 €
Total	-	-	**36.86 €**

Tabla 3. Tabla de costes de los elementos relacionados con el proyecto.

E. 2 $$\textbf{\textit{Costes atribuidos al proyecto}} = Coste\ del\ producto \cdot \frac{Tiempo\ trabajado}{Tiempo\ de\ amortización}$$

E. 3 $$\textbf{\textit{Costes totales antes de beneficio}} = 2942.1 + 36.86 = \mathbf{2978.93}\ €$$

Una vez se tienen los costes totales falta por añadir los gastos generales y beneficio industrial. Con base en el expediente 40/19 de la junta consultiva de contratación pública del estado [16] se asumirá un 14 % de gastos generales y un 6 % de beneficio industrial.

E. 4 $$\textbf{\textit{Costes totales despues de beneficio}} = 2978.93 \cdot 1.2 = \mathbf{3574.72}\ €$$

Capítulo 5. SISTEMA/MODELO DESARROLLADO

5.1 ANÁLISIS DEL SISTEMA

Antes de comenzar a diseñar la extensión, como se parte de un núcleo totalmente desconocido que ha sido diseñado por otros desarrolladores, las primeras horas se han de dedicar a entender cómo funciona el núcleo y que conexiones internas tienen.

5.1.1 ESTUDIO DEL NÚCLEO NEORV32

Para comenzar se descargó desde la página de github el núcleo NeoRV32 [17], en el cual se incluyen varias carpetas con distintos archivos. En un primer momento se investigó que contenía cada una de las carpetas, ya que la mayoría de ellas contenían un archivo titulado readme.md en el que se hacía una pequeña descripción de que contenía, tal y como se puede ver en la Figura 18.

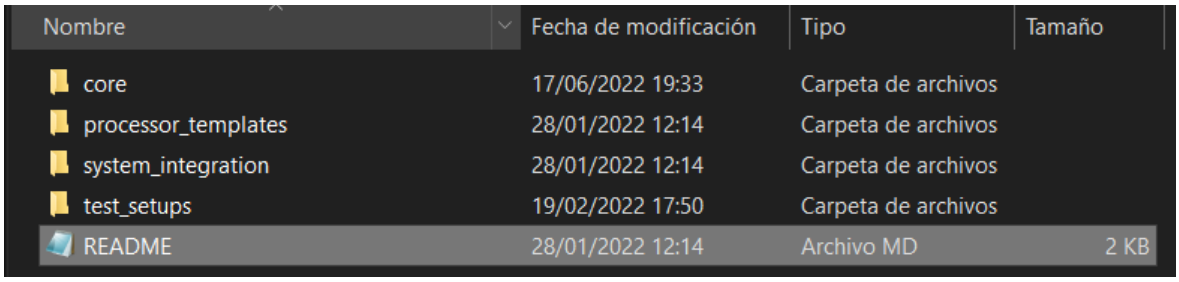


Figura 18. Archivo readme.

Tras localizar los archivos que describían los circuitos digitales del núcleo, los archivos VHDL, lo primero era cargar estos en Quartus II y conseguir compilarlos todos. Para ello se disponía de una guía básica [18] en la que se explicaban los pasos a seguir. Lo primero es crear un proyecto en el que se indica que FPGA y de que familia es en la que se van a sintetizar los circuitos descritos, seguidamente se cargan todos los archivos, están contenidos en la carpeta NeoRV32/rtl/core. Se tiene que crear una nueva librería llamada NeoRV32, ya que la gran mayoría de los archivos necesitan acceder al paquete de recursos

que se introducirá dentro de esta librería. Ahora se elige uno de los archivos de prueba que nos ofrecen, en este caso se elige el neorv32_testsetup_bootloader.vhd ya que incluye UART, y lo seleccionamos como *"top entity"*. Por último, solo faltaría ajustar las configuraciones genéricas para que estas se adapten a nuestra FPGA, tales como la frecuencia del reloj, la cantidad de memoria que se va a usar, etcétera, tal y como se puede observar en el Código 1.

```
entity neorv32_test_setup_bootloader is
  generic (
    -- adapt these for your setup --
    CLOCK_FREQUENCY   : natural := 50000000; -- clock frequency of clk_i in Hz
    MEM_INT_IMEM_SIZE : natural := 2*1024;   -- size of processor-internal
instruction memory in bytes
    MEM_INT_DMEM_SIZE : natural := 1*1024     -- size of processor-internal data
memory in bytes
  );
```

Código 1. Ajustes generales de la CPU.

En principio ya estaría todo ajustado como para ser compilado, y por ello se compiló. Sin embargo, la compilación tuvo errores. Tras analizar a que se referían los errores, se comprendió que los anteriores desarrolladores decidieron que en el caso de la memoria la entidad se iba a declarar en un archivo diferente al de la propia memoria, cuando en el resto de los archivos la entidad se definía en el propio archivo. La solución que se adopto fue la más sencilla, pero al mismo tiempo la más práctica, definir la entidad en el mismo archivo de la memoria. De esta manera ahora al ser compilado ya no se obtenía ningún error.

Ahora que ya se ha conseguido que el núcleo compile el siguiente paso consiste en estudiar cómo funciona, es decir, ir viendo como han ido conectando cada uno de los componentes, las señales de comunicación interna que se usan, etcétera. Lo primero es leer cada uno de los encabezados de los que disponen cada uno de los archivos VHDL, en los que se da una pequeña descripción de la función principal del bloque y de que otros bloques está compuesto, tal y como se puede ver en la Figura 19.

```
-- ################################################################################################################
-- # << NEORV32 - CPU Top Entity >>                                                                               #
-- # ***************************************************************************************************** #
-- # NEORV32 CPU:                                                                                                 #
-- # * neorv32_cpu.vhd                   - CPU top entity                                                         #
-- #   * neorv32_cpu_alu.vhd             - Arithmetic/logic unit                                                  #
-- #     * neorv32_cpu_cp_bitmanip.vhd   - Bit-manipulation co-processor                                          #
-- #     * neorv32_cpu_cp_fpu.vhd        - Single-precision FPU co-processor                                      #
-- #     * neorv32_cpu_cp_muldiv.vhd     - Integer multiplier/divider co-processor                                #
-- #     * neorv32_cpu_cp_shifter.vhd    - Base ISA shifter unit                                                  #
-- #   * neorv32_cpu_bus.vhd             - Instruction and data bus interface unit                                #
-- #   * neorv32_cpu_control.vhd         - CPU control and CSR system                                             #
-- #     * neorv32_cpu_decompressor.vhd  - Compressed instructions decoder                                        #
-- #   * neorv32_cpu_regfile.vhd         - Data register file                                                     #
-- # * neorv32_package.vhd               - Main CPU & Processor package file                                      #
-- #                                                                                                              #
-- # Check out the CPU's online documentation for more information:                                               #
-- #  HQ:         https://github.com/stnolting/neorv32                                                            #
-- #  Data Sheet: https://stnolting.github.io/neorv32                                                             #
-- #  User Guide: https://stnolting.github.io/neorv32/ug                                                          #
-- # ***************************************************************************************************** #
```

Figura 19. Captura del encabezado del archivo neorv32_cpu.vhd

Una vez que ya se tiene una pequeña idea de cómo está compuesto y que señales internas usa, lo mejor es centrarse en un único archivo y comprender totalmente su funcionamiento. La mejor opción sería empezar con un archivo muy sencillo para luego poder ir escalando en nivel de complejidad y que realizase un proceso similar al de la extensión a desarrollar, por lo que se eligió empezar con la extensión M, la cual está localizada en la ALU y cuya función es realizar las operaciones aritméticas de multiplicación y división entre dos números escalares. Lo primero es leer todo el código y comprender su funcionamiento. Cuando ya tiene una buena noción de su funcionamiento se pasa a desarrollar un testbench para simular las entradas y ver que las salidas son las esperadas, de manera que se pueda comprobar el conocimiento de la extensión M.

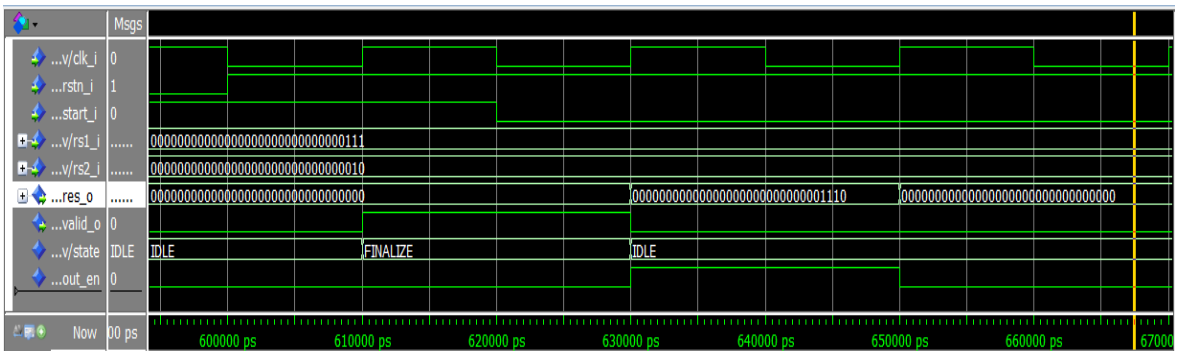


Figura 20. Salida del testbench del archivo neorv32_cpu_cp_muldiv.vhd (Extensión M).

Ahora se analizará la Figura 20, para comprobar que está bien hecho el testbench. Hasta los 610ns, a pesar de tener la señal de iniciar la multiplicación (start_i), no empieza a realizar la operación, ya que se tenía el reset (rstn_i) activado. Tarda un ciclo de reloj en realizar la operación y otro en sacarla, ya que esta activado el enable de la salida. Al siguiente ciclo de reloj al tener desactivada la señal de iniciar la multiplicación la salida se pone a 0 y se desactiva el eneable de la salida (out_en).

Una vez que se ha comprobado el buen conocimiento de la extensión M ahora hay que subir en el nivel de jerarquía, y es por ello que los esfuerzos se centran en comprender el funcionamiento de la ALU, por lo que se va a seguir el mismo procedimiento anterior. Ahora se tiene nuevas señales de control, con las que el núcleo va a informar a la ALU de que operaciones realizar y donde guardar la información que procese.

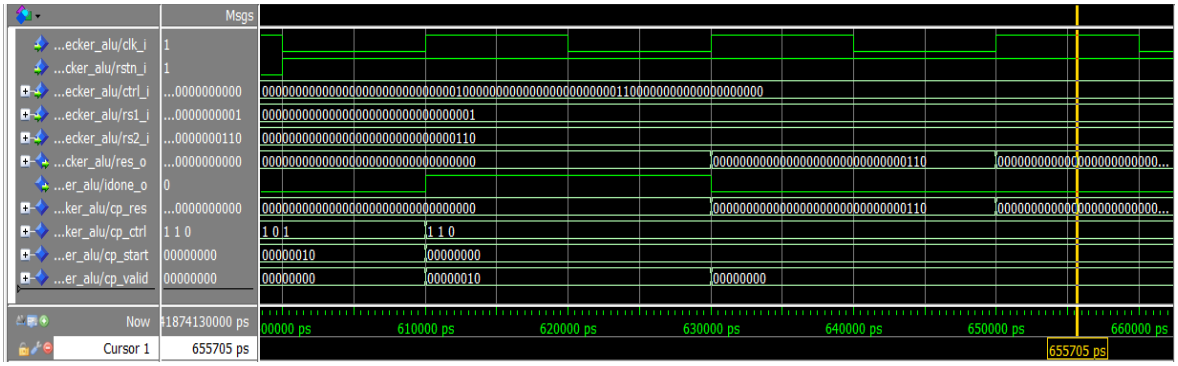


Figura 21. Salida del testbench del archivo neorv32_cpu_alu.vhd.

Se analiza la Figura 21. Ahora se tiene la señal de control (ctrl_i) con la que el núcleo se comunica con todos los componentes que lo conforman, por eso tiene una longitud de 74 bits. Esta señal está codificada de manera que active la ALU y le ordene realizar una multiplicación de los dos números binarios de entrada (rs1_i y rs2_i), pero hasta los 600ns no empieza, ya que el reset (rstn_i) estaba en 0, en el siguiente ciclo de reloj realiza la multiplicación, y se ve porque la señal done_o se activa. En el siguiente ciclo de reloj

transmite el resultado mediante la señal res_o, ya que en el ciclo anterior la señal cp_valid se codificó automáticamente de manera correcta. Esta salida va directamente al registro escalar, para que luego pueda ser almacenado en la memoria que corresponda. Tras este testbench se comprueba el buen conocimiento de las señales con las que el núcleo se comunica con los distintos componentes.

5.1.2 ESTUDIO DE LA EXTENSIÓN ZVE32X

Una vez que ya se tiene un conocimiento más profundo sobre el funcionamiento del núcleo en el que se implantará la extensión el siguiente paso es entender el funcionamiento y conocer los requisitos de la extensión. Para ello se comienza estudiando la especificación oficial que contiene toda la información necesaria para implementarla. En el documento [19] en el que se recogen todas las extensiones vectoriales compatibles con RISC-V, que contiene descritas todas las instrucciones necesarias por cada extensión. Se va a implementar concretamente la extensión Zve32x, que ha sido diseñada para sistemas empotrados y que opera registros vectoriales con números de 32 bits en coma fija. El registro contiene 4 bytes.

Como nunca se había trabajado con núcleos que operasen de manera vectorial se buscó información de su funcionamiento. De esta manera se consultaron diversos estudios, gracias a los cuales se empezó a tener una concepción básica de que es lo que hacían. Básicamente este tipo de procesadores realizan varios tipos de operaciones con vectores de manera mucho más eficiente que los escalares, ya que son capaces de trabajar con todos los elementos de un vector al mismo tiempo, mientras que los escalares solo pueden operar con un elemento del vector en cada instrucción.

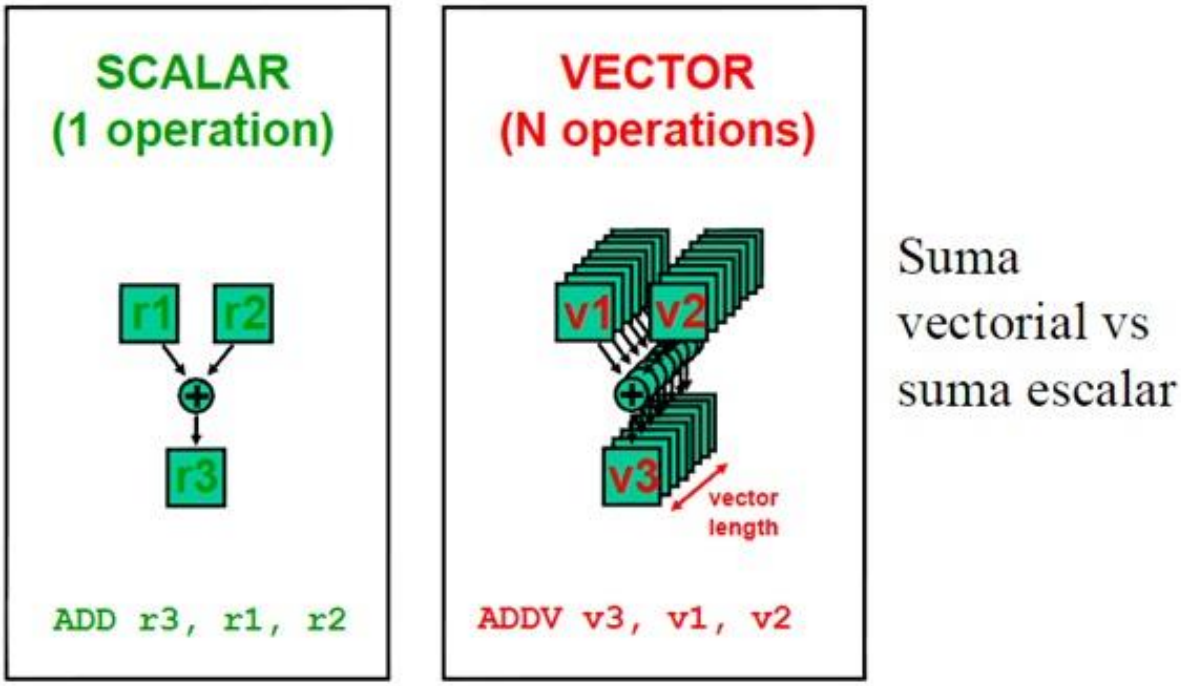


Figura 22. Imagen comparativa entre la suma escalar y la suma vectorial. [1]

La Figura 22 resume muy bien la principal diferencia. Supóngase que tenemos dos vectores de datos de 10 elementos y que nuestro procesador vectorial es capaz de operar en un ciclo de reloj con 5 elementos de cada vector. La suma la tendrá finalizada en tan solo 2 ciclos de reloj, mientras que un procesador escalar necesitará 10 ciclos para obtener el mismo resultado, es decir, es 5 veces más lento.

Además de permitir operaciones aritméticas, la extensión tiene que permitir otras funcionalidades:

- La carga y descarga de datos, para lo que se diseñará un registro vectorial que estará conectado con la memoria RAM.
- Dar soporte para tener vectores con elementos de distintas longitudes, es decir, vectores con todos los elementos de 8, 16 y 32 bits, para lo que se diseñara un parámetro llamado SEW (Selected Element Width).

- Tener instrucción de enmascaramiento. Con este tipo de instrucciones se utiliza un vector máscara para comprimir o expandir un vector a un vector índice.
- Tener instrucciones de permutación. Estas instrucciones permiten la permutación de bits en un vector.
- Leer y escribir en los CSR de la extensión.

La forma en la que se tratará de dar servicio a todas estas funcionalidades consistirá en ir creando primero los bloques más esenciales, para posteriormente ir añadiendo el resto de las funcionalidades. Una vez que ya tengamos todos los bloques diseñados y con todas las funcionalidades cubiertas se buscara la manera de integrar la extensión al núcleo.

Para ello se observará cómo se ha integrado la ALU y el registro escalar, y valorando la posibilidad de seguir la misma metodología. Ambos componentes están localizados dentro del archivo neorv_cpu.vhd, por lo que se tiene que estudiar si las señales que se utiliza en ese nivel de jerarquía son las suficientes como para que nuestros componentes funcionen. Para ello primero se observa que componentes están englobados y que funciones realizan, descritos al inicio del código.

```
--
#################################################################################
-- # << NEORV32 - CPU Top Entity >>
--********************************************************************************#
-- # NEORV32 CPU:
-- #
-- # * neorv32_cpu.vhd                   - CPU top entity
-- #
-- #   * neorv32_cpu_alu.vhd             - Arithmetic/logic unit
-- #
-- #     * neorv32_cpu_cp_bitmanip.vhd   - Bit-manipulation co-processor
-- #
-- #     * neorv32_cpu_cp_fpu.vhd        - Single-precision FPU co-processor
-- #
-- #     * neorv32_cpu_cp_muldiv.vhd     - Integer multiplier/divider co-
-- #                                       processor
-- #     * neorv32_cpu_cp_shifter.vhd    - Base ISA shifter unit
-- #
-- #   * neorv32_cpu_bus.vhd             - Instruction and data bus interface
-- #                                       unit
-- #   * neorv32_cpu_control.vhd         - CPU control and CSR system
-- #
-- #     * neorv32_cpu_decompressor.vhd  - Compressed instructions decoder
-- #
```

```
-- #   * neorv32_cpu_regfile.vhd         - Data register file
-- #
-- # * neorv32_package.vhd               - Main CPU & Processor package file
```

Código 2. Inicio del código del archivo noerv32_cpu.vhd, que describe sus componentes.

Mediante el Código 2 se puede concretar si es plausible integrar dentro de este archivo la extensión. Contiene el archivo neorv32_cpu_control.vhd, el cual es un archivo que permite conectarse al bus de datos que permite conectarse directamente con la memoria RAM. También contine el archivo neorv32_cpu_control que permite tanto la lectura como la escritura de las señales de control del núcleo y de los CSR. Con estos dos archivos tenemos la mayoría de las señales que se utilizaran en la extensión, por lo que es el lugar idóneo en la jerarquía para implementar la extensión, y en caso de necesitar alguna nueva se modificarían los archivos pertinentes para tenerlas. En la parte de diseño se verá como conectar las distintas señales, ya que se necesitarán multiplexores para distinguir que datos van a la ALU y el registro vectorial, y cuales a las unidades escalares.

5.1.3 TOOLCHAIN PARA CREAR ARCHIVOS DE IMAGEN

Para poder realizar unas simulaciones buenas y precisas se crearán archivos en ensamblador que nos permitirán asegurar el buen funcionamiento de los componentes desarrollados. Para ello se necesitará de una toolchain que nos permita pasar el código en ensamblador a código máquina, es por ello que se usará una toolchain preconstruida que proporciona la guía del núcleo llamada riscv-gnu-toolchain. Para poder usarla se tendrá que usar la consola de comando de Linux, y como se está utilizando Windows se instalará un programa que nos permitirá usar la consola de Linux en Windows. Para su instalación se descarga el archivo de la página web [20] y se descomprime, se ejecutan unos comandos que nos indican y por último se añade al PATH los archivos de la carpeta bin, tal y como se ve en la Figura 23. Comando para añadir los archivos de la carpeta bin al PATH.. Cada vez que se inicie la consola hay que ejecutar el comando de la Figura 23.

```
$ export PATH=$PATH:/mnt/c/Users/moihe/Documents/TFG_Riscv/riscv32-unknown-elf.gcc-10.2.0.rv32i.ilp32.newlib/bin
```

Figura 23. Comando para añadir los archivos de la carpeta bin al PATH.

Para comprobar el buen funcionamiento de la toolchain se ejecuta el comando "*make check"* que comprueba todos los componentes de la misma, indicando cuales no funcionan si surge el caso.

Una vez terminada la instalación se creará un archivo en ensamblador de prueba. Se aprovecharán los archivos de la carpeta blink_led que ya contiene archivos creados. El archivo creado contendrá el código que se muestra en el Código 3.

```
.file   "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li   t1, 0 /* initialize counter */


blink_loop:

  addi t0, t0, 1

  j    blink_loop

.end
```

Código 3. Código en ensamblador que suma constantemente 1 en el registro t0.

Este programa consiste en sumar 1 en el registro t0, y sirve simplemente para comprobar el buen funcionamiento de la toolchain y poder usarlo para simular el núcleo. Para ello también se utilizará un testbench ya creado para simular el núcleo, llamado neorv32_tb.simple.vhd.

Para crear el archivo de imagen se usa el comando "*make clean_all install"*, cuya función es borrar todos los archivos de imagen existentes, crear uno nuevo en base al programa en ensamblador e instalarlo en el archivo neorv32_aplication_image.vhd, para que lo pueda

usar el testbench y poder simularlo. La toolchain también genera la ROM para poder iniciar el procesador. En la Figura 24 se puede ver en funcionamiento el comando.

```
moi@[redacted]:/mnt/c/Users/moihe/documents/TFG_Riscv/neorv32_master/sw/example/blink_led_test$ make clean_all install
Memory utilization:
   text    data     bss     dec     hex filename
    404       0       0     404     194 main.elf
Compiling ../../../sw/image_gen/image_gen
Installing application image to ../../../rtl/core/neorv32_application_image.vhd
```

Figura 24. Ejecución del comando "make clean_all install".

Una vez instalado el programa en ensamblador genera un archivo como el del Código 4, que consiste en las instrucciones que hemos programado en ensamblador, pero ahora convertidas en código máquina, un numero en hexadecimal de 32 bits, es decir, este archivo es la descripción en VHDL de una ROM con el programa en ensamblador que hemos creado, la cual se integra dentro del núcleo NeoRV32.

```
-- The NEORV32 RISC-V Processor, https://github.com/stnolting/neorv32
-- Auto-generated memory init file (for APPLICATION) from source file
<blink_led_test/main.bin>
-- Size: 436 bytes

library ieee;
use ieee.std_logic_1164.all;

library neorv32;
use neorv32.neorv32_package.all;

package neorv32_application_image is

  constant application_init_image : mem32_t := (
    00000000 => x"00000037",
    00000001 => x"80002117",
    00000002 => x"ff810113",
    00000003 => x"80000197",
    00000004 => x"7f418193",
    00000005 => x"00000517",
    00000006 => x"12850513",
    00000007 => x"30551073",
    00000008 => x"34151073",
    00000009 => x"30001073",
    00000010 => x"30401073",

    […]

    00000129 => x"00012883",
    00000130 => x"00810113",
    00000131 => x"011801d7",
```

```
    00000132 => x"091801d7"
  );

end neorv32_application_image;
```

Código 4. Parte del archivo noerv32_application_image.vhd

El siguiente paso consiste en ejecutar la simulación del archivo neorv32_tb.simple.vhd, con el fin de comprobar que realiza la suma progresiva.

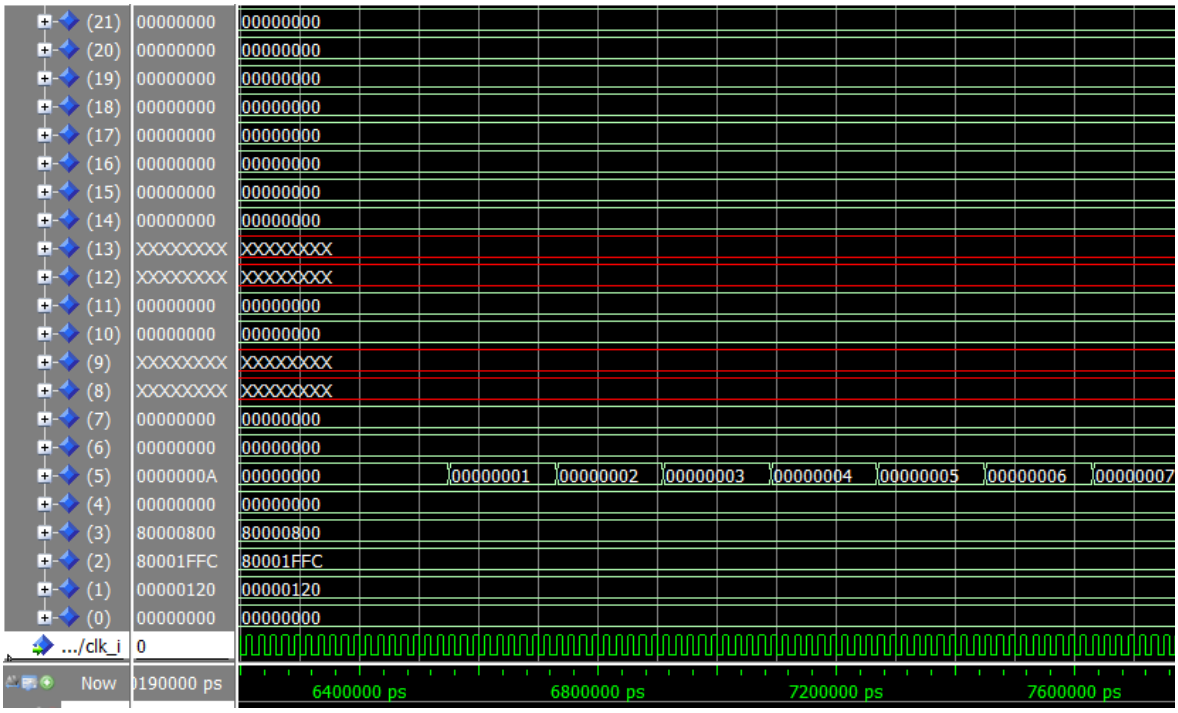


Figura 25. Simulación del programa en ensamblador.

Tal y como se puede ver en la Figura 25, la simulación es correcta, ya que el registro t0 correspondiente al número 5, se va sumando 1 progresivamente, por lo que la toolchain funciona correctamente, y se podrá usar para realizar simulaciones.

Registro	Nombre simbólico	Descripción	¿Preservado?
x0	zero	Constante Cero	–
x1	ra	Dirección de retorno	No
x2	sp	Puntero de pila	Si
x3	gp	*global pointer*	–
x4	tp	*Thread pointer*	–
x5-7	t0-2	Valores temporales	No
x8	s0/fp	Variable/*frame pointer*	Si
x9	s1	Variable	Si
x10-11	a0-1	Argumentos/Valor de retorno	No
x12-17	a2-7	Argumentos	No
x18-27	s2-11	Variables	Si
x28-31	t3-6	Valores temporales	No

Figura 26. Convención sobre el uso de registros. [21]

5.1.4 ENSAMBLAJE DE LAS NUEVAS INSTRUCCIONES

Como se está implementando una nueva extensión hay que comprobar que la toolchain que se está utilizando sea capaz de procesar estas nuevas instrucciones. Para ello es muy sencillo, se prueba escribiendo una de estas nuevas instrucciones en ensamblador y en caso de soportarlas proporcionará el código máquina para que puedan ser ejecutadas. En caso contrario nos dará un error. Es por ello que se desarrolla el programa del Código 5.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li a0, 1

  li a1, 2

  li a2, 3

  vsetvl a2, a1, a0 /*Instruccion vectorial*/

.end
```

Código 5. Programa con una instrucción vectorial.

Ahora simplemente se ejecuta el comando *"make clean_all install"* en la consola de comandos.

```
moi@                :/mnt/c/Users/moihe/documents/TFG_Riscv/neorv32_master/sw/example/blink_led_test2$ make clean_all  install
blink_led_in_asm.S: Assembler messages:
blink_led_in_asm.S:14: Error: unrecognized opcode `vsetvl a2,a1,a0'
make: *** [../../../sw/common/common.mk:172: blink_led_in_asm.S.o] Error 1
```

Figura 27. Ejecución del programa de prueba de la instrucción vectorial.

Tal y como se puede ver en la Figura 27, concretamente en la tercera línea, donde se lee *"Error: unrecognized opcode 'vsetvl a2, a1, a0'"*, la toolchain utilizada no tiene implementadas las instrucciones vectoriales.

Llegado a este punto se tienen dos opciones.

5.1.4.1 Instalar una nueva toolchain

La primera es buscar una toolchain distinta que sí que nos permita utilizar instrucciones vectoriales, sin embargo, tras una gran búsqueda se encontraron pocas opciones. Se probó a instalarlas, pero como se está utilizando Linux dentro de Windows nunca se consiguió hacerlas funcionar. Es por ello que se pensó en instalar una máquina virtual de Linux, pero habría que instalar también todos los programas y archivos de nuevos en esta maquina virtual, y esta solución tampoco garantizaba que funcionase de manera segura. Es por ello que finalmente se optó por la segunda opción.

5.1.4.2 Ensamblar a mano

Esta opción consiste en traducir estas nuevas instrucciones a código máquina, pero en vez de hacerlo un programa, será realizado por nosotros. Para ello se tendrá en cuenta como se codifican todas estas instrucciones. Para dejarlo un poco más claro se realizará un ejemplo con tan solo una instrucción. Proceso que se tendrá que replicar para todas aquellas nuevas instrucciones vectoriales que se implementen.

Por ejemplo, se quiere pasar a código maquina la instrucción vsetvl, lo primero que se necesita es saber a qué corresponden cada uno de los 32 bits que la componen. Es por ello que se busca esta información, obteniendo algo similar a lo mostrado en la Figura 28.

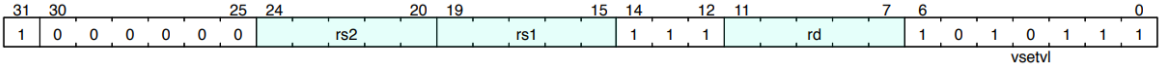


Figura 28. Formato de la instrucción vsetvl. [19]

Además, se tiene que saber para que se utilizan cada uno de los bits en la arquitectura RISC-V, es por ello que es necesaria la información de la Figura 29.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Figura 29. Formatos de instrucción de la arquitectura RV32I. [22]

Una vez se tiene tanto la codificación de la nueva instrucción, como el formato que se utiliza en RISC-V, en este caso es de tipo R, se procede a asignar los valores.

Para ello supóngase la instrucción *vsetvl a2, a1, a0.* El registro de destino seria a2, por tanto, en los bits correspondientes a rd, se introduciría la dirección del registro a2, que en este caso gracias a la Figura 26 sería el x12 → 1100_2. Si se hace lo mismo para todos los bits finalmente se obtiene:

$$1000\ 0000\ 1010\ 0101\ 1111\ 0110\ 0101\ 0111_2 = 80A5F657_{16}$$

Ahora se tendría que añadir el numero en hexadecimal al archivo *neorv32_application_image.vhd,* tal y como se ve en el Código 6.

```
-- The NEORV32 RISC-V Processor, https://github.com/stnolting/neorv32
-- Auto-generated memory init file (for APPLICATION) from source file
<blink_led_test/main.bin>
-- Size: 436 bytes

library ieee;
use ieee.std_logic_1164.all;

library neorv32;
use neorv32.neorv32_package.all;

package neorv32_application_image is

  constant application_init_image : mem32_t := (
    00000000 => x"00000037",

    […]

    00000105 => x"00128293",
    00000106 => x"00f29073",
    00000107 => x"80A5F657" -- vsetvl a2, a1, a0
);

end neorv32_application_image;
```

Código 6. Instrucción vsetvl añadida al archivo neorv32_application_image.vhd.

Ahora ya estaría todo listo para que el núcleo ejecutase esa instrucción.

5.2 DISEÑO Y SIMULACIÓN

La estrategia de diseño se basará en desarrollar cada uno de los componentes por separado, teniendo en cuanto las especificaciones que deberá cumplir cada uno. Una vez este descrito en VHDL se realizará un testbench para comprobar que todo funciona correctamente, y en caso afirmativo se pasará al siguiente bloque a diseñar. Para su diseño seguiremos la especificación oficial que describe cada una de las funcionalidades.

5.2.1 DISEÑO Y SIMULACIÓN DE LOS CSR

Se tiene que diseñar 6 nuevos CSR para la extensión, y son los que se representan en la Figura 30.

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

Figura 30. Tabla descriptiva de los nuevos CSR. [19]

5.2.1.1 Diseño de los CSR

Para poder añadir los nuevos CSR lo primero que se tiene que hacer es fijarse en donde están creados todos los otros CSR que ya utiliza el núcleo. De esta manera se encuentra que están localizados en el archivo neorv32_cpu_control.vhd, por lo que se añaden dentro de la clase, ya creada, csr_t. En cuanto a las longitudes de los mismos, se siguen las indicaciones de la especificación oficial, quedando descritos tal y como se puede ver en el Código 7.

```
type csr_t is record

[…]

  vstart            : std_ulogic_vector(7 downto 0);
  vtype             : std_ulogic_vector(31 downto 0);
  vl                : std_ulogic_vector(5 downto 0);
  vlenb             : std_ulogic_vector(7 downto 0);
  vxrm              : std_ulogic_vector(31 downto 0);
  vxsat             : std_ulogic_vector(31 downto 0);
  vcsr              : std_ulogic_vector(2 downto 0);

end record;
signal csr : csr_t;
```

Código 7. Código descriptivo de los nuevos CSR.

Una vez creados se tiene que indicar también su dirección de memoria. Para saber el lugar donde tiene que ir indicada se sigue la misma estrategia que anteriormente, descubriéndose que están en el archivo neorv32_package.vhd. Se introducirán como constantes en este archivo, tal y como se ve en el Código 8.

```
-- Vector Extension
package neorv32_package is
  […]
  constant csr_vstart_c         : std_ulogic_vector(11 downto 0) := x"008";
  constant csr_vsat_c           : std_ulogic_vector(11 downto 0) := x"009";
  constant csr_vxrm_c           : std_ulogic_vector(11 downto 0) := x"00A";
  constant csr_vcsr_c           : std_ulogic_vector(11 downto 0) := x"00F";
  constant csr_vl_c             : std_ulogic_vector(11 downto 0) := x"C20";
  constant csr_vtype_c          : std_ulogic_vector(11 downto 0) := x"C21";
  constant csr_vlenb_c          : std_ulogic_vector(11 downto 0) := x"C22";
  […]
```

Código 8. Direcciones de memoria de los CSR.

Una vez definidos los CSR hay que implementar el circuito de escritura y lectura de los mismos. Para ello se busca el archivo destinado para esta función, el archivo neorv32_cpu_control.vhd. Para el desarrollo hay que fijarse como se ha implementado para el resto de los CSR y adaptarlo. Se tiene en cuenta que los cuatro primeros son de lectura y escritura, mientras que los tres siguientes son solo de lectura. De esta manera se implementa tal y como se puede ver en el Código 9 y el Código 10.

```
-- Control and Status Registers - Vector Extension ------------------------
-- -----------------------------------------------------------------------------
csr_vector: process(rstn_i, clk_i)
  begin
    -- Vector CSRs
    if (rstn_i = '0') then
      csr.vstart <= (others => def_rst_val_c);
      csr.vxsat  <= (others => def_rst_val_c);
      csr.vxrm   <= (others => def_rst_val_c);
      csr.vcsr   <= (others => def_rst_val_c);

    elsif rising_edge(clk_i) then

      -- [m]vstart --
      if (CPU_EXTENSION_RISCV_Zve32x = true) then
        if (csr.we = '1') and (csr.addr = csr_vstart_c) then -- write access
          csr.vstart(7 downto 0) <= csr.wdata(7 downto 0);
        end if;
      else
        csr.vstart <= (others => '-');
```

```
        end if;

        -- [m]vxsat --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          if (csr.we = '1') and (csr.addr = csr_vxsat_c) then -- write access
            csr.vxsat(31 downto 0) <= csr.wdata(31 downto 0);
            csr.vcsr(0)            <= csr.wdata(0);
          end if;
        else
          csr.vxsat <= (others => '-');
        end if;

        -- [m]vxrm --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          if (csr.we = '1') and (csr.addr = csr_vxrm_c) then -- write access
            csr.vxrm(31 downto 0) <= csr.wdata(31 downto 0);
            csr.vcsr(2 downto 1)  <= csr.wdata(1 downto 0);
          end if;
        else
          csr.vxrm <= (others => '-');
        end if;

        -- [m]vcsr --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          if (csr.we = '1') and (csr.addr = csr_vcsr_c) then -- write access
            csr.vcsr <= csr.wdata(31 downto 0);
          end if;
        else
          csr.vcsr <= (others => '-');
        end if;

        -- [m]vtype --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          if(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) = vsetvl_c)
then -- vsetvl instruction enable
            if (csr.we = '1') then -- write access
              csr.vtype <= rs2_i;
              csr.vl <= rd_i(5 downto 0);
              avl <= rs1_i;
             end if;
             elsif(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) =
vsetvli_c) then -- vsetvli instruction enable
            if (csr.we = '1') then -- write access
              csr.vtype(31 downto 8) <= (others => '0');
              csr.vtype(7 downto 5) <= execute_engine.i_reg(instr_funct7_msb_c-4
downto instr_funct7_lsb_c);
              csr.vtype(4 downto 0) <= ctrl_i(ctrl_rf_rs2_adr4_c downto
ctrl_rf_rs2_adr0_c);
              csr.vl <= rd_i(5 downto 0);
              avl <= rs1_i;
             end if;
             elsif(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) =
vsetivli_c) then -- vsetivli instruction enable
```

```
            if (csr.we = '1') then -- write access
              csr.vtype(31 downto 8) <= (others => '0');
              csr.vtype(7 downto 5) <= execute_engine.i_reg(instr_funct7_msb_c-4
downto instr_funct7_lsb_c);
              csr.vtype(4 downto 0) <= ctrl_i(ctrl_rf_rs2_adr4_c downto
ctrl_rf_rs2_adr0_c);
              csr.vl <= rd_i(5 downto 0);
              avl(4 downto 0) <= ctrl_i(ctrl_rf_rs1_adr4_c downto
ctrl_rf_rs1_adr0_c);
              avl(31 downto 5) <= (others => '0');
             end if;
             end if;
                   vsew  <= csr.vtype(5 downto 3);
                    vlmul <= csr.vtype(2 downto 0);
                    vsew_o <= vsew;

                    case vsew is
                      when sew_8  => f_sew <= 8;
                      when sew_16 => f_sew <= 16;
                      when sew_32 => f_sew <= 32;
                      when sew_64 => f_sew <= 64;
                      when others => f_sew <= 8;
                    end case;
                    case vlmul is
                      when lmul_1_8  => f_lmul <= 1/8;
                      when lmul_1_4  => f_lmul <= 1/4;
                      when lmul_1_2  => f_lmul <= 1/2;
                      when lmul_1    => f_lmul <= 1;
                      when lmul_2    => f_lmul <= 2;
                      when lmul_4    => f_lmul <= 4;
                      when lmul_8    => f_lmul <= 8;
                      when others    => f_lmul <= 1;
                    end case;
            else
              csr.vtype <= (others => '-');
            end if;

             vlmax <= f_lmul*VLEN/f_sew;

          if (to_integer(unsigned(avl)) <= vlmax) then
            csr.vl <= avl(5 downto 0);
          elsif (to_integer(unsigned(avl)) <= 2*vlmax) then
            if (to_integer(unsigned(csr.vl)) > vlmax) then
              csr.vl <= std_ulogic_vector(to_signed(vlmax, 6));
            elsif (to_integer(unsigned(csr.vl)) < (to_integer(unsigned(avl))/2))
then
              csr.vl <= std_ulogic_vector(to_signed(to_integer(unsigned(avl))/2,
6));
            end if;
          elsif (to_integer(unsigned(avl)) >= 2*vlmax) then
            csr.vl <= std_ulogic_vector(to_signed(vlmax, 6));
          end if;
```

```
        -- [m]vlenb --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          csr.vlenb <= std_ulogic_vector(to_signed(VLEN/8,8));
        else
          csr.vlenb <= (others => '-');
        end if;

      end if;

    end process csr_vector;
```

Código 9. Código para la escritura de los CSR.

```
-- Control and Status Registers - Read Access -----------------------------------
-- ------------------------------------------------------------------------------
  csr_read_access: process(rstn_i, clk_i)
    variable csr_addr_v : std_ulogic_vector(11 downto 0);
  begin
    if rising_edge(clk_i) then
      csr.rdata <= (others => '0'); -- default output, unimplemented CSRs are
hardwired to zero

     […]

      if (CPU_EXTENSION_RISCV_Zve32x = true) then
        case csr_addr_v is
          when csr_vstart_c => -- [m]vstart (r/w): Cycle counter LOW
           csr.rdata(7 downto 0) <= csr.vstart(7 downto 0);
          when csr_vxsat_c => -- [m]vxsat (r/w): Cycle counter LOW
           csr.rdata(31 downto 0) <= csr.vxsat(31 downto 0);
          when csr_vxrm_c => -- [m]vxrm (r/w): Cycle counter LOW
           csr.rdata(31 downto 0) <= csr.vxrm(31 downto 0);
          when csr_vcsr_c => -- [m]vcsr (r/w): Cycle counter LOW
           csr.rdata(2 downto 0) <= csr.vcsr(2 downto 0);
          when csr_vl_c => -- [m]vl (r/w): Cycle counter LOW
           csr.rdata(5 downto 0) <= csr.vl(5 downto 0);
          when csr_vtype_c => -- [m]vtype (r/w): Cycle counter LOW
           csr.rdata(31 downto 0) <= csr.vtype(31 downto 0);
          when csr_vlenb_c => -- [m]vlenb (r/w): Cycle counter LOW
           csr.rdata(7 downto 0) <= csr.vlenb(7 downto 0);
          when others =>
            NULL; -- not implemented, read as zero
        end case;
      end if;
    end if;
  end process csr_read_access;

  -- CSR read data output --
  csr_rdata_o <= csr.rdata;
```

Código 10. Código para la escritura de los CSR.

Analizando la descripción de estos circuitos encontramos que para poder tanto leer como escribir en estos nuevos CSR se tiene que tener activada la extensión vectorial, ya que son CSR que solo se utilizan para la misma, por lo que no tendría ningún sentido que estuviesen accesibles si la extensión esta desactivada. Es por ello que en todos ellos encontramos una condición booleana, CPU_EXTENSION_RISCV_Zve32x = true, que sirve para identificar si se implementa o no la extensión.

En los CSR de escritura se añade una condición más, csr.we, que simplemente es un csr que verifica si hay una instrucción que quiere escribir, es decir, CSR Write Enable, por lo que cuando este a 1 nos permitirá escribir en los CSR de escritura.

Por último, tanto para los CSR de escritura como de lectura, se tiene que escribir o leer del correcto, y es por ello que se dispone del csr.addr , que almacena la dirección de destino, por lo que simplemente tendremos que comprobar que el contenido de éste coincida con la dirección que anteriormente habíamos declarado. Para todos los CSR la lectura se realiza de la misma manera.

Los CSR vstart, vxsat y vxrm son CSR de lectura y escritura normales, en cambio vcsr y vlenb tienen características diferentes, por lo que se desarrollaran un poco mas.

5.2.1.2 CSR vcsr

Este CSR tiene la cualidad de que contiene los valores de los CSR vxrm y vsat copiados. Es decir, siempre que se escribe en alguno de los dos, automáticamente se escribe también el vcsr en la posición correspondiente, según la Figura 31. También se tiene habilitada la escritura manual del vcsr por si fuese necesario, pero en principio no debería de serlo. También se puede acceder a los valores de vxrm y vxsat a través de vcsr.

Bits	Name	Description
2:1	vxrm[1:0]	Fixed-point rounding mode
0	vxsat	Fixed-point accrued saturation flag

Figura 31. Tabla de codificación del CSR vcsr. [19]

Para lograrlo simplemente se introdujo que cuando se estuviese escribiendo en uno de los csr también se escribiese en vcsr en la posición adecuada.

5.2.1.3 CSR vlenb

Es un csr solo de lectura y que tiene que almacenar la longitud de los vectores del registro vectorial en Bytes. Para conseguirlo simplemente hay que dividir VLEN, constante usada para definir la longitud de los vectores en bits, por 8.

5.2.1.4 Simulación de los CSR

Una vez implementada la escritura y la lectura de los CSR, se tiene que hacer un testbench para poder comprobar su funcionamiento. Para ello se desarrollará un pequeño programa en ensamblador que carga los CSR desde un registro, y que luego lea los CSR y cargue su contenido en otro registro. De esta manera se desarrolla el Código 11.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li t0, 0 /*Incializo t0 a 0*/

  addi t0, t0, 1 /*Sumo 1 en t0*/

  csrw 0x008, t0 /*Escritura de vstart*/

  addi t0, t0, 1 /*Sumo 1 en t0*/

  csrw 0x009, t0 /*Escritura de vxsat*/

  addi t0, t0, 1 /*Sumo 1 en t0*/

  csrw 0x00A, t0 /*Escritura de vxrm*/

  csrr t2, 0x008 /*Lectura de vxstart*/

  csrr t2, 0x009 /*Lectura de vxsat*/

  csrr t2, 0x00A /*Lectura de vxrm*/

  csrr t2, 0x00F /*Lectura de vcsr*/

  csrr t2, 0xC22 /*Lectura de vlenb*/
```

```
.end
```

Código 11. Programa para comprobar la escritura y la lectura de los CSR.

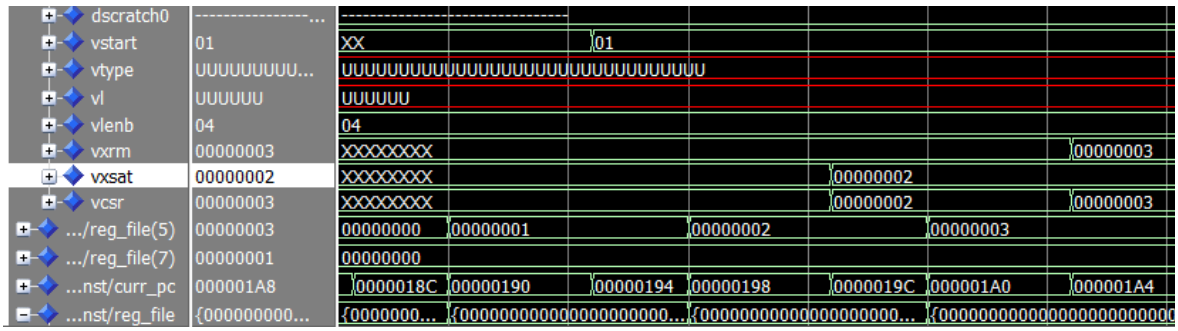


Figura 32. Simulación de la escritura de los CSR.

Analizando la Figura 32 se puede ver como se va cargando progresivamente el registro t0, número 5, a medida que avanzan las instrucciones, curr_pc, y como estos valores se almacenan en los CSR correspondientes. El registro vlenb adquiere ese valor al principio del inicio del programa, ya que este es solo de lectura. Además, se ve como los valores de vcsr son la copia de los registros vxrm y vxsat en cada caso. Cabe destacar que los números han sido elegidos para poder seguir de mejor manera la carga de los datos, ya que en algunos CSR no tiene ningún sentido cargar ese número.

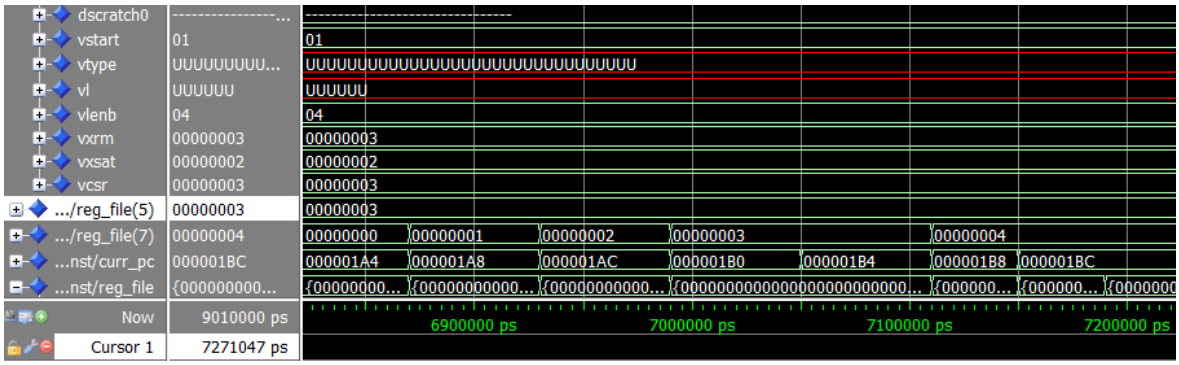


Figura 33. Simulación de la lectura de los CSR.

En la Figura 33 se observa la lectura de los CSR y como estos son almacenados en el registro t2, número 7. En un primer momento el registro está a 0, al llegar a la instrucción de lectura de vstart esta carga su contenido en t2, y así sucesivamente. Tal y como se puede

apreciar, cuando carga el número 3 se pasa el doble del tiempo guardado que el resto, esto es porque primero carga vxrm y luego carga vcsr, pero como ambos tienen el mismo valor parece que el registro no ha cambiado.

Por último, se puede ver que los CSR vtype y vl no tienen ningún valor asignado, esto es porque tienen una instrucción propia para ser cargados, por lo que se verá en otro apartado.

5.2.2 DISEÑO Y SIMULACIÓN DE LAS INSTRUCCIONES VSETVLI, VSETIVLI Y VSETVL

Para poder modificar los CSR vtype y vl se necesita implementar nuevas instrucciones, ya que estos no van a ser modificados simplemente escribiendo en ellos, tal y como lo hacíamos con el resto de los nuevos CSR.

Cada una de estas tres nuevas instrucciones modifican vtype y vl de manera distintas, ya sea usando inmediatos o registros. Es por ello que se verá cómo realizar cada una de las instrucciones por separado.

5.2.2.1 Restricciones del CSR vl

El CSR vl no es un CSR normal, ya que tiene que cumplir una serie de condiciones, y dependiendo de cuales cumpla tendrá un valor u otro. Se empieza con este apartado ya que este método de obtener vl es común para las tres instrucciones. En la Figura 34 se ven las condiciones que tiene que cumplir:

1. `vl = AVL` if `AVL ≤ VLMAX`
2. `ceil(AVL / 2) ≤ vl ≤ VLMAX` if `AVL < (2 * VLMAX)`
3. `vl = VLMAX` if `AVL ≥ (2 * VLMAX)`
4. Deterministic on any given implementation for same input AVL and VLMAX values
5. These specific properties follow from the prior rules:
    a. `vl = 0` if `AVL = 0`
    b. `vl > 0` if `AVL > 0`
    c. `vl ≤ VLMAX`
    d. `vl ≤ AVL`

Figura 34. Condiciones para determinar el valor de vl. [19]

VMAX se define como LMUL*VLEN/SEW. Teniendo en cuenta lo anterior no resulta muy difícil implementar la lógica necesaria. Es por ello que se desarrolla el Código 12.

```
vsew  <= csr.vtype(5 downto 3);
vlmul <= csr.vtype(2 downto 0);
vsew_o <= vsew;

case vsew is
   when sew_8  => f_sew <= 8;
   when sew_16 => f_sew <= 16;
   when sew_32 => f_sew <= 32;
   when sew_64 => f_sew <= 64;
   when others => f_sew <= 8;
end case;
case vlmul is
   when lmul_1_8  => f_lmul <= 1/8;
   when lmul_1_4  => f_lmul <= 1/4;
   when lmul_1_2  => f_lmul <= 1/2;
   when lmul_1    => f_lmul <= 1;
   when lmul_2    => f_lmul <= 2;
   when lmul_4    => f_lmul <= 4;
   when lmul_8    => f_lmul <= 8;
   when others    => f_lmul <= 1;
 end case;

vlmax <= f_lmul*VLEN/f_sew;

if (to_integer(unsigned(avl)) <= vlmax) then
  csr.vl <= avl(5 downto 0);
elsif (to_integer(unsigned(avl)) <= 2*vlmax) then
  if (to_integer(unsigned(csr.vl)) > vlmax) then
    csr.vl <= std_ulogic_vector(to_signed(vlmax, 6));
  elsif (to_integer(unsigned(csr.vl)) < (to_integer(unsigned(avl))/2)) then
    csr.vl <= std_ulogic_vector(to_signed(to_integer(unsigned(avl))/2, 6));
  end if;
elsif (to_integer(unsigned(avl)) >= 2*vlmax) then
   csr.vl <= std_ulogic_vector(to_signed(vlmax, 6));
end if;
```

Código 12. Condiciones para obtener el valor de vl implementado.

La función que desarrolla el Código 12 es muy sencilla. En primer lugar carga desde el CSR vtype los valores de sew y lmul, que los necesitaremos para calcular VLMAX. Una vez obtenido se compara con el valor de avl, y en función del rango en el que se encuentre se tendrá un valor de vl.

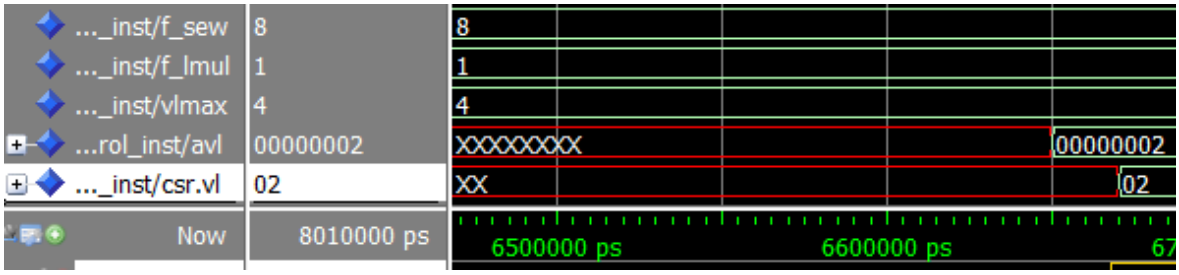
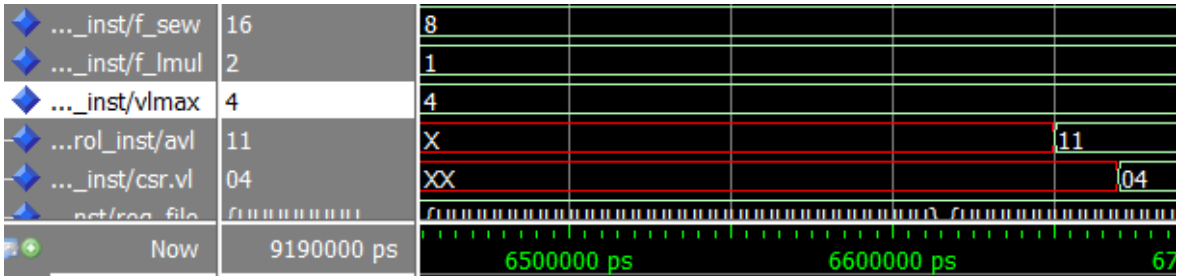


Figura 35. Simulación para comprobar el funcionamiento de vl.



Como se ve en la Figura 35 avl es 2, y como avl es menor que vlmax entonces se tiene que vl tiene que ser igual que avl.

En las simulaciones de las instrucciones no se entrará a valorar más el valor de vl ya que es siempre lo mismo.

5.2.2.2 Diseño de la instrucción vsetvl

Se comienza con la instrucción vsetvl, que tal y como se puede ver en la Figura 36 y la Figura 37 utiliza dos registros escalares para poder codificar los CSR vtype y vl.

```
vsetvl  rd, rs1, rs2      # rd = new vl, rs1 = AVL, rs2 = new vtype value
```

Figura 36. Instrucción vsetvl en ensamblador. [19]

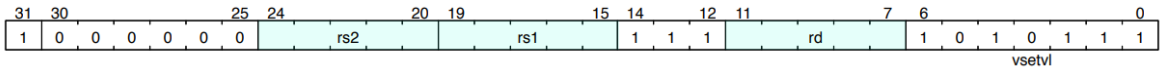


Figura 37. Formato de la instrucción vsetvl. [19]

En este caso el registro con dirección rs2 proporcionará el nuevo valor de vtype, el registro con dirección rd proporciona el nuevo posible valor de vl. Por último, el registro con

dirección rs1 nos proporciona el valor de una nueva variable llamada AVL (Application Vector Length), la cual nos servirá para determinar el valor final de vl.

Lo primero es ver donde se introducirá la lógica necesaria, llegando a que el mejor sitio es el archivo neorv32_cpu_control.vhd, y concretamente se introducirá en el mismo proceso que se utiliza para escribir los demás CSR. Se eligió este lugar en la jerarquía ya que contiene la mayoría de las señales necesarias, y las que no son fácilmente accesibles, y porque de esta manera se tienen todos los CSR de la extensión vectorial en un mismo lugar recogidos.

Una vez escogido el lugar se desarrolla el Código 13:

```
-- Control and Status Registers - Vector Extension ----------------------------
-- -----------------------------------------------------------------------------

  csr_vector: process(rstn_i, clk_i)
    begin
      -- Vector CSRs
      if (rstn_i = '0') then
        csr.vstart <= (others => def_rst_val_c);
        csr.vxsat  <= (others => def_rst_val_c);
        csr.vxrm   <= (others => def_rst_val_c);
        csr.vcsr   <= (others => def_rst_val_c);

      elsif rising_edge(clk_i) then

       […]
      -- [m]vtype --
        if (CPU_EXTENSION_RISCV_Zve32x = true) then
          if(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) = vsetvl_c)
then -- vsetvl instruction enable
            if (csr.we = '1') then -- write access
              csr.vtype <= rs2_i;
              csr.vl <= rd_i(5 downto 0);
              avl <= rs1_i;
             end if;

       […]
```

Código 13. Instrucción vsetvl implementada.

Analizando el Código 13 se ve que lo primero que tenemos es la condición por la cual esta instrucción solo es válida si la extensión está habilitada, que se comprueba de la misma manera que se viene haciendo con el resto de los CSR.

La siguiente condición comprueba que se trata de la instrucción vsetvl_c, para ello se añadieron dos nuevos bits a la señal ctrl_i. La señal ctrl_i es una señal que utiliza la cpu para comunicarse entre sus distintos componentes, es por eso que se decidió añadir dos nuevos bits, de tal manera que se pudiese codificar las tres nuevas instrucciones. Para añadir estos dos nuevos bits lo primero es localizar la señal ctrl.

```
-- local signals --
signal ctrl       : std_ulogic_vector(ctrl_width_c-1 downto 0); -- main control bus
```

Figura 38. Señal ctrl (main control bus).

Esta señal la localizamos en el archivo neorv32_cpu.vhd, y tal y como se ve en la Figura 38 su longitud viene definida por la constante ctrl_width_c. El siguiente paso es localizar esta constante y añadirle los dos nuevos bits. La gran mayoría de las constantes se encuentran en el archivo neorv32_package.vhd, por lo que se empieza buscando en él. Efectivamente se encuentra en este archivo, por lo que tal y como se ve en la Figura 39 pasa de vale 75 a 77.

```
-- control bus size --
constant ctrl_width_c          : natural := 77; -- control bus size
```

Figura 39. Constante ctrl_width_c modificada

Ya se tienen dos nuevos bits para codificar las señales, y para facilitar su uso se declararán dos constantes que identifiquen las posiciones. De esta manera se crean las constantes ctrl_cc_vtype_en0_c y ctrl_cc_vtype_en1_c tal y como se muestra en la

```
constant ctrl_cc_vtype_en0_c  : natural := 76; -- vtype en0
constant ctrl_cc_vtype_en1_c  : natural := 77; -- vtype en1
```

Figura 40. Constantes ctrl_cc_vtype_en0_c y ctrl_cc_vtype_en1_c.

Una vez ya se tienen los bits necesarios se necesita crear la lógica para que identifique correctamente las instrucciones. Es por ello que se vuelve al archivo neorv32_cpu_control.vhd. De nuevo se busca el lugar indicado en el que diseñar, y se

decide implementar la lógica dentro del case llamado CSR_ACCESS, case al que solo se accede cuando detecta que es una instrucción de lectura o escritura de los CSR.

Con base en la Figura 29 se deduce que tenemos una instrucción de tipo R, por lo que tendremos que analizar que señales son necesarias. Tras una investigación por varios archivos se descubrió que los bits correspondientes a rs2, rs1, funct3, rd y opcode, es decir del 24 al 0 estan codificados dentro de la señal ctrl, tal y como se puede ver en la Figura 41, la Figura 42 y la Figura 43.

```
constant ctrl_rf_rs1_adr0_c   : natural :=  1; -- source register 1 address bit 0
constant ctrl_rf_rs1_adr1_c   : natural :=  2; -- source register 1 address bit 1
constant ctrl_rf_rs1_adr2_c   : natural :=  3; -- source register 1 address bit 2
constant ctrl_rf_rs1_adr3_c   : natural :=  4; -- source register 1 address bit 3
constant ctrl_rf_rs1_adr4_c   : natural :=  5; -- source register 1 address bit 4
constant ctrl_rf_rs2_adr0_c   : natural :=  6; -- source register 2 address bit 0
constant ctrl_rf_rs2_adr1_c   : natural :=  7; -- source register 2 address bit 1
constant ctrl_rf_rs2_adr2_c   : natural :=  8; -- source register 2 address bit 2
constant ctrl_rf_rs2_adr3_c   : natural :=  9; -- source register 2 address bit 3
constant ctrl_rf_rs2_adr4_c   : natural := 10; -- source register 2 address bit 4
constant ctrl_rf_rd_adr0_c    : natural := 11; -- destination register address bit 0
constant ctrl_rf_rd_adr1_c    : natural := 12; -- destination register address bit 1
constant ctrl_rf_rd_adr2_c    : natural := 13; -- destination register address bit 2
constant ctrl_rf_rd_adr3_c    : natural := 14; -- destination register address bit 3
constant ctrl_rf_rd_adr4_c    : natural := 15; -- destination register address bit 4
```

Figura 41. Bits de rs1,rs2 y rd en la señal ctrl.

```
-- instruction's control blocks (used by cpu co-processors) --
constant ctrl_ir_funct3_0_c   : natural := 48; -- funct3 bit 0
constant ctrl_ir_funct3_1_c   : natural := 49; -- funct3 bit 1
constant ctrl_ir_funct3_2_c   : natural := 50; -- funct3 bit 2
```

Figura 42. Bits de funct3 en la señal ctrl.

```
constant ctrl_ir_opcode7_0_c  : natural := 63; -- opcode7 bit 0
constant ctrl_ir_opcode7_1_c  : natural := 64; -- opcode7 bit 1
constant ctrl_ir_opcode7_2_c  : natural := 65; -- opcode7 bit 2
constant ctrl_ir_opcode7_3_c  : natural := 66; -- opcode7 bit 3
constant ctrl_ir_opcode7_4_c  : natural := 67; -- opcode7 bit 4
constant ctrl_ir_opcode7_5_c  : natural := 68; -- opcode7 bit 5
constant ctrl_ir_opcode7_6_c  : natural := 69; -- opcode7 bit 6
```

Figura 43. Bits de opcode7 en la señal ctrl.

Sin embargo los bits correspondientes a funct7, del 31 al 25, no están en la señal ctrl, por lo que se buscó a que señal pertenecían, localizándolos en la señal execute_engine.i_reg. En la se pueden ver los bits correspondientes a funct7 en dicha señal.

```
constant instr_funct7_lsb_c  : natural := 25; -- funct7 bit 0
constant instr_funct7_msb_c  : natural := 31; -- funct7 bit 6
```

Figura 44. Bits de funct7 en la señal execute_engine.i_reg.

Una vez entendido como se transmite la instrucción por la CPU, tan solo falta por implementar las condiciones que caracterizan a cada una de las instrucciones. De esta manera se desarrolla el Código 14.

```
when CSR_ACCESS => -- read & write status and control register (CSR) - no
read/write if illegal instruction
-- ------------------------------------------------------------
-- CSR write access -

[…]

       if (ctrl_i(ctrl_ir_opcode7_6_c downto ctrl_ir_opcode7_0_c) =
"1010111")then
          if (ctrl_i(ctrl_ir_funct3_2_c downto ctrl_ir_funct3_0_c) = "111") then
            if (execute_engine.i_reg(instr_funct7_msb_c downto
instr_funct7_lsb_c) = "1000000") then
              ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <=
vsetvl_c;

             else
              ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <= (others
=> '-');
            end if;
          else
            ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <= (others
=> '-');
          end if;
        else
          ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <= (others =>
'-');
        end if;
```

Código 14. Código para comprobar si es la instrucción vsetvl.

En la primera condición se comprueba que el opcode7 es el de la instrucción vsetvl, en la siguiente se comprueba el funct3. Estas dos condiciones van a ser iguales para las tres instrucciones, ya que todas comparten los mismos valores en esos campos. Por último, se

comprueba el campo funct7. Si se cumplen estas tres condiciones entonces los bits correspondientes al enable implementado son puestos con la codificación vsetvl_c.

```
-- Vtype enable
constant vsetvli_c  : std_ulogic_vector(1 downto 0) := "01"; -- vsetvli enable
constant vsetivli_c : std_ulogic_vector(1 downto 0) := "10"; -- vsetivli enable
constant vsetvl_c   : std_ulogic_vector(1 downto 0) := "11"; -- vsetvl enable
```

Figura 45. Codificación de los enable de las instrucciones vsetvl, vsetvli y vsetivli.

Una vez ya se ha comprobado que se está ejecutando la instrucción vsetvl, volviendo al Código 13, el siguiente paso es que se permita la escritura en los CSR mediante crs.we. Una vez ya se han completado las comprobaciones es el momento de escribir en vtype. Para ello es necesario modificar el archivo neorv32_cpu_control.vhd, ya que no cuenta con rs1, rs2 y rd como entradas. Para ello simplemente hay que modificar el apartado port, dentro de la entidad, al inicio del archivo, el port en el component y conectarlo con el archivo noerv32_cpu_regfile.vhd mediante el port map. Todos estos cambios se pueden ver en el Código 15, el Código 16 y el Código 17.

```
entity neorv32_cpu_control is
  generic (
      […]
  );
  port (
    […]

    rs1_i        : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source 1
    rs2_i        : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source 2
    rd_i         : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source d

    […]
  );
end neorv32_cpu_control;
```

Código 15. Entity de neorv32_cpu_control modificado.

```
-- Component: CPU Control ------------------------------------------------------
-- ------------------------------------------------------------------------------
  component neorv32_cpu_control
    generic (
        […]
    );
    port (
      […]

      rs1_i      : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source 1
      rs2_i      : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source 2
      rd_i       : in  std_ulogic_vector(data_width_c-1 downto 0); -- rf source d

    […]
    );
  end component;
```

Código 16. Component de neorv32_cpu_control modificado.

```
-- Control Unit ----------------------------------------------------------------
-- ------------------------------------------------------------------------------
  neorv32_cpu_control_inst: neorv32_cpu_control
  generic map (
      […]
  )
  port map (
    […]

    rs1_i         => rs1,         -- rf source 1
    rs2_i         => rs2,         -- rf source 2
    rd_i          => rd,          -- rf d

    […]
  );
```

Código 17. Port map de neorv32_cpu_control conectado con neorv32_cpu_regfile.

Por último, se modificó el archivo neorv32_cpu_regfile para que una de sus salidas fuese el registro con dirección rd, para poder usarlo para la nueva instrucción. Simplemente se añadió la nueva señal rd tal y como se hace con las señales rs1 y rs2. En el se puede ver la implementación mencionada.

```
-- Register File Access ------------------------------------------------------
-- ---------------------------------------------------------------------------
  rf_access: process(clk_i)
  begin
    if rising_edge(clk_i) then -- sync read and write
      if (CPU_EXTENSION_RISCV_E = false) then -- normal register file with 32
entries
        if (ctrl_i(ctrl_rf_wb_en_c) = '1') then
          reg_file(to_integer(unsigned(opa_addr(4 downto 0)))) <= rf_wdata;
        end if;
        rs1 <= reg_file(to_integer(unsigned(opa_addr(4 downto 0))));
        rs2 <= reg_file(to_integer(unsigned(opb_addr(4 downto 0))));
        rd  <= reg_file(to_integer(unsigned(dst_addr(4 downto 0))));
      else -- embedded register file with 16 entries
        if (ctrl_i(ctrl_rf_wb_en_c) = '1') then
          reg_file_emb(to_integer(unsigned(opa_addr(3 downto 0)))) <= rf_wdata;
        end if;
        rs1 <= reg_file_emb(to_integer(unsigned(opa_addr(3 downto 0))));
        rs2 <= reg_file_emb(to_integer(unsigned(opb_addr(3 downto 0))));
      end if;
    end if;
  end process rf_access;
-- data output --
  rs1_o <= rs1;
  rs2_o <= rs2;
  rd_o  <= rd;
```

Código 18. Archivo noerv32_cpu_regfile modificado para sacar la señal rd.

5.2.2.3 Simulación de la instrucción vsetvl

Para comprobar el buen funcionamiento de la instrucción vsetvl se desarrollará un sencillo programa en ensamblador que escriba en el CSR vtype. Como es una instrucción vectorial se tendrá que ensamblar a mano.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li a0, 1

  li a1, 2

  li a2, 3

  vsetvl a2, a1, a0 /*Instruccion vectorial*/

.end
```

Código 19. Programa para probar la instrucción vsetvl.

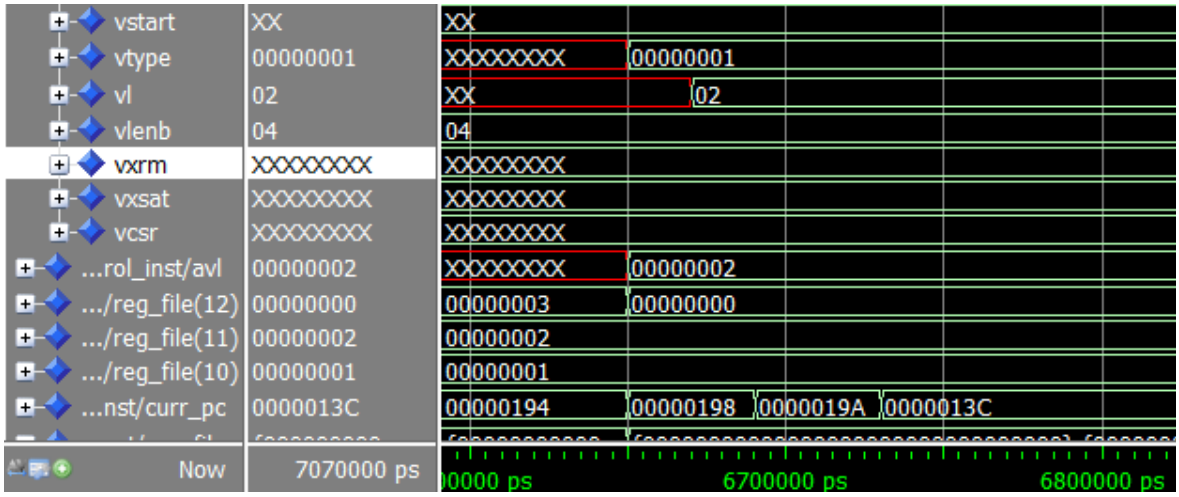


Figura 46. Simulación de la instrucción vsetvl.

Como se ve en la Figura 46 el vtype se carga con el registro a0, número 10, avl con el registro a1, número 11, y por último tenemos que vl también se carga con el valor correspondiente.

5.2.2.4 Diseño de la instrucción vsetvli

En la Figura 47 y la Figura 48 se puede ver como la instrucción vsetvli utiliza dos registros para cargar los datos de avl y vl, mientras que utiliza un inmediato para cargar el valor de vtype, que es en lo que se diferencia con la instrucción anterior.

```
vsetvli rd, rs1, vtypei   # rd = new vl, rs1 = AVL, vtypei = new vtype setting
```

Figura 47. Instrucción vsetvli en ensamblador. [19]

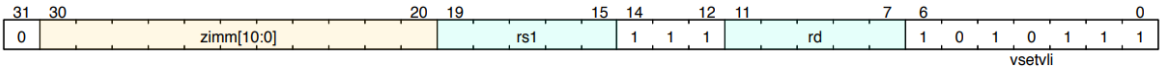
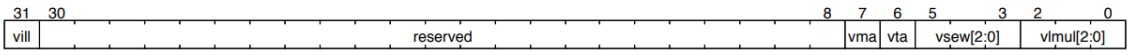


Figura 48. Formato de la instrucción vsetvli. [19]

Tal y como se puede ver en la Figura 49 tan solo hay 8 bits de escritura en vtype, pero en el inmediato zimm se tienen un vector de 11 bits, por lo que hay que decidir qué hacer con los bits restantes. Como lo más normal es que se introduzca el inmediato como un número lo más lógico sería quedarse con los bits más representativos, por lo que los bits 30, 29 y 28 se descartaran a la hora de escribir en vtype.



Note This diagram shows the layout for RV32 systems, whereas in general `vill` should be at bit XLEN-1.

Table 2. `vtype` register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5:3	vsew[2:0]	Selected element width (SEW) setting
2:0	vlmul[2:0]	Vector register group multiplier (LMUL) setting

Figura 49. Formato y codificación del CSR vtype. [19]

Para la implementación de la instrucción se usará la señal ctrl de la misma manera que se hizo para la señal vsetvl, y es por ello que se desarrolla el Código 20.

```
[…]

if (ctrl_i(ctrl_ir_opcode7_6_c downto ctrl_ir_opcode7_0_c) = "1010111")then
 if (ctrl_i(ctrl_ir_funct3_2_c downto ctrl_ir_funct3_0_c) = "111") then

  […]

   elsif (execute_engine.i_reg(instr_funct7_msb_c) = '0') then
      ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <= vsetvli_c;
  […]
```

Código 20. Código para comprobar si es la instrucción vsetvl.

El Código 20 se implementa justo al terminar la condición "*if (execute_engine.i_reg(instr_funct7_msb_c downto instr_funct7_lsb_c) = "1000000") then",* en el Código 14. Con esto se consigue identificar la instrucción vsetvli, con base en la Figura 48. Una vez es identificada se escribe en los bits correspondientes a la señal del enable de vtype la codificación de vsetvli, para luego poder utilizarla.

El siguiente paso es implementar las funciones que realiza vsetvli. Estas se implementarán justo donde termina el Código 13, desarrollándose de esta manera el Código 21.

```
  elsif(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) = vsetvli_c) then
-- vsetvli instruction enable
     if (csr.we = '1') then -- write access
     csr.vtype(31 downto 8) <= (others => '0');
     csr.vtype(7 downto 5) <= execute_engine.i_reg(instr_funct7_msb_c-4 downto
instr_funct7_lsb_c);
     csr.vtype(4 downto 0) <= ctrl_i(ctrl_rf_rs2_adr4_c downto
ctrl_rf_rs2_adr0_c);
     csr.vl <= rd_i(5 downto 0);
     avl <= rs1_i;
  end if;
```

Código 21. Instrucción vsetvl implementada.

Tal y como se hizo con vsetvl, lo primero es comprobar que la codificación del enable de vtype sea la correspondiente a vsetvli, y luego que se tenga permiso para escribir en el CSR. Una vez se cumplen estas condiciones se pasa a escribir en vtype, avl y vl. Ni avl, ni vl cambian con respecto a vsetvl, ya que se aprovecha que para esa instrucción introdujeron en el archivo rs1 y rd entre otras.

En cambio, vtype ahora se carga con un inmediato, por lo que, con base en Figura 29, se deduce que se tiene una instrucción de tipo I. Para cargar los datos, primero se tiene que averiguar cómo se transmiten estos por la CPU, lo cual se descubrió en la realización de vsetvl como ya se ha explicado. Es por ello que vtype se carga en tres partes, la primera consiste en poner a 0 todos los bits reservados más el vill. En la segunda se carga de la señal execute_engine.i_reg los datos correspondientes a los bits 7 a 5 de vtype. Y, por último, se cargan los datos correspondientes a los bits 5 a 0.

5.2.2.5 Simulación de la instrucción vsetvli

Para comprobar el buen funcionamiento de la instrucción vsetvli se realizará un sencillo programa en ensamblador. Se recuerda que se utiliza una instrucción vectorial, por lo que esta deberá ser ensamblada a mano.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li a1, 2

  li a2, 3

  vsetvli a2, a1, 15 /*Instruccion vectorial*/

.end
```

Código 22. Programa para probar la instrucción vsetvli.

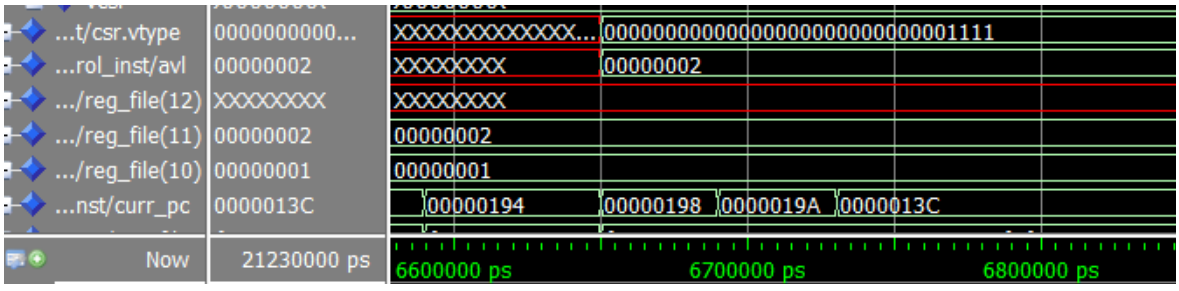


Figura 50. Simulación de la instrucción vsetvli.

En la Figura 50 se puede ver como la instrucción vsetvli carga el número 15, 1111 en binario y avl con el valor del registro a2, registro número 11.

5.2.2.6 Diseño de la instrucción vsetivli

En la Figura 51 y la Figura 52 se puede ver como ahora tan solo vl se carga desde un registro, mientras que tanto vtype y avl utilizan inmediatos.

```
vsetivli rd, uimm, vtypei # rd = new vl, uimm = AVL, vtypei = new vtype setting
```

Figura 51. Instrucción vsetivli en ensamblador. [19]

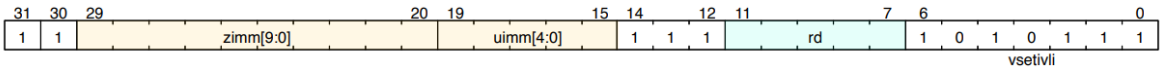


Figura 52. Formato de la instrucción vsetili. [19]

Con el inmediato zimm se vuelve a tener el mismo problema que en la instrucción vsetvli, ya que tiene mas bits que en vtype se pueden escribir, es por ello que se no se utilizaran los bits no necesarios de la misma manera que en vsetvli. Para el caso de avl no hay problema, por lo que se cargara uimm en las primeras posiciones de avl y el resto se llenarán con 0.

Lo primero es hacer que se identifique la instrucción vsetivli, y para ello se volverá a utilizar la señal ctrl y execute_engine.i_reg. De la misma manera que se ha realizado para las otras instrucciones se desarrolla el Código 23.

```
[…]

if (ctrl_i(ctrl_ir_opcode7_6_c downto ctrl_ir_opcode7_0_c) = "1010111")then
 if (ctrl_i(ctrl_ir_funct3_2_c downto ctrl_ir_funct3_0_c) = "111") then

  […]

   elsif (execute_engine.i_reg(instr_funct7_msb_c downto instr_funct7_msb_c-1) =
"11") then
      ctrl_nxt(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) <= vsetivli_c;
[…]
```

Código 23. Código para comprobar si es la instrucción vsetivli.

Con el Código 23 se consigue identificar cuando se está ejecutando la instrucción vsetivli, y cuando se identifica se escribe en la señal ctrl en los bits correspondientes al enable de vtype la codificación correspondiente a vsetivli.

Una vez identificada el siguiente paso es implementar las funcionalidades que realiza esta instrucción, es por ello que se desarrollará el Código 24 y se implementará al terminar el Código 21.

```
  elsif(ctrl_i(ctrl_cc_vtype_en1_c downto ctrl_cc_vtype_en0_c) = vsetivli_c) then
-- vsetivli instruction enable
    if (csr.we = '1') then -- write access
      csr.vtype(31 downto 8) <= (others => '0');
      csr.vtype(7 downto 5) <= execute_engine.i_reg(instr_funct7_msb_c-4 downto
instr_funct7_lsb_c);
      csr.vtype(4 downto 0) <= ctrl_i(ctrl_rf_rs2_adr4_c downto
ctrl_rf_rs2_adr0_c);
      csr.vl <= rd_i(5 downto 0);
      avl(4 downto 0) <= ctrl_i(ctrl_rf_rs1_adr4_c downto ctrl_rf_rs1_adr0_c);
      avl(31 downto 5) <= (others => '0');
    end if;
  end if;
```

Código 24. Instrucción vsetivli implementada.

Analizando el Código 24 vemos que lo primero que se comprueba es que efectivamente se trate de la instrucción vsetivli. Lo siguiente es comprobar que se tiene acceso para escribir en los CSR. La carga de vl no cambia con respecto a las instrucciones anteriores. Vtype se carga de la misma manera que en vsetvl. Por último, se tiene avl, que ahora se carga con un inmediato, es por ello que se le pasa la señal ctrl en los bits correspondientes a rs1, y el resto de los bits simplemente se cargan a 1.

5.2.2.7 Simulación de la instrucción vsetivli

Con el fin de comprobar que vsetivli es capaz de escribir en vtype se desarrollará un programa sencillo en ensamblador. Es por ello que se crea el Código 25.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li a1, 2

  li a2, 3

  vsetivli a2, 11, 9 /*Instruccion vectorial*/

.end
```

Código 25. Programa para probar la instrucción vsetivli.

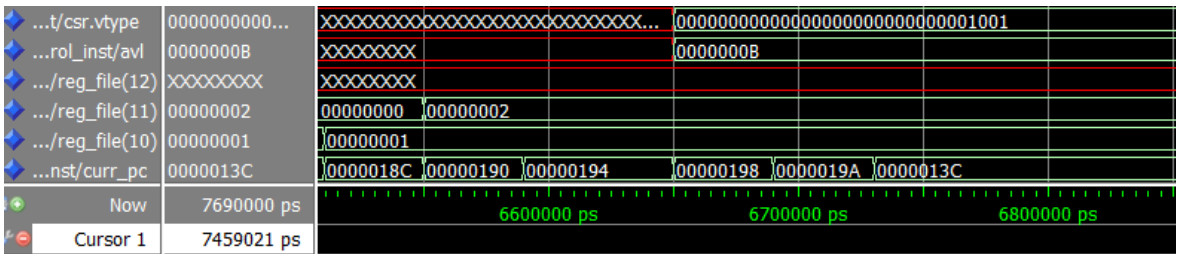


Figura 53. Simulación de la instrucción vsetivli.

Tal y como se observa en la Figura 53 se tiene que el CSR vtype se carga con el valor 9, 1001 en binario, y avl se carga con el valor hexadecimal B, 11 en decimal. Los datos se cargan tal y como se espera, por lo que la instrucción funciona correctamente.

5.2.3 DISEÑO Y SIMULACIÓN DEL BANCO DE REGISTROS VECTORIALES

5.2.3.1 Diseño del banco registros vectoriales

Se tiene que diseñar un nuevo banco de registros al que tenga acceso la extensión, en el que se almacenaran los vectores con los que trabaje. Se especifica que se tienen que crear 32 nuevos registros de una longitud fija que se denominara VLEN, es decir, el número de bits que tendrá cada registro. La longitud mínima será de 32 bits, pero es un parámetro completamente modificable, de manera que se pueda adaptar mejor al uso que se le dé.

En cuanto a su diseño no es muy complicado, ya que se trata de un registro normal, en el que para escribir en uno de sus registros tendremos que pasarle los datos a escribir más la dirección en donde escribirlos, y estos se escribirán siempre que la señal de enable lo permita. En cuanto a la lectura de datos, el registro consta de dos salidas en paralelo, compuestas cada una por un multiplexor al que se le pasa la dirección del dato que se desea leer, y este lo saca por la salida del registro.

Una vez definidas sus características básicas se procede a describirlo en VHDL, teniendo en cuenta que este no será su diseño final, ya que otras funcionalidades de la extensión podrán añadir pequeñas modificaciones, pero no modificarán su funcionamiento.

5.2.3.2 Simulación del registro de vectores

Seguidamente se realizará un testbench, el cual consistirá en cargar datos en todos los registros menos en el registro 0, pasado un tiempo se leerán algunos de los datos y se recibirá una entrada simulando la salida de la ALU, que escribirá un dato diferente en un registro.

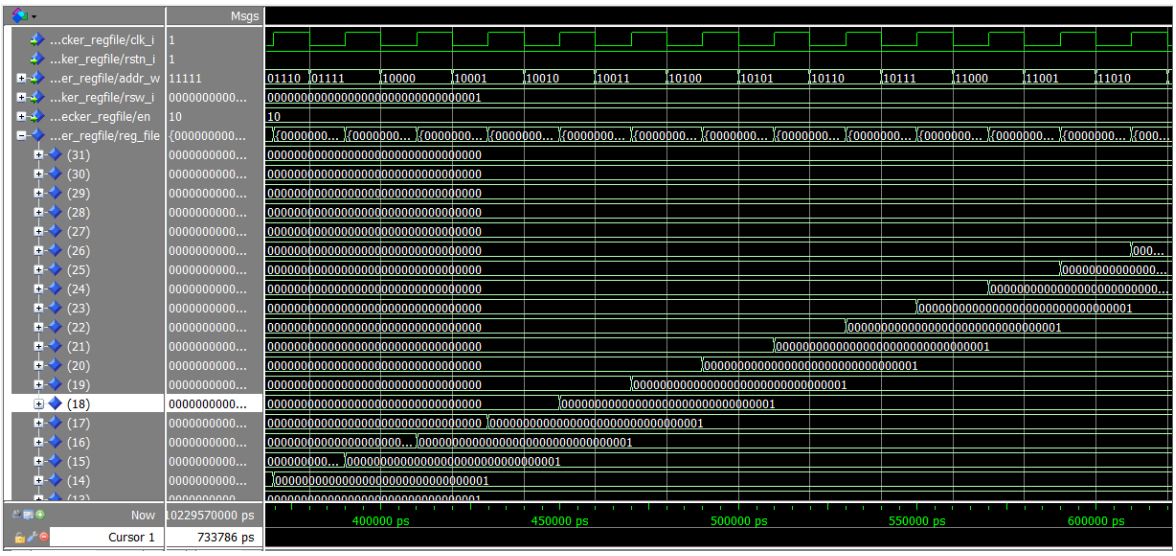


Figura 54. Salida del testbench del archivo neorv32_cpu_vector_regfile.vhd, parte de la carga de datos en los registros.

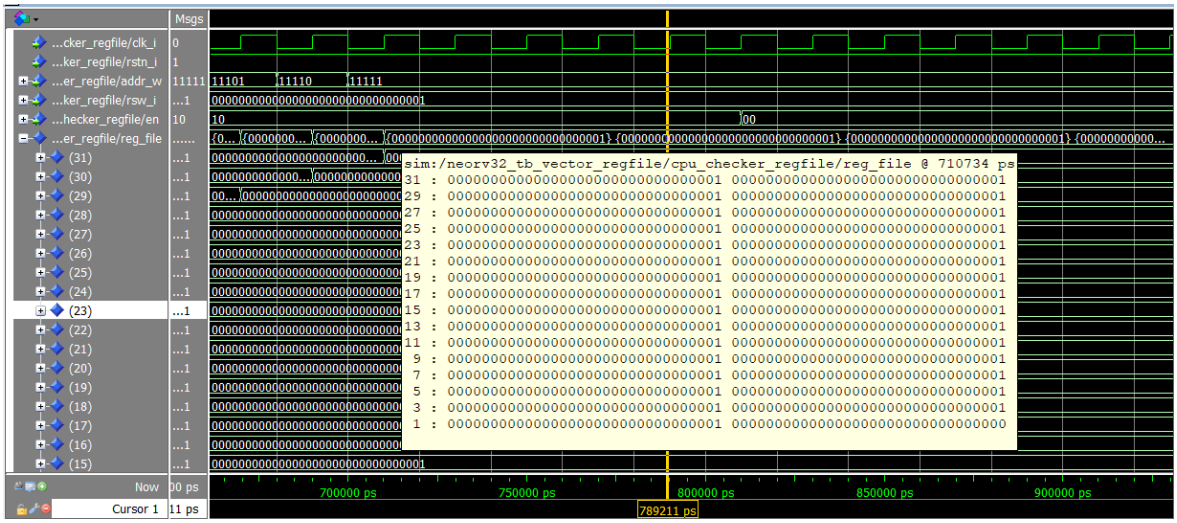


Figura 55. Salida del testbench del archivo neorv32_cpu_vector_regfile.vhd, registro completamente cargado con datos menos el registro 0.

Tal y como se puede ver en la Figura 54 a cada ciclo de reloj la dirección de escritura (addr_w) y por tanto como la señal de los datos de entrada (rsw_i) no cambia a cada ciclo de reloj se escribe un 1 en todos los registros menos en el 0. En la Figura 55 podemos ver como se ha cargado perfectamente todo el registro menos el registro 0.

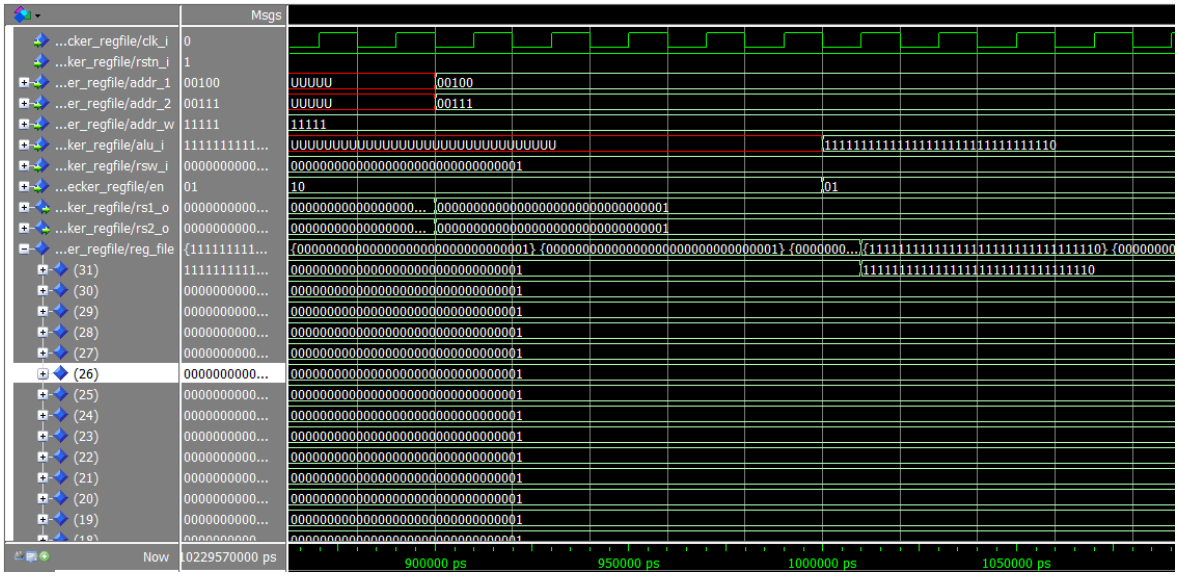


Figura 56. Salida del testbench del archivo neorv32_cpu_vector_regfile.vhd, lectura y escritura del registro.

Para ver la siguiente fase hay que fijarse en la Figura 56, en la que se observa como a los 900ns se leen dos datos del registro, al activarse las señales addr_1 y addr_2, por lo que en las salidas rs1_o y rs2_o tenemos los datos que contenían esas direcciones. Tras pasar un tiempo, en el que se supone que otro bloque opera con esos datos, se tiene que a un 1ms se recibe la salida de la ALU (alu_i) y también se activa el eneable, por lo que en el siguiente ciclo de reloj se escribe en el registro 31, ya que es dirección de escritura (addr_w).

5.2.4 DISEÑO Y SIMULACIÓN DE LA INSTRUCCIÓN VLEX.V

Como ya se ha visto, se tiene un banco de registros nuevo y completamente funcional, por lo que hay que buscar una manera de cargar datos en él. Es por ello que aparece la instrucción vlex.v, la cual nos permitirá cargar datos desde la memoria ram en el registro de vectores, para su posterior procesamiento.

5.2.4.1 Diseño de la instrucción vlex.v

Antes de poder empezar a diseñar los circuitos digitales que nos permitan utilizar la nueva instrucción, primero hay que aprender cómo se hace la carga de datos desde la memoria RAM hasta el registro escalar, ya que este ya está implementado y nos servirá. De esta manera se empieza por aprender cómo se hace en ensamblador para cargar y descargar datos de la memoria RAM, utilizando para ello el libro Apuntes de la asignatura Sistemas Digitales II [23].

En él se describe como usar la pila para poder cargar datos y descargarlo de la memoria RAM. Para ello siempre que se quiera guardar un dato nuevo hay que hacer hueco en la pila, este hueco siempre tiene que ser múltiplo de 4. Una vez hecho el hueco se almacena el valor de un registro en la pila. Ahora que se tiene guardado el dato en la pila se puede sobrescribir el registro sin problema y luego poder recuperar su valor original, ya que este queda almacenado en la RAM.

Una vez almacenado se puede acceder a este dato para almacenarlo en cualquier otro registro, y es aquí donde nuestra instrucción deberá actuar, ya que se utilizará la RAM para cargar datos en el registro de vectores. Una vez se ha terminado de usar la pila hay que restaurar su valor al inicial.

Con una idea más clara de cómo se manejan datos de la RAM se desarrolla el programa del Código 26.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li   a0, 14
  li   a1, 255
  li   sp, 0x80001000
  addi sp, sp, -8
  sw   a1, 4(sp)
  sw   a0, 0(sp)
  li   a1, 0
  li   a0, 0
  lw   a7, 4(sp)
  lw   a0, 0(sp)
  addi sp, sp, 8

.end
```

Código 26. Programa para cargar y descargar datos de la memoria RAM.

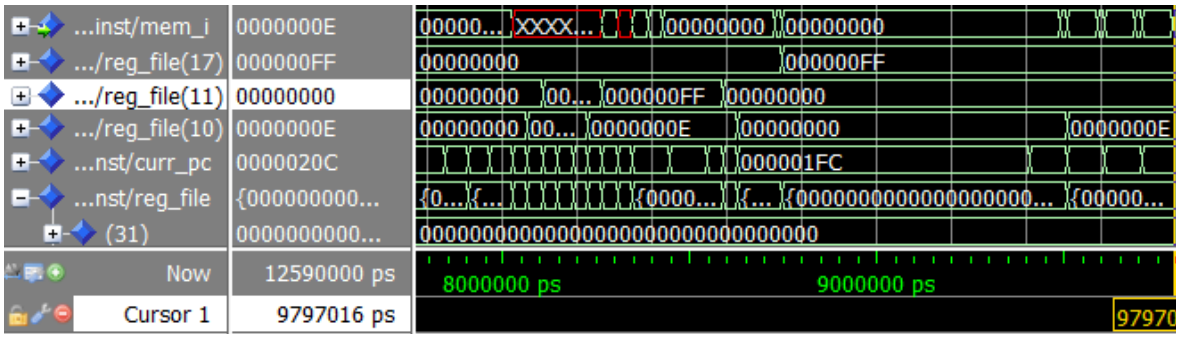


Figura 57. Simulación del Código 26.

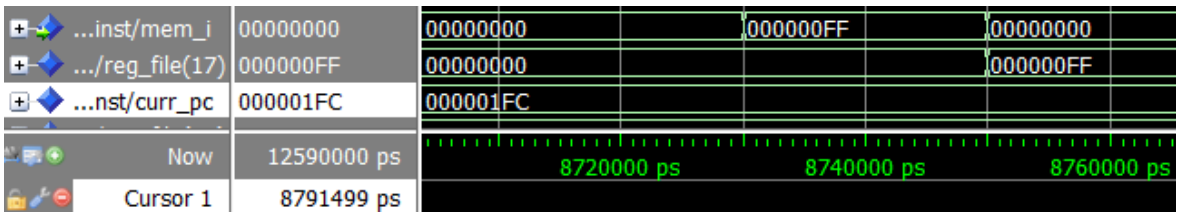


Figura 58. Señal mem_i guardando 0xFF ampliada.

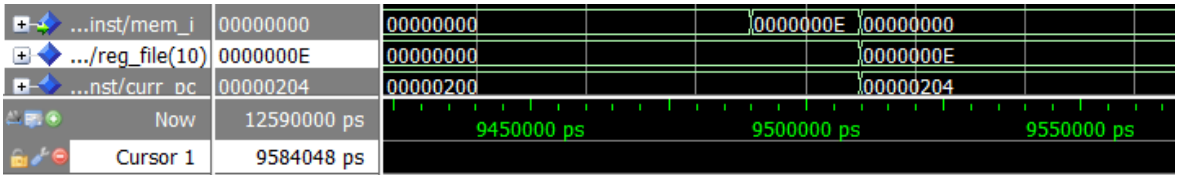


Figura 59. Señal mem_i guardando 0xE ampliada.

Analizando la simulación, se ve en la Figura 57 como en un primer momento los registros a0 y a1 son puestos con los valores 14 y 255 respectivamente. Las siguientes instrucciones consisten en almacenar dichos datos en la memoria RAM para luego poder recuperarlos. Una vez almacenados tanto a0 como a1 son puestos a 0. Ahora con el fin de comprobar si realmente está funcionando se cargarán en a0 14 y en a7 255, y tal y como se puede ver en la Figura 57 así sucede.

Sin embargo, no se aprecia claramente como se reciben esos datos, es por ello que en la Figura 58 y en la Figura 59 sí que se ve más claramente como es la señal mem_i la encargada de traer todos esos datos desde la memoria RAM para luego cargarlos en los registros pertinentes.

Una vez que ya se sabe cómo realiza la operación de carga de datos desde la memoria lo que hay que hacer es seguir esa señal mem_i hasta su origen, para poder ver cuáles son las condiciones bajo las cuales transmite datos. Una vez las encontremos tan solo se tendrá que añadir las condiciones necesarias para que también funcione con las instrucciones de carga vectorial y almacene los datos en el banco de registros vectorial.

Cuando se detecte la instrucción se tendrá que transmitir una señal para que el banco de registros vectorial permita su escritura, de esta manera se añade un nuevo bit a la señal ctrl, de la misma manera que se ha hecho en anteriores ocasiones. En la Figura 60 se puede ver dicho bit.

```
constant ctrl_v_rf_wb_en_c     : natural := 82; -- write back enable vector rf
```

Figura 60. Bit añadido a la señal ctrl.

Siguiendo la señal mem_i hasta su origen se llega hasta el archivo neorv32_cpu_control.vhd, archivo en el cual se implementará la lógica necesaria para realizar la carga de datos desde la RAM.

Para ello se busca entre los estados posibles de la CPU, y se localiza dentro del estado execute la condición que permite el acceso a la memoria. Es en esta condición donde tenemos que añadir las nuestras para tener dicho acceso.

Para ello se crea una constante que contenga el valor del opcode de la instrucción de carga vectorial, tal y como se ve en la Figura 61.

```
constant opcode_v_load_c : std_ulogic_vector(6 downto 0) := "0000111"; -- vector load
```

Figura 61. Opcode de la instrucción vlex.v.

Ahora simplemente hay que añadir dicho opcode a la condición, para que cada vez que se utilice la instrucción se tenga acceso a la memoria.

```
-- state machine --
    case execute_engine.state is

    […]

      when EXECUTE =>
        opcode_v := execute_engine.i_reg(instr_opcode_msb_c downto
instr_opcode_lsb_c+2) & "11";

      […]

      when opcode_load_c | opcode_store_c | opcode_atomic_c | opcode_v_load_c =>
-- load/store / atomic memory access
         […]

         execute_engine.state_nxt     <= LOADSTORE_0;
```

Código 27. Condición de acceso a la memoria modificada.

En el Código 27 en la última línea se ve como se pasa al estado LOADSTORE_0, que es el primer estado que permite acceso a la memoria, pero hay otros dos más. Leyendo y entendiendo que hacían cada uno de ellos se llegó a la conclusión de que lo mejor era modificar el estado LOADSTORE_2, al cual se le añadiría el código necesario para

permitir el acceso a la memoria. No solamente hay que permitir que se pueda escribir en el banco de registros vectorial, sino que también hay que tener en cuenta que solo se puede escribir en uno de los dos, y es por ello que se tienen dos señales de enable distintas para cada uno de los registros.

```
when LOADSTORE_2 => -- wait for bus transaction to finish

   […]

            case opcode_v is
              when opcode_v_load_c =>
                ctrl_nxt(ctrl_v_rf_wb_en_c) <= '1'; -- valid vector RF write-back
              when others =>
                ctrl_nxt(ctrl_rf_wb_en_c) <= '1'; -- valid RF write-back
            end case;
    […]
```

Código 28. Selector para permitir el acceso al registro vectorial o escalar.

Una vez realizada todas estas modificaciones ya se debería de permitir cargar datos desde la memoria RAM al registro de vectores.

5.2.4.2 Simulación de la instrucción vlex.v

Para comprobar el funcionamiento de la instrucción se creará un programa en ensamblador que cargue en la memoria una serie de datos, los cuales serán escritos mediante la instrucción en el registro vectorial.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li   a1, 255
  li   sp, 0x80001000
  addi sp, sp, -4
  sw   a1, 4(sp)
  li   a1, 0
  vle8.v v16, (sp)
  vle8.v v17, (sp)
  addi sp, sp, 4

.end
```

Código 29. Programa para comprobar la instrucción vlex.v.

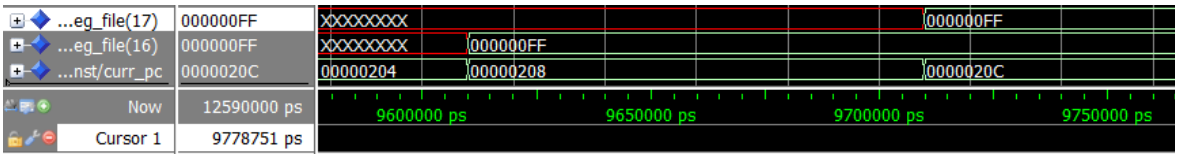


Figura 62. Simulación de la instrucción vlex.v

Lo primero que realiza el programa es poner el registro a1 a 255, y luego realiza el mismo proceso que se realizó anteriormente para cargar los datos en la memoria. Lo nuevo es lo que se puede ver en la Figura 62, en la que se aprecia el buen funcionamiento de la instrucción, ya que al finalizar el primer vle8.v nos carga el 255 en el registro v16, y lo mismo sucede para la siguiente instrucción, por lo que su funcionamiento es correcto.

5.2.5 DISEÑO Y SIMULACIÓN DE LA ALU VECTORIAL

Para poder realizar todos los cálculos pertinentes se necesita una ALU que sea capaz de trabajar con este tipo de datos. De esta manera se diseñará una ALU vectorial, con la que se ha de ser capaz de trabajar con vectores de distintos número y longitudes de elementos.

5.2.5.1 Diseño de la ALU vectorial

Lo primero es codificar las instrucciones que se van a usar en la ALU, que este caso serán la suma y la resta vectorial, teniendo en cuenta su formato de instrucción.

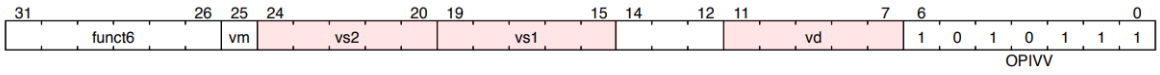


Figura 63. Formato de instrucciones vectoriales aritméticas. [19]

funct6				
000000	V	X	I	vadd
000001				
000010	V	X		vsub

Figura 64. Codificación de funct6 para la suma y la resta vectorial. [19]

Para comprobar si se está ejecutando una instrucción vectorial aritmética lo primero que se comprobará será el opcode, es decir, los bits de 6 a 0. Después lo siguiente es comprobar de que tipo de instrucción se trata, para ello se mirará en el apartado funct6, que según la codificación de la Figura 64 nos dirá si se trata de una suma o resta. De esta manera se implementa el Código 30.

```
-- Execute Engine FSM Comb ---------------------------------------------------
-- ---------------------------------------------------------------------------
execute_engine_fsm_comb: process(execute_engine, debug_ctrl, trap_ctrl,
decode_aux, fetch_engine, cmd_issue, csr, ctrl, alu_idone_i, bus_d_wait_i,
excl_state_i)
    variable opcode_v : std_ulogic_vector(6 downto 0);
  begin

  […]

  -- Vector extension (Zve32x) --
    if (execute_engine.i_reg(instr_funct7_msb_c downto instr_funct7_lsb_c+1) =
"000000") and
       (ctrl_i(ctrl_ir_opcode7_6_c downto ctrl_ir_opcode7_0_c) = "1010111") then
      ctrl_nxt(ctrl_v_alu_op1_c downto ctrl_v_alu_op0_c) <= v_alu_op_add_c;
    elsif (execute_engine.i_reg(instr_funct7_msb_c downto instr_funct7_lsb_c+1) =
"000010" and
        ctrl_i(ctrl_ir_opcode7_6_c downto ctrl_ir_opcode7_0_c) = "1010111") then
      ctrl_nxt(ctrl_v_alu_op1_c downto ctrl_v_alu_op0_c) <= v_alu_op_sub_c;
    else
      ctrl_nxt(ctrl_v_alu_op1_c downto ctrl_v_alu_op0_c) <= (others => '-');
    end if;

  […]
```

Código 30. Código para comprobar si es una instrucción aritmética y si es una de suma o de resta.

Para identificar las instrucciones lo primero es crear dos nuevos bits en la señal ctrl que nos servirán poder identificar, dentro de la ALU vectorial, si se tiene una instrucción de suma, de resta o ninguna de ambas. La forma de proceder es exactamente la misma que se venía haciendo, modificando la longitud del vector ctrl y creando dos nuevas constantes que nos sirvan para identificar los bits.

```
constant ctrl_v_alu_op0_c     : natural := 78; -- VALU operation select bit 0
constant ctrl_v_alu_op1_c     : natural := 79; -- VALU operation select bit 1
```

Figura 65. Bits para la codificación de las instrucciones de la ALU vectorial.

Una vez se comprueba y confirma que es una de las dos instrucciones se añade la codificación pertinente en los bits destinados para la ALU vectorial de la señal ctrl, para ello se declaran dos constantes, las de la Figura 66.

```
-- Vector ALU core -------------------------------------------------------------------------
constant v_alu_op_add_c     : std_ulogic_vector(1 downto 0) := "00"; -- valu_result <= A + B
constant v_alu_op_sub_c     : std_ulogic_vector(1 downto 0) := "01"; -- valu_result <= A - B
```

Figura 66. Constantes para la codificación de la suma y la resta vectorial.

En cuanto a la longitud de cada uno de los elementos que componen el vector viene definido por en el CRS vytpe dentro del campo vsew, que según esté codificado tendrá más o menos bits cada elemento del vector, tal y como se ve en la Figura 67.

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	X	X	Reserved

Figura 67. Tabla de la codificación de vsew. [19]

Por ejemplo, si se tiene una codificación vsew = 000, entonces SEW es igual a 8, que quiere decir que cada uno de los elementos del vector tendrá una longitud de 8 bits.

Para el cálculo del número de elementos que tiene el vector será necesario conocer la longitud de cada uno de los vectores, es decir, VLEN. Para tener el número de elementos del vector simplemente se divide VLEN entre SEW.

Teniendo claras estas ideas se desarrolla la primera parte de la ALU vectorial.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library neorv32;
use neorv32.neorv32_package.all;

entity neorv32_cpu_vector_alu is
```

```
  port (
    -- global control --
    clk_i       : in  std_ulogic; -- global clock, rising edge
    rstn_i      : in  std_ulogic; -- global reset, low-active, async
    ctrl_i      : in  std_ulogic_vector(ctrl_width_c-1 downto 0);

    -- data input --
    vs1_i       : in  std_ulogic_vector(VLEN-1 downto 0); -- rf source 1
    vs2_i       : in  std_ulogic_vector(VLEN-1 downto 0); -- rf source 2
    vsew_i      : in  std_ulogic_vector(2 downto 0); -- CSR read data

    -- data output --
    alu_o       : out std_ulogic_vector(VLEN-1 downto 0); -- ALU result
    done_o      : out std_ulogic

  );
end neorv32_cpu_vector_alu;

architecture neorv32_cpu_cpu_vector_rtl of neorv32_cpu_vector_alu is

  signal start_add  : std_ulogic;
  signal start_sub  : std_ulogic;
  signal start_mul  : std_ulogic;
  --signal start_div  : std_ulogic;

  signal res_add    : std_ulogic_vector(VLEN-1 downto 0);
  signal res_sub    : std_ulogic_vector(VLEN-1 downto 0);
  signal res_mul    : std_ulogic_vector(VLEN-1 downto 0);
  --signal res_div    : std_logic_vector(VLEN-1 downto 0);
  signal res_0      : std_ulogic_vector(VLEN-1 downto 0) := (others => '-');

  signal cp_op      : std_ulogic_vector(1 downto 0);

  signal done : std_ulogic;
  signal alu_res : std_ulogic_vector(VLEN-1 downto 0);

  signal vsew   : std_ulogic_vector(2 downto 0);
  signal f_sew    : natural := 8;
  signal n_elements : natural;

begin

  vsew <= vsew_i;
  vtype_vsew : process(clk_i, vsew)
  begin
    case vsew is
          when sew_8  => f_sew <= 8;
          when sew_16 => f_sew <= 16;
          when sew_32 => f_sew <= 32;
          when sew_64 => f_sew <= 64;
          when others => f_sew <= 8;
        end case;
      end process vtype_vsew;
```

```
      n_elements <= VLEN/f_sew;

  cp_op <= ctrl_i(ctrl_v_alu_op1_c downto ctrl_v_alu_op0_c);

  start_add <= '1' when cp_op = v_alu_op_add_c else '0';
  start_sub <= '1' when cp_op = v_alu_op_sub_c else '0';
```

Código 31. Primera parte desarrollada de la ALU vectorial.

La segunda parte de la ALU vectorial consiste en permitir la realización de las operaciones aritméticas, que en nuestro caso serán la suma y la resta. Para ello hay que tener en cuenta el número de elementos que tiene un vector y su longitud, ya que, si se realizase una suma normal entre dos registros y uno de los elementos rebose, ese rebose pasaría al siguiente elemento, corrompiendo los elementos del vector. Es por ello que se diseñara una lógica que permita evitarlo, desarrollándose el Código 32.

```
  add_core: process(start_add, clk_i)
  begin
    if rising_edge(clk_i) and start_add = '1' then
    add : for i in 0 to n_elements-1 loop
        res_add((f_sew-1)+(f_sew*i) downto f_sew*i) <=
std_ulogic_vector(unsigned(vs1_i((f_sew-1)+(f_sew*i) downto f_sew*i)) +
unsigned(vs2_i((f_sew-1)+(f_sew*i) downto f_sew*i)));
    end loop add;
    else
      res_add <= (others => '-');
    end if;
  end process add_core;

  sub_core: process(start_sub, clk_i)
  begin
    if rising_edge(clk_i) and start_sub = '1' then
      sub : for i in 0 to n_elements-1 loop
        res_sub((f_sew-1)+(f_sew*i) downto f_sew*i) <=
std_ulogic_vector(unsigned(vs1_i((f_sew-1)+(f_sew*i) downto f_sew*i)) -
unsigned(vs2_i((f_sew-1)+(f_sew*i) downto f_sew*i)));
    end loop sub;
    else
      res_sub <= (others => '-');
    end if;
  end process sub_core;
```

Código 32. Núcleos de suma y resta de la ALU vectorial.

Por último, se tiene que sacar de la ALU vectorial el resultado de la suma o la resta, una vez estas se hayan completado. Para ello se comprobará que las señales res_add y res_sub contengan algo, y en ese caso se activará la señal que permite su salida.

Una vez permitida la salida de la ALU se tiene que decidir que resultado es el que sale, y es por ello que se comprueba que tipo de instrucción se tiene, mediante los enable creados anteriormente. Todo ello se ve reflejado en el Código 33.

```
  done_core: process(start_add, start_sub, start_mul, clk_i)
  begin
    if (res_add /= res_0 or res_sub /= res_0) and rstn_i = '1' then
      done <= '1';
      case cp_op is
        when v_alu_op_add_c => alu_res <= res_add;
        when v_alu_op_sub_c => alu_res <= res_sub;
        when others  => alu_res <= res_0;
      end case;
    else
      done <='0';
    end if;
  end process done_core;

  done_o <= done;
  alu_o  <= alu_res;

end neorv32_cpu_cpu_vector_rtl;
```

Código 33. Selector de la salida de la ALU vectorial.

5.2.5.2 Simulación de la ALU vectorial

Con el fin de ver en funcionamiento todo lo implementado en la ALU vectorial se desarrolló un programa sencillo en el que se cargan datos en dos registros vectoriales y se realiza una suma y una resta de los mismos.

```
.file  "blink_led_in_asm.S"
.section .text
.balign 4
.global main

main:

  li   a1, 255
  li   sp, 0x80001000
  addi sp, sp, -4
  sw   a1, 4(sp)
  li   a1, 0
  vle8.v v16, (sp)
  vle8.v v17, (sp)
  addi sp, sp, 4
  vadd.vv v3, v16, v17
  vsub.vv v3, v16, v17

.end
```

Código 34. Programa para comprobar las instrucciones vadd.vv y vsub.vv.

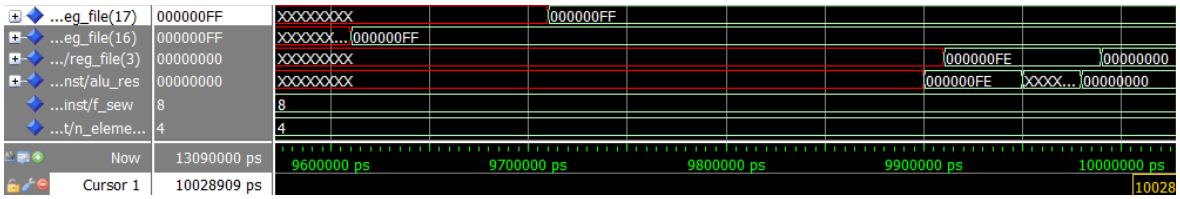


Figura 68. Simulación del Código 34.

Tal y como se observa en la Figura 68, lo primero es cargar dos registros vectoriales con dos números, en este caso el 255. Una vez cargados se tiene que la longitud de los elementos del vector es de 8 bits, es decir, dos números hexadecimales, y por lo cual cada vector tiene 4 elementos. De esta manera hay que comprobar que el rebose no pase al siguiente elemento del vector. Tal y como se ve en alu_res la suma de FF + FF da EF, por lo que el rebose no ha pasado al siguiente elemento del vector. Por último, se tiene que los resultados de la ALU vectorial son guardados en el registro v3. En la Figura 69 se observa cómo sería una suma si no se tuviese en cuenta el rebose de los elementos, viendo como está pasa al siguiente elemento.

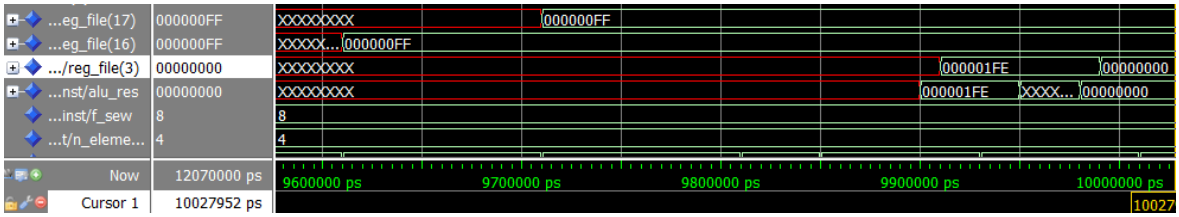


Figura 69. Simulación del Código 34 sin tener en cuenta la saturación de los elementos.

5.2.6 CONEXIÓN DE TODOS LOS COMPONENTES DISEÑADOS

Por último, tan solo se explicará que todos los componentes que se han ido diseñando a lo largo de este Trabajo Fin de Grado se han ido interconectando entre ello para que se puedan usar e interactúen entre ellos. Con ello se hace referencia a que a situaciones como la ALU vectorial, que carga y descarga datos del registro vectorial, y también obtiene información del CSR vtype, el SEW.

Para ello todos los componentes se incorporaron al mismo nivel de jerarquía en el que estaban el registro y la ALU escalar, por lo que se conectaron en el archivo noerv32_cpu.vhd.

```
-- Vector Register File ------------------------------------------------------
-- ----------------------------------------------------------------------------
    neorv32_cpu_vregfile_inst: neorv32_cpu_vregfile
  port map(
    -- global control --
    clk_i    => clk_i, -- global clock, rising edge
    ctrl_i   => ctrl,
    -- data input --
    rs1_i => rs1,
    rs2_i => rs2,
    alu_i  => valu_res,
    mem_i    => mem_rdata,
    done_i   => valu_idone,
    -- data output --
    vs1_o    => vs1,  -- operand 1
    vs2_o    => vs2   -- operand 2
  );
```

Código 35. Conexión del registro de vectores.

```
-- Vector ALU ------------------------------------------------------------------
-- ----------------------------------------------------------------------------
  neorv32_cpu_vector_alu_inst : neorv32_cpu_vector_alu
  port map (
    -- global control --
    clk_i       => clk_i, -- global clock, rising edge
    rstn_i      => rstn_i, -- global reset, low-active, async
    ctrl_i      => ctrl, -- main control bus

    -- data input --
    vs1_i       => vs1, -- vrf source 1
    vs2_i       => vs2, -- vrf source 2
    vsew_i      => vsew,  -- CSR vtype.vsew read data

    -- data output --
    alu_o       => valu_res, -- VALU result
    done_o      => valu_idone   -- iterative processing units done?
  );
```

Código 36. Conexión de la ALU vectorial.

Capítulo 6. ANÁLISIS DE RESULTADOS

Durante el desarrollo del Trabajo Fin de Grado se han obtenido resultados muy importantes y relevantes, y durante este capítulo se analizarán.

Se ha conseguido implementar las instrucciones más básicas que permitirían el uso de la extensión Zve32x, y entre ellas destacan principalmente las instrucciones de carga de datos en el registro de vectores, de escritura y lectura de los CSR, de configuración de vtype y vl y aritméticas vectoriales.

6.1 INSTRUCCIONES DE CARGA DE DATOS

Con la implementación de este tipo de instrucciones se ha permitido la carga de datos desde la memoria RAM al registro vectorial, posibilitando que posteriormente estos datos puedan ser procesados por instrucciones vectoriales. Esta instrucción es la base de la extensión, ya que sin ella no se podría tener datos sobre los que actuar.

6.2 INSTRUCCIONES DE ESCRITURA Y LECTURA DE CSR

El conjunto de instrucciones desarrolladas que permiten tanto escribir, como leer los nuevos CSR diseñado han sido realizas aprovechando muchos de los componentes y señales que ya se habían implementado para otros CSR, reduciendo en gran medida los recursos necesarios, si se compara con el caso en el que se hubiese diseñado una lógica distinta solo para ellos. De esta manera se implementaron sin apenas modificar la lógica ya existente obteniendo una gran integración y eficiencia en la ejecución de las instrucciones.

6.3 INSTRUCCIONES DE CONFIGURACIÓN DE VTYPE Y VL

Estas nuevas instrucciones permiten configurar, mediante código en ensamblador, dos CSR que son de lectura, como son vtype y vl. Como en el caso de las instrucciones de lectura y

escritura de los CSR, se aprovechó muchas señales y componentes ya existentenes, pero para estas instrucciones sí que fue necesario añadir lógica nueva, ya que requierian de una serie de condiciones propias.

En el caso del desarrollo de las condiciones propias, muchas de ellas se realizaron con instrucciones de alto nivel, como son la suma o la división, lo que pude conllevar a un mayor uso de recursos. Si bien es cierto que para alguna condición la mejora no compensaría el trabajo de diseño, en otros casos, en especial la división, se puede intentar optimizar el proceso, por ejemplo, con el uso de desplazamientos.

6.4 INSTRUCCIONES DE ARITMÉTICAS VECTORIALES

Estas instrucciones son las que permitirán realizar las operaciones vectoriales mas importantes, tales como la suma y la resta vectorial. La principal diferencia entre este tipo de instrucciones y la suma o resta escalar es que estas permiten sumar vectores enteros en un solo proceso, reduciendo el tiempo que tarda en comparación con la suma escalar, que tendría que sumar elemento por elemento.

En cuanto a su implementación en el núcleo, se desarrolló una ALU vectorial para realizar todos estos procesos. Su diseño no difiere mucho al de una ALU escalar, con la diferencia de que es capaza de identificar los elementos de un vector y operar con ellos, tal y como se demostró con el rebose de los elementos.

En este caso vuelve a pasar que ciertas condiciones que utilizan instrucciones de alto nivel, y otra vez especialmente con la división, se podrían optimizar con el uso de otro tipo de lógica, pero en este caso se justifica ya que gracias a ellos permite una mayor adaptabilidad a distintos tipos de vectores, con distintas longitudes y distintos números de elementos.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

7.1 OBJETIVOS CUMPLIDOS

7.1.1 COMPRENSIÓN DE LA ARQUITECTURA RISC-V

En primer lugar, se ha comprendido el funcionamiento de la arquitectura RISC-V, tanto en su parte de desarrollo de circuitos digitales, como la parte de desarrollo de programas en ensamblador, es decir, tanto la parte de software como la de hardware. Gracias al desarrollo en VHDL de la extensión basada en RISC-V se ha sido capaz de alcanzar un gran nivel de conocimiento de dicha arquitectura.

7.1.2 COMPRENSIÓN DEL NÚCLEO NEORV32

Se ha adquirido un gran nivel de conocimiento del núcleo NeoRV32, del cual no se conocía nada al inicio, ya que una buena parte del tiempo se ha empleado en conocer las señales que utiliza y cómo actúan, para poder implementar la extensión de la manera mas óptima posible. Para su completa comprensión también se utilizaron guías que mostraban su funcionamiento y documentos que desarrollaban las funcionalidades de algunos componentes.

7.1.3 CONOCER CÓMO SE DISEÑAN LOS MICROPROCESADORES

Este probablemente sea el objetivo más importante personalmente y el que se ha cumplido superando todas las expectativas, ya que cada proceso que se ha realizado durante el desarrollo de este Trabajo Fin de Grado ha sido parte del mismo proceso que se sigue para diseñar un microprocesador. Si bien es cierto que tan solo se ha desarrollado una pequeña parte, también ha servido para conocer el proceso de desarrollo.

7.1.4 SER CAPAZ DE IMPLEMENTAR UN NUEVO COMPONENTE

Al termino de este Trabajo Fin de Grado se ha conseguido implementar un nuevo componente en el núcleo, el cual permite implementar la extensión Zve32x que opera con vectores de 32 bits, aunque es cierto que no se ha implementado completamente, dejando algunas facetas para proyectos futuros.

7.1.5 TENER UN CONOCIMIENTO MÁS AMPLIO DE VHDL Y DE LAS HERRAMIENTAS DE SIMULACIÓN DE CIRCUITOS DIGITALES

Todo el Trabajo Fin de Grado se ha desarrolla en VHDL, por lo que se han aprendido nuevas funcionalidades de VHDL y se tiene un conocimiento mayor. En cuanto a las herramientas de simulación de circuitos digitales, principalmente se utilizó ModelSim, herramienta que se había utilizado durante la carrera, pero no al nivel de este Trabajo Fin de Grado ya que la complejidad de las simulaciones era muy superior, por lo que se adquirió un mayor nivel de conocimiento.

7.2 APORTACIONES

En cuanto a las aportaciones realizadas, lo más importante es que se ha implementado los componentes necesarios para que la extensión Zve32x pueda realizar operaciones aritméticas vectoriales. También se han implementado la gran mayoría de las señales necesarias para el funcionamiento de la mayoría de las instrucciones vectoriales, dejando allanado el camino para posibles continuaciones del proyecto.

Algunos de estos proyectos futuros se podrían basar en implementar funcionalidades que este proyecto no ha cubierto, tales como, las instrucciones de enmascaramiento o las de permutación.

Otro camino por el que se podría seguir desarrollando sería el de implementar una toolchain que incluya las instrucciones vectoriales, olvidándose de esta manera del ensamblar a mano.

Capítulo 8. ALINEACIÓN CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE

En 2015 los lideres mundiales acordaron una serie de 17 objetivos de desarrollo sostenible, por lo que en este capítulo se enumeraran los objetivos con los que se alinea el proyecto desarrollado y se indicará la contribución a los mismos.

8.1 INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURA

La extensión que se desarrolla consiste en permitir que un núcleo pueda realizar operaciones vectoriales, y tras una investigación profunda no se encontraron otros trabajos que realizasen alguna extensión que desempeñe una función parecida, por lo que se considera que es un trabajo innovador.

Figura 70. Icono del objetivo 9 Industria, innovación e infraestructura. [24]

8.2 PRODUCCIÓN Y CONSUMO RESPONSABLE

Gracias a actualizaciones como la desarrollada en este proyecto el núcleo NoeRV32 se mantiene en constante actualización, lo que evita que se tenga que comprar un nuevo componente desechando el anterior porque este no tenga unas funciones determinadas, por lo que se evita el consumo irresponsable de componentes.

Figura 71. Icono del objetivo 12 Producción y consumo responsable. [24]

8.3 ACCIÓN POR EL CLIMA

Esta extensión permite acelerar algoritmos de inteligencia artificial, mejorando el rendimiento del proceso, es decir, ahora tardará menos tiempo en realizar una misma operación, por lo que el consumo energético será menor, ayudando en la medida de lo posible en la reducción de la contaminación producida al generar energía eléctrica.

Figura 72. Icono del objetivo 13 Acción por el clima. [24]

8.4 ALIANZAS PARA LOGRAR LOS OBJETIVOS

Para el desarrollo de la extensión se parte de un núcleo que ha sido diseñado por otros desarrolladores de la comunidad, que ofrecen su trabajo de manera gratuita para que otros puedan continuarlo y usarlo. De la misma manera mi proyecto también será libre, para que cualquier persona pueda continuarlo o usarlo.

Figura 73. Icono del objetivo 17 Alianzas para lograr los objetivos. [24]

Capítulo 9. BIBLIOGRAFÍA

[1] «Electropediadigital,» 22 abril 2013. [En línea]. Available: http://electropediadigital.blogspot.com/2013/04/risc-vs-procesadores-vectoriales.html. [Último acceso: 2022 junio 19].

[2] «wikipedia,» 31 mayo 2022. [En línea]. Available: https://es.wikipedia.org/wiki/VHDL. [Último acceso: 16 junio 2022].

[3] J. Roca, «HardZone,» 06 julio 2021. [En línea]. Available: https://hardzone.es/reportajes/que-es/diseno-circuitos-verilog-vhdl/. [Último acceso: 16 junio 2022].

[4] «wikipedia,» 2022 junio 14. [En línea]. Available: https://es.wikipedia.org/wiki/RISC-V. [Último acceso: 2022 junio 16].

[5] «RISC-V,» [En línea]. Available: https://riscv.org/. [Último acceso: 29 junio 2022].

[6] J. D. M. Frías, de *Apuntes de la asignatura Sistemas Digitales II*, 2021, p. 143.

[7] J. D. M. Frías, de *Apuntes de la asignatura Sistemas Digitales II*, 2021, p. 145.

[8] «electropedia,» 22 abril 2013. [En línea]. Available: http://electropediadigital.blogspot.com/2013/04/procesadores-vectoriales.html. [Último acceso: 27 junio 2022].

[9] «wikipedia,» 2020 junio 30. [En línea]. Available: https://es.wikipedia.org/wiki/SIMD#/media/Archivo:SIMD.svg. [Último acceso: 27

junio 2022].

[10] R. Velasco, «SoftZone,» 14 enero 2020. [En línea]. Available: https://www.softzone.es/programas/sistema/sistema-operativo-superordenadores/. [Último acceso: 29 junio 2022].

[11] «Iberdrola,» [En línea]. Available: https://www.iberdrola.com/documents/20125/40288/Infografia_inteligencia_artificial.pdf/9140d25f-e071-345b-52d0-afc81297d9dd?t=1627271721296. [Último acceso: 29 junio 2022].

[12] M. Morales-Sandoval, «ResearchGate,» Enero 2012. [En línea]. Available: https://www.researchgate.net/figure/Figura-3-Flujo-de-diseno-de-un-FPGA-La-captura-del-diseno-de-entrada-puede-hacerse_fig1_241686407. [Último acceso: 2022 junio 29].

[13] «talent.com,» [En línea]. Available: https://es.talent.com/salary?job=Ingeniero+T%C3%A9cnico+Industrial. [Último acceso: 21 junio 2022].

[14] «Agencia Tributaria,» 2022 abril 29. [En línea]. Available: https://sede.agenciatributaria.gob.es/Sede/ayuda/manuales-videos-folletos/manuales-practicos/folleto-actividades-economicas/3-impuesto-sobre-renta-personas-fisicas/3_5-estimacion-directa-simplificada/3_5_4-tabla-amortizacion-simplificada.html. [Último acceso: 2022 junio 21].

[15] Sergio, «COMORECURRIR.ES,» [En línea]. Available: https://www.comorecurrir.es/cuantas-horas-laborales-tiene-un-mes/#:~:text=en%20un%20mes.-,Las%20constantes%20conocidas%20son%2040%20horas%20por%20semana%2C%2052%20semanas,A%20continuaci%C3%B3n%2C%20observa%20los%20resulta

dos.. [Último acceso: 21 junio 2022].

[16] M. d. Hacienda, «Hacienda.gob.es,» [En línea]. Available: https://www.hacienda.gob.es/Lists/DGPatrimonio/junta%20consultiva/informes/informes%202020/2019-040gastosgrales.pdf. [Último acceso: 21 junio 2022].

[17] Comunidad, «github,» 2022 junio 14. [En línea]. Available: https://github.com/stnolting/neorv32. [Último acceso: 2022 junio 18].

[18] Comunidad, «github,» [En línea]. Available: https://stnolting.github.io/neorv32/ug/#_simulation_using_a_shell_script_with_ghdl. [Último acceso: 2022 junio 18].

[19] Comunidad, «github,» 21 septiembre 2021. [En línea]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0. [Último acceso: 2022 junio 19].

[20] Comunidad, «Github,» [En línea]. Available: https://github.com/stnolting/riscv-gcc-prebuilt. [Último acceso: 19 julio 2022].

[21] J. D. M. Frías, «Apuntes de la asignatura de Sistemas Digitales II,» Madrid, 2021, p. 72.

[22] J. D. M. Frías, «Apuntes de la asignatura Sistemas Digitales II,» 2021, p. 84.

[23] J. D. M. Frías, Apuntes de la asignatura de Sistemas Digitales II, 2021.

[24] «Objetivos de desarrollo sostenible,» [En línea]. Available: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/. [Último acceso: 30 junio 2022].

[25] «wikipedia,» 2019 julio 30. [En línea]. Available:

https://es.wikipedia.org/wiki/Procesador_vectorial#:~:text=Un%20procesador%20vectorial%20es%20un,s%C3%B3lo%20un%20dato%20cada%20vez.. [Último acceso: 2022 junio 27].

[26] «wikipedia,» 14 junio 2022. [En línea]. Available: https://es.wikipedia.org/wiki/RISC-V. [Último acceso: 29 junio 2022].

[27] «mouser,» 10 marzo 2022. [En línea]. Available: https://www.mouser.es/new/terasic-technologies/terasic-fpga-dev-cyclone-kits/. [Último acceso: 2022 junio 29].