UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Diseño, implementación y desarrollo de procesador mínimo basado en arquitectura RISC-V

PROYECTO FIN DE GRADO

Óscar Domínguez Ardaiz
Grado en Ingeniería de Computadores

Madrid, 2025

UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Grado en Ingeniería de Computadores

Diseño, implementación y desarrollo de procesador mínimo basado en la arquitectura RISC-V

PROYECTO FIN DE GRADO

Óscar Domínguez Ardaiz
Grado en Ingeniería de Computadores

Bajo la dirección de:
Dr. Borja Bordel Sánchez

Madrid, 2025

Título: Diseño, implementación y desarrollo de procesador mínimo basado en arquitectura RISC-V

Autor: Óscar Domínguez Ardaiz

Grado en Ingeniería de Computadores

Dirección:

Dr. Borja Bordel Sánchez, Universidad Politécnica de Madrid

A mis padres y mi hermana

Agradecimientos

Gracias a mi familia por siempre estar ahí cuando se les necesita y que me han dado el privilegio de poder estar aquí, a mis amigos por apoyarme en todo, y a los profesores que me han guiado hasta donde he llegado.

Abstract

Due to the increasing tensions between different nations, it is increasingly valued to be able to develop and build processors locally. In addition, it has been seen a raise in the demand of embedded systems, so a RISC processor is more interesting.

In order to avoid designing the whole architecture we will use the open-source architecture RISC-V, but not all instructions will be implemented, we will implement the 10 most indispensable instructions for the basic function of the processor, meaning that the executed code might not be the most optimised. This is because in order to try to implement some functions, the coder or the compiler will have to do some intermediate steps.

The project is developed in an FPGA, for testing the correctness of the architecture, using VHDL.

The designed processor contains an ALU which executes 7 arithmetic or logic operations, a 16-register bank, a program counter, a control unit, and following the Harvard memory model, a data memory module and an instruction memory module which can be programmed using the provided inputs.

Even though the chosen instructions have shown some functionality, they weren't as flexible as it is expected from a general-purpose processor due to the lack of branching options, because the unconditional branch is constraining.

It has shown that the lack an operation that allows the programmer to introduce an external value to the memory or the registers, is a very remarkable limitation.

However, if it is decided that the shift operations are being kept, with the addition of a logical or arithmetic instruction using immediates, and the addition of a conditional Branch of equality would be enough to fulfill the deficits of the instruction set, summing a total of 12 instructions.

Lastly, the legal, ethical, social and environmental implications have been taken into account.

Resumen

Debido al aumento de tensiones entre distintos países, cada vez es más valorada la capacidad de desarrollo y manufacturación de procesadores localmente. Además, se ha observado un crecimiento en la demanda de sistemas empotrados, por lo que se valora positivamente el bajo consumo y tamaño limitado de estos procesadores, por lo tanto, es preciso el uso de un procesador de tipo RISC.

Para evitar el diseño de la arquitectura nos basaremos en la arquitectura open-source RISC-V, pero no se implementarán todas las instrucciones del conjunto de instrucciones de RISC-V, tan sólo se implementarán las 10 instrucciones más indispensables para el funcionamiento del procesador, lo que significa que es el código ejecutado en unidad puede no ser el más óptimo, ya que para implementar ciertas funciones con software hay que dar pasos intermedios.

El proyecto se desarrolla en una FPGA, para la comprobación del funcionamiento de la arquitectura, programada con VHDL.

El procesador diseñado cuenta con una ALU que ejecuta las 7 instrucciones aritméticas o lógicas, un banco de 16 registros, un contador de programa, una unidad de control, y siguiendo el modelo de memoria de Harvard, un módulo de memoria de datos y un módulo de memoria de instrucciones programable mediante las entradas proporcionadas.

Aunque el conjunto de instrucciones elegido ha demostrado cierta funcionalidad, no ha sido todo lo versátil que cabría esperar de un procesador de uso general por falta de opciones de ramificación, ya que el salto incondicional resulta limitante.

También se ha encontrado una limitación considerable al no contar con ninguna operación que permita introducir un valor a los registros o a la memoria en un inicio.

No obstante, si se quiere mantener las instrucciones de desplazamiento, con una instrucción lógica o aritmética que use inmediatos, y la instrucción de ramificación bajo condición de igualdad sería suficiente para suplir estas carencias sumando un total de 12 instrucciones.

Por último, se han tenido en cuenta las consecuencias legales, éticas, sociales y medioambientales del proyecto.

Tabla de Contenido

1. Introducción 1

2. Objetivos 3

3. Estado de la cuestión 4

3.1. RISC vs CISC 4

3.1.1. Historia de RISC 4

3.2. MISC5

3.2.1. MuP21 6

3.2.2. F21 6

3.2.3. P24 6

4. Material y métodos 9

4.1. RISC-V 9

4.1.1. Historia de RISC-V 9

4.1.2. Especificaciones de la arquitectura 10

4.2. Vivado 13

4.2.1. VHDL 13

4.3. Diseño top-down 14

4.4. Diseño bottom-up 14

5. Diseño e implementación 15

5.1. ISA 15

5.2. Módulos 17

5.2.1. Banco de registros 21

5.2.2. Contador de programa 23

5.2.3. Memoria 25

5.2.4. ALU 27

5.2.5. Unidad de control 28

5.3. Diseño MISC 32

6. Resultados 34

6.1. Especificaciones 34

6.2. Pruebas unitarias 34

6.2.1. Prueba registro básico 35

6.2.2. Prueba decodificador 4 bits 37

6.2.3. Prueba banco registros 39

6.2.4. Prueba memoria 41

6.2.5. Prueba ALU 43

6.3. Prueba MISC 45
6.3.1. Ejecución de todas las instrucciones 46
6.3.2. Programa de comprobación de overflow 49
7. Aspectos Sociales, Éticos, Legales y Ambientales 51
8. Discusión 52
8.1. Banco de registros 52
8.2. PAI 52
8.2.1. Inicialización 52
8.2.2. Ramificación 53
8.3. Anotación de diseño 53
9. Conclusiones 55
Referencias 57
Anexo 59
Anexo 1: Código 59

Lista de Figuras

Figura 5.1: Código de registro básico de 32 bits 17
Figura 5.2 : Diseño registro 32 bits 18
Figura 5.3 : Código de decodificador de 4 bits 19
Figura 5.4 : Diseño decodificador 20
Figura 5.5 : Código de banco de registros 21
Figura 5.6 : Diseño banco de registros 22
Figura 5.7 : Código de contador de programa 23
Figura 5.8 : Diseño contador de programa 24
Figura 5.9: Código de módulo de memoria 25
Figura 5.10 : Diseño parcial de la memoria 26
Figura 5.11: Código de ALU 27
Figura 5.12 : Diseño ALU 27
Figura 5.13 : Señales de la unidad de control 30
Figura 5.14: Construcción de inmediatos en la unidad de control 31
Figura 5.15 : Conexión de los módulos 32
Figura 6.1: Prueba de registro básico 35
Figura 6.2 : Código generación de reloj 35
Figura 6.3: Código prueba registro 32 bits 36
Figura 6.4: Prueba de decodificador de 4 bits 37
Figura 6.5: Parte del código del banco de prueba del decodificador 38
Figura 6.6: Prueba del banco de registros 39
Figura 6.7 : Parte del código de las pruebas del banco de registros 40
Figura 6.8: Prueba de memoria 41
Figura 6.9 : Parte del código del banco de pruebas de la memopria 42

Figura 6.10: Prueba de ALU 43
Figura 6.11 : Parte del código de banco de pruebas de la ALU.. 44
Figura 6.12: Modificación del banco de registros para las pruebas 45
Figura 6.13: Banco de pruebas del procesador 46
Figura 6.14: Simulación del banco de pruebas con 1 iteración 47
Figura 6.15: Simulación del banco de pruebas con 2 iteraciones 47
Figura 6.16 : Banco de pruebas de programa de comprobación de overflow 49
Figura 6.17 : Simulación del programa de comprobación de overflow 50

Lista de Tablas

Tabla 4.1 : Estado de los módulos de RISC-V 10

Tabla 4.2 : Tipos de instrucción en RISC-V 12

Tabla 5.1: Tipos de instrucción en el MISC 16

Tabla 5.2: Codificación instrucciones MISC 17

Abreviaturas y Acrónimos

UPM	Universidad Politécnica de Madrid
GPIO	General Purpose Input/Output
AC	Alternationg Current
DC	Direct Current
RISC	Reduced Instruction Set Computing
MIPS	Microprocessor without Interlocked Pipeline Stages
MISC	Minimal Instruction Set computer
ISA	Instruction Set Architecture
BSD	Berkeley Software Distribution
CISC	Complex Instruction Set Computing
FPGA	Field-programmable gate array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
ALU	Unidad Aritmético-Lógica
STEREO	Solar Terrestrial Relations Observatory
NASA	National Aeronautics and Space Administration
ODS	Objecitvos de Desarrollo Sostenible

1. Introducción

En los últimos años se han observado inversiones millonarias por parte de Europa y Estados Unidos en un aumento de demanda de sistemas empotrados y en el diseño y construcción de sus procesadores en sus respectivos territorios debido a las crecientes tensiones internacionales y la incertidumbre que crean estas tensiones.

Por esta última razón se evidencian las ventajas del desarrollo local de sistemas empotrados. Para ello hay que ver entender sus componentes y abordar la cuestión del diseño de su componente esencial.

Los sistemas empotrados se componen de distintos módulos de comunicación, como SPI, I^{2}C o Wi-Fi, algún módulo de entrada/salida, como pines GPIO (General Purpose Input/Output), un módulo de reloj, un módulo de energía que alimenta el resto de componentes, algún conversor AC/DC o DC/DC y un microprocesador, siendo este último el componente esencial, ya que sin él no se podría ejecutar ningún programa en el sistema [1] [2].

Los procesadores de sistemas empotrados, por diseño, precisan de un menor consumo y una mayor eficiencia energética que sus contrapartes convencionales, debido a que se suelen desplegar en sistemas que alimentan otros componentes, tienen una baja alimentación o están alimentados por una batería, y así, con el bajo consumo se consigue alargar la vida de esta misma.

Este tipo de microprocesadores se usan también porque al consumir menos, se calientan menos y hace falta menor refrigeración, haciéndolos perfectos para poder implementarlos en un circuito de pequeña escala.

Al ser necesario un procesador eficiente y de bajo consumo, la arquitectura que primero se viene a la cabeza es la RISC (Reduced instruction set computing), ya que por su filosofía de diseño son más eficientes que los CISC (Complex Instruction Set Computing) y usan menos recursos energéticos.

2. Objetivos

El objetivo principal del proyecto se centra en crear un procesador basado en la arquitectura RISC-V usando su juego de instrucciones.

A partir del juego de instrucciones de RISC-V, se ha pretendido minimizar a la parte más esencial de dicho juego para crear un procesador tipo MISC (Minimal Instruction Set Computing) que permita la ejecución de cualquier tipo de programa usando un solo núcleo.

Para este propósito se ha decidido usar VHDL (VHSIC Hardware Description Language) como lenguaje descriptivo, y AMD Vivado Design Suite como entorno de desarrollo.

Por último, se ha establecido el objetivo de probar el correcto funcionamiento del procesador mediante simulación de la ejecución de un programa en una FPGA (Field-programmable gate array) usando el anteriormente mencionado Vivado.

3. Estado de la cuestión

3.1. RISC vs CISC

El término RISC surge para definir a los procesadores que se empezaron a diseñar desde 1975, pero tuvieron un mayor auge en cantidad de diseños en la época de 1980, en los que se simplificaban las instrucciones y se reducía la cantidad total de instrucciones en las arquitecturas.

Actualmente los RISC no tienen por qué referirse a la cantidad de instrucciones, si no que se refiere a la complejidad de las instrucciones, poniendo un énfasis a la cantidad de accesos a memoria por instrucción. De hecho, a veces se prefiere usar el nombre de arquitectura de carga-almacenamiento, referenciando de esa forma a sus únicas 2 instrucciones de gestión de memoria [15].

Mientras tanto, CISC se creó después del término RISC para referirse a todo lo que no es un RISC.

Las principal diferencia es, como se ha mencionado, la cantidad de accesos a memoria por instrucción, pero también se diferencian las dos filosofías de diseño porque en los procesadores RISC los tamaños de las instrucciones no son variables, mientras que en los CISC pueden ser de distintos tamaños incluso dentro de la misma arquitectura; los RISC ejecutan por lo menos una instrucción por ciclo; todos los registros de uso general de los RISC se pueden usar de igual forma como fuente o destino de las operaciones realizadas; los modos de direccionamiento son simples; y hay pocos tipos de datos a nivel hardware.

3.1.1. Historia de RISC

Según Flynn, el considerado primer RISC es el microprocesador que montaba la computadora IBM 801 diseñado durante la década de 1970 (es importante no confundirlo con el modelo IBM 801 de 1934), basándose en el modelo carga/almacenamiento con el objetivo de mejorar el rendimiento simplificando las instrucciones, reduciendo su complejidad [3]. Este modelo implementó sus instrucciones para que se ejecuten en un ciclo, fijó el tamaño de sus instrucciones a 32 bits, y se establecieron 32 registros de uso general [4].

No obstante, no empieza a usarse el concepto RISC hasta que Andrew S. Tanenbaum publicó una investigación en la que demostraba que no sólo no se usaban la gran mayoría de las instrucciones de un conjunto de instrucciones de un procesador promedio de ese entonces, si no que, de un programa complejo de 10.000 líneas, se podría representar al completo en un conjunto de instrucciones con un código de operación con tamaño fijo de 8 bits [5].

Un laboratorio que fue importante en el progreso de los RISC fue el laboratorio de Berkeley de la Universidad de California, que diseñando el RISC-I en 1980 llegaron a las mismas conclusiones que los investigadores de IBM: simplificando las instrucciones se puede obtener un mayor rendimiento. [6] El RISC-I contaba con 39 instrucciones de 32 bits con un tamaño de código de operación de 7 bits, y con 32 registros.

En la universidad de Standford, tras ver los resultados del IBM 801 y del laboratorio de Berkeley, se inicia el proyecto MIPS, compuesto por investigadores con experiencia en compiladores, que pretendía bajar el nivel computacional de los compiladores y minimizar los parones del pipeline reduciendo la cantidad de ciclos que tardaban en ejecutarse cada instrucción a 1, exceptuando las instrucciones de carga y almacenamiento de datos [7]. Esta arquitectura contaba con 11 instrucciones de 32 bits con 32 registros de 32 bits.

La última arquitectura RISC más relevante es ARM, que se basó en los RISC diseñados en el laboratorio de Berkeley. A este diseño le incluyeron varios cambios como la habilidad de servir rápidamente interrupciones, entre otras cosas, disminuyendo el tamaño del contador de programa a 24 bits, permitiendo que el contador entre en una instrucción de 32 bits y guardando el estado del programa en una sola acción [8].

3.2. MISC

Un procesador del tipo MISC, es un procesador que, como su nombre indica, tiene un conjunto de instrucciones mínimo. Normalmente este conjunto se suele componer por menos de 32 instrucciones, aunque el término se concibió para nombrar los procesadores con un número e instrucciones de entre 8 y 16.

Un procesador MISC podría llegar a considerarse una simplificación de un RISC a base de eliminar instrucciones no esenciales, haciendo que a la vez se simplifique la arquitectura, pero, muy probablemente, perdiendo algunas funcionalidades de los RISCs.

A continuación, se van a explicar algunos de los procesadores tipo MISC más significativos:

3.2.1. MuP21

El MuP21 es un procesador diseñado por Charles H. Moore tras el aumento de diseños de procesadores tipo RISC en los años 80, en 1990, aunque tuvo revisiones posteriores. Inicialmente contaba con un total de 24 instrucciones, las cuales son:

- Instrucciones de transferencia: *JUMP*, *CALL*, *RET*, *JZ, JCZ*
- Instrucciones de memoria: *LOAD, STORE, LOADP, STOREP, LIT*
- Instrucciones de ALU: *COM, XOR, AND, ADD, SHL, SHR, ADDNZ*
- Instrucciones de registros: *LOADA, STOREA, DUP, DROP, OVER, NOP*

El MuP21 encontró sus usos en el procesamiento de vídeo, por eso incluyó un coprocesador gráfico que incluía 7 instrucciones más: *Black*, *Sync*, *Refresh*, *Skip*, *Burst*, *Pixel* y *Jump* [9].

El MuP21 contaba con una pila de retorno y una pila de datos, además, el contador de programa estaba separado del banco de registros. También, en el mismo banco de registros se marca el registro más alto (Top, nombrado registro T) como el registro por defecto para las operaciones. Contaba con una ALU (Unidad Aritmético-Lógica) que tomaba como parámetro el contenido de T y la cima de la pila de datos, registrando el resultado también en T. Contiene un registro A (Address) donde se guarda la dirección de memoria para consultas, guardar datos en o desde memoria. Por último, tiene una pequeña memoria donde se almacenan las 5 instrucciones siguientes a ejecutar.

3.2.2. F21

El F21 es una evolución del MuP21, también de Charles H. Moore, que además de mejoras de velocidad y memoria, incluía más instrucciones de ramificación y más coprocesadores para dar soporte a la entrada y salida analógica y digital, entrada y salida de conexión a redes y un puerto de entrada salida paralelo [10].

3.2.3. P24

Para la misión STEREO (Solar Terrestrial Relations Observatory) de la NASA (National Aeronautics and Space Administration) se construyeron 2 satélites, y cada uno montaba al menos un procesador P24. El P24 se basó en el MuP21

mencionado anteriormente [11]. En la misión se usaron para el telescopio de baja potencia y para la unidad de control de los sensores. Se sintetizó en una FPGA con triple redundancia para soportar las condiciones del espacio. El total de instrucciones implementadas fueron 26:

- Instrucciones de transferencia: *JUMP, CALL, RET, JZ, JNC*
- Instrucciones de memoria: *LDP, LDI, LD, STP, ST*
- Instrucciones de ALU: *COM, XOR, AND, ADD, SHL, SHR, MUL, DIV*
- Instrucciones de registros: *LDA, STA, DUP, DROP, OVER, NOP*

Como se puede ver, la mayoría de instrucciones son las mismas que el MuP21, o simplemente se han cambiado los nombres, aunque se debe destacar la inclusión de *MUL* y *DIV*, es decir, se añadieron la instrucción de multiplicar y dividir [12].

Por último, es de suma importancia indicar que, como el MuP21 el P24 cuenta con el registro A, la memoria de instrucciones I, el contador de programa P y el acumulador de la ALU T, y además define la cima de la pila de retorno R, la cima y la cima de la pila de datos S.

Un procesador que en ocasiones es nombrado como RISC es el Transputer, que, aunque sea cierto que su número de instrucciones es reducido, teniendo 16 principales y otras 16 secundarias, es difícil considerarlo un RISC debido a que sus instrucciones están implementadas con microcódigo, es decir, no se ejecuta directamente según se decodifica, y tenía instrucciones relativas a la memoria más propias de los procesadores tipo CISC.

4. Material y métodos

4.1. RISC-V

RISC-V es una arquitectura con un conjunto estándar de instrucciones abierto, y como su nombre indica es tipo RISC. La especificación de la ISA se liberó al dominio público, aunque los documentos que la definen se han licenciado bajo *Creative Commons 4.0* requiriendo el citado de la fuente del manual.

El proyecto RISC-V está respaldado por la RISC-V Foundation, que es una organización sin ánimo de lucro con sede en Suiza.

4.1.1. Historia de RISC-V

Los primeros pasos del RISC-V se remontan a las investigaciones del Profesor Krste Asanović, y los estudiantes Yunsup Lee y Andrew Waterman quienes empezaron el conjunto de instrucciones en mayo de 2010 como parte del Laboratorio de Computación Paralela (*Par Lab*) en la Universidad de California en Berkeley [13].

Al crearse en la Universidad de California en Berkeley, al inicio RISC-V usaba la licencia BSD (Berkeley Software Distribution), que se incluyó en la primera publicación, que fue el 13 de mayo de 2011 [14]. En esta publicación se indicaban los detalles del conjunto de instrucciones.

En 2015 se crea la RISC-V Foundation, y aunque la especificación de la ISA ya era de dominio público desde la primera publicación, los documentos descriptivos no recibieron la licencia *Creative Commons* hasta la creación de la fundación.

En 2018 la fundación anuncia una colaboración con Linux Foundation, en la que Linux Foundation proporcionará ayuda operacional, técnica y estratégica.

En diciembre de 2018 RISC-V Foundation anunció su traslado a Suiza debido a preocupaciones geopolíticas que involucraban a Estados Unidos y al acceso a la propiedad intelectual de la fundación por parte de la comunidad.

4.1.2. Especificaciones de la arquitectura

No hay una única ISA de RISC-V ya que es una arquitectura abierta y en este proyecto nos hemos centrado en el conjunto de instrucciones sin privilegios. A fecha de julio de 2025 la última versión de la ISA sin privilegios es de mayo de 2025 [15]. (**¡Error! No se encuentra el origen de la referencia.**)

Tabla 4.1 : Estado de los módulos de RISC-V

Base	**Version**	**Status**
RV32I	2.1	Ratified
RV32E	2.0	Ratified
RV64E	2.0	Ratified
RV64I	2.1	Ratified
Extension	**Version**	**Status**
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
Zicntr	2.0	Ratified
Zihintntl	1.0	Ratified
Zihintpause	2.0	Ratified
Zimop	1.0	Ratified
Zicond	1.0	Ratified
Zilsd	1.0	Ratified
M	2.0	Ratified
Zmmul	1.0	Ratified
A	2.1	Ratified
Zawrs	1.01	Ratified
Zacas	1.0	Ratified
Zabha	1.0	Ratified
RVWMO	2.0	Ratified

Ztso	1.0	Ratified
CMO	1.0	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
Zfh	1.0	Ratified
Zfhmin	1.0	Ratified
BF16	1.0	Ratified
Zfa	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
C	2.0	Ratified
Zce	1.0	Ratified
Zclsd	1.0	Ratified
B	1.0	Ratified
V	1.0	Ratified
Zbkb	1.0	Ratified
Zbkc	1.0	Ratified
Zbkx	1.0	Ratified
Zk	1.0	Ratified
Zks	1.0	Ratified
Zvbb	1.0	Ratified
Zvbc	1.0	Ratified
Zvkg	1.0	Ratified
Zvkned	1.0	Ratified

Zvknhb	1.0	Ratified
Zvksed	1.0	Ratified
Zvksh	1.0	Ratified
Zvkt	1.0	Ratified
Zicfiss	1.0	Ratified
Zicfilp	1.0	Ratified

El espacio direccionable depende de la ISA base escogida: para las RV32x es de 32 bits, mientras que las RV64x el tamaño es de 64 bits. El espacio direccionable es circular, es decir, la palabra máxima direccionable es contigua a la dirección 0.

El tamaño de palabra es, independientemente de la ISA base, de 32 bits.

En el manual del conjunto de instrucciones de RISC-V no se especifica el tamaño que debe tener la memoria, pero se indica que el espacio de memoria es único y que se debe hacer posible el acceso a cada byte de forma independiente [15].

El tamaño de los registros es de 32 bits independientemente de la ISA escogida, pero la cantidad de registros cambia dependiendo de la ISA base. Las ISA RVxxI 32 registros y otro que es el contador de programa, mientras que las ISA RVxxE tienen 16 registros, más el contador de programa. En los dos tipos de ISA el registro *x00* está conectado a 0.

Las instrucciones de las ISAs pertenecen a uno de los siguientes tipos: R, I, S, B, U o J. (Tabla 4.2)

Tabla 4.2 : Tipos de instrucción en RISC-V

31	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		Tipo R
inm[11:0]				rs1		funct3		rd		opcode		Tipo I
inm[11:5]		rs2		rs1		funct3		inm[4:0]		opcode		Tipo S
inm[12|10:5]		rs2		Rs1		funct3		inm[4:1|11]		opcode		Tipo B
inm[31:12]								rd		opcode		Tipo U
imm[20|10:1|11|19:12]								rd		opcode		Tipo J

Los tipos B y J son variaciones de los tipos S y U respectivamente. La diferencia entre los tipos S y B es que el tipo B codifica inmediatos de 12 bits múltiplos de 2, lo que significa que, en la instrucción, el bit 7, lo que era el bit 0 del inmediato, ahora es el bit 11 del inmediato; y en el bit 31 de la instrucción, lo que antes era el bit 11 del inmediato, ahora es el bit 12.

La diferencia entre el tipo U y el tipo J es que para formar el inmediato de tipo U, se desplaza 12 bits hacia la izquierda, mientras que el inmediato se crea desplazándolo un bit a la izquierda y fijando el bit 20 y 11, para que haya mayor solapamiento con el resto de instrucciones.

4.2. Vivado

Vivado es un programa utilizado para la síntesis y análisis de lenguajes de descripción de hardware como Verilog o VHDL. En este proyecto se ha utilizado VHDL para describir el comportamiento del MISC.

Se ha utilizado Vivado por la simplificación del flujo de diseño de VHDL, ya que, por ejemplo, hace el enrutamiento de forma automática. Otra razón por la elección de Vivado es que ya tenía experiencia previa con Vivado.

4.2.1. VHDL

VHDL es un lenguaje de descripción de hardware, es decir, que se usa para definir circuitos digitales. Este lenguaje se usa comúnmente para programar FPGAs o similares.

Hay 4 formas principales de describir circuitos en VHDL:

- Funcional: Se describe únicamente teniendo en cuenta el comportamiento de las entradas y salidas. Se suelen encapsular sentencias en procesos que se ejecutan de forma paralela.
- Flujo de Datos: Se describe cómo las señales interactúan las unas con las otras.
- Estructural: Se describen componentes que interactúan entre ellos y definen cómo funciona el circuito.
- Mixta: Es una combinación de las anteriores.

Tras programar el circuito deseado se debe hacer la simulación funcional para comprobar que, efectivamente, funciona como se espera. Después se hace la

síntesis, que consiste en adaptar el diseño en VHDL a una FPGA o similar. A continuación, se simula el circuito sintetizado para comprobar que funciona correctamente tras este proceso, ya que hay sentencias VHDL que no se pueden sintetizar. Por último, se deben enrutar los bloques digitales obtenidos para obtener el menor retardo posible. En este punto nuestro dispositivo ya se puede programar, aunque es recomendable anotar los retardos de nuestro circuito y simularlo, para evitarnos el posible caso de que los retardos internos hagan que el diseño falle.

4.3. Diseño top-down

La estrategia de diseño top-up consiste en desglosar el problema en sus piezas fundamentales e ir implementando cada una por separado e ir juntándolas según la jerarquía en la que interactúan entre sí.

En nuestro caso se ha aplicado de la siguiente forma:

1. Se ha dividido la arquitectura en distintos chips que hacen una función distinta, que son: unidad de control, memoria de instrucciones, memoria de datos, banco de registros, contador de programa y ALU.
2. A continuación, en caso de ser posible, estos componentes se han vuelto a dividir en componentes más pequeños repetidamente hasta llegar a sus componentes fundamentales.

Esta estrategia se ha usado para definir los componentes que va a tener el MISC.

4.4. Diseño bottom-up

La estrategia de diseño bottom-up consiste en diseñar primero las partes básicas a detalle y luego ir entrelazándolas. Esta estrategia se ha usado para que, una vez definidos los módulos a diseñar, ser capaces de definir los componentes básicos e ir subiendo en la jerarquía.

5. Diseño e implementación

5.1. ISA

El proyecto se ha basado en la especificación base de la ISA RV32E al buscar la mayor simplicidad de la estructura.

Para reducir al máximo las instrucciones se ha seguido el siguiente proceso:

1. Como con lógica *NAND* se puede definir el resto de sentencias lógicas se ha escogido la sentencia *AND*, y *XOR* ya que la operación 1 *XOR* x es equivalente a *NOT* x, por lo que al encadarlo a *AND* se tiene la lógica *NAND*.
2. Es necesario una sentencia de *LOAD* y otra de *STORE* para poder usar la memoria, y como vamos a usar únicamente palabras de 32 bits, usamos las sentencias *LW* y *SW*.
3. Para hacer comparaciones de cualquier tipo se ha escogido *SLT*, ya que lo único que no está contemplado en la instrucción base es la igualdad, pero se puede obtener si dos números son iguales con *SLT r1, r2, r3; SLT r1, r4, r5* (r4 = *r2-1); *SLT r3, r0, r6; SLT r5, r0, r7* (r1 <= r2 && r1>r2-1 entonces r=r2).
4. Como ya podemos comprobar si un número es menor, igual o mayor que otro, es cuando debemos añadir una instrucción de ramificación para, por ejemplo, poder hacer bucles. Como ya tenemos una instrucción para comprobar relaciones entre números, sólo es necesario hacer el salto, por lo que añadimos la instrucción *JAL*.
5. Por último, porque puede simplificar algunas operaciones, se han añadido las instrucciones *SRL* y *SLL*.

No se ha considerado necesario añadir la instrucción, por muy elemental que sea, ya que se puede sustituir por una instrucción como *ADD r0, r0, r0*, tal y como se especifica en el manual, aunque en la implementación final se puede ejecutar un equivalente introduciendo una instrucción no válida [15].

No se ha añadido *FENCE* porque se ha decidido que no va a tener I/O. Tampoco se han añadido instrucciones que usen inmediatos, exceptuando *JAL, LW* y *SW*, ya que se entiende que si se quiere usar un inmediato siempre se puede cargar de memoria.

Con las instrucciones seleccionadas se hace uso de las instrucciones tipo R, I, S y J. (Tabla 5.1)

Tabla 5.1: Tipos de instrucción en el MISC

<table>
<tr><td>31</td><td>25</td><td>24</td><td>20</td><td>19</td><td>15</td><td>14</td><td>12</td><td>11</td><td>7</td><td>6</td><td>0</td><td></td></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td colspan="2">rs1</td><td colspan="2">funct3</td><td colspan="2">rd</td><td colspan="2">opcode</td><td>Tipo R</td></tr>
<tr><td colspan="4">inm[11:0]</td><td colspan="2">rs1</td><td colspan="2">funct3</td><td colspan="2">rd</td><td colspan="2">opcode</td><td>Tipo I</td></tr>
<tr><td colspan="2">inm[11:5]</td><td colspan="2">rs2</td><td colspan="2">rs1</td><td colspan="2">funct3</td><td colspan="2">inm[4:0]</td><td colspan="2">opcode</td><td>Tipo S</td></tr>
<tr><td colspan="8">inm[20|10:1|11|19:12]</td><td colspan="2">rd</td><td colspan="2">opcode</td><td>Tipo J</td></tr>
</table>

En la tabla anterior se pueden ver los apartados “funct7”, “funct3”, “rs2”, “rs1”, “rd”, “opcode” e “inm”. “Opcode” es el código de identificación de la instrucción, que junto a “funct7” y “funct3”, si procede, se especifica qué instrucción es en concreto. Rs1 y rs2 son los registros que se introducen como parámetros en la operación, y rd es el registro destino del resultado. Por último, “inm” es el inmediato introducido como parámetro en la instrucción.

Con funct3 y funct7 que están en el manual de RISC-V tenemos la codificación [15]. (Tabla 5.2)

Tabla 5.2: Codificación instrucciones MISC

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	111	rd	0110011	AND
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	010	rd	0110011	SLT
inm[11:0]		rs1	010	rd	0000011	LW
inm[11:5]	rs2	rs1	010	inm[4:0]	0100011	SW
inm[20|10:1|11|19:12]				rd	1101111	JAL

5.2. Módulos

Se ha empezado el diseño haciendo un registro básico de 32 bits, ya que el tamaño de la palabra es de 32 bits, y los registros son de 32 bits.

El registro creado cuenta con entrada "reset", entrada "chip select" y entrada de lectura/escritura, con 0 modo escritura y 1 escritura (Figura 5.1). Para comprobar su funcionamiento más fácilmente se ha añadido la señal data que simula el funcionamiento de 32 biestables D. (Figura 5.2)

```
entity register32bits is
    Port ( dataIn : in STD_LOGIC_VECTOR (31 downto 0);
           reset : in STD_LOGIC;
           read_write : in STD_LOGIC;
           clk : in STD_LOGIC;
           cs: in STD_LOGIC;
           dataOut : out STD_LOGIC_VECTOR (31 downto 0));
end register32bits;

architecture Behavioral of register32bits is

signal data : std_logic_vector(31 downto 0) := (others => '0');

begin

data <= (others => '0') when reset = '1' else datain when rising_edge(clk) and read_write = '1' and cs = '1';
dataOut <= data when cs = '1' else (others => '0');

end Behavioral;
```

Figura 5.1: Código de registro básico de 32 bits

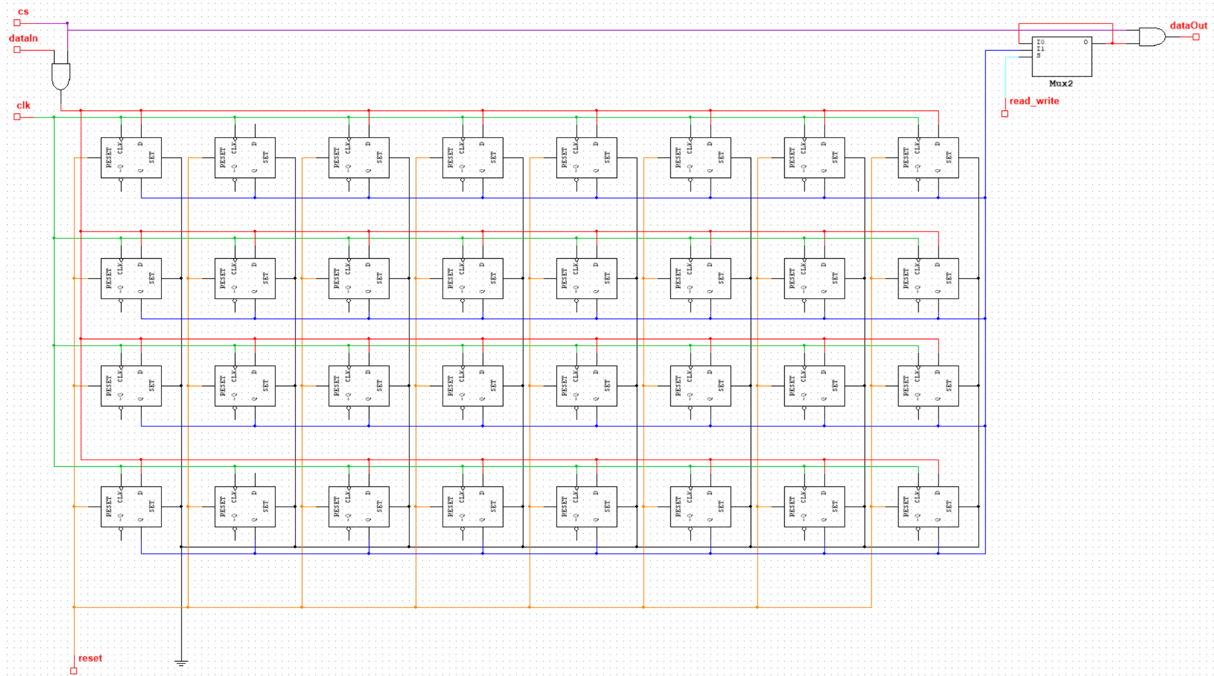

Figura 5.2 : Diseño registro 32 bits

En caso de que reset esté activo, el valor almacenado pasa a ser 0.

A data se le asocia los valores entrantes desde "dataIn" cuando "reset" no está activo, está en modo escritura, el chip está activado y hay un flanco de subida en el reloj. En caso contrario, no se cambia el valor de data.

La salida de datos proporcionará el valor de data cuando el chip esté activo, y cuando no, proporcionará un valor de 0 como valor por defecto.

Por último, es importante destacar que en el diseño realizado (Figura 5.2), la entrada "dataIn" y la salida "dataOut" son buses de datos. Se ha decidido simplificar la representación en el diseño para ayudar a la legibilidad.

La operación *AND* de "dataIn" y "cs" se hace en cada bit, y cada bit de la entrada va a su respectivo biestable tipo D. Lo mismo se puede decir de la salida, cada bit va de su respectivo biestable al multiplexor.

Este registro se usa en el banco de registros y en los chips de memoria. También se ha utilizado en varios módulos un decodificador de 4 bits. (Figura 5.3)

```
entity four_bit_decoder is
    Port ( cs : in STD_LOGIC;
           input : in STD_LOGIC_VECTOR (3 downto 0);
           output : out STD_LOGIC_VECTOR (15 downto 0));
end four_bit_decoder;

architecture decoder of four_bit_decoder is

begin

output(0) <= cs and not input(0) and not input(1) and not input(2) and not input(3);
output(1) <= cs and input(0) and not input(1) and not input(2) and not input(3);
output(2) <= cs and not input(0) and input(1) and not input(2) and not input(3);
output(3) <= cs and input(0) and input(1) and not input(2) and not input(3);
output(4) <= cs and not input(0) and not input(1) and input(2) and not input(3);
output(5) <= cs and input(0) and not input(1) and input(2) and not input(3);
output(6) <= cs and not input(0) and input(1) and input(2) and not input(3);
output(7) <= cs and input(0) and input(1) and input(2) and not input(3);
output(8) <= cs and not input(0) and not input(1) and not input(2) and input(3);
output(9) <= cs and input(0) and not input(1) and not input(2) and input(3);
output(10) <= cs and not input(0) and input(1) and not input(2) and input(3);
output(11) <= cs and input(0) and input(1) and not input(2) and input(3);
output(12) <= cs and not input(0) and not input(1) and input(2) and input(3);
output(13) <= cs and input(0) and not input(1) and input(2) and input(3);
output(14) <= cs and not input(0) and input(1) and input(2) and input(3);
output(15) <= cs and input(0) and input(1) and input(2) and input(3);

end decoder;
```

Figura 5.3 : Código de decodificador de 4 bits

En el decodificador se ha aplicado el diseño de flujo de datos, definiendo cómo es la salida a partir de la interacción de las entradas. Para que el decodificador de un valor, tiene que estar activa la entrada de "chip select", si es así, se activa un canal cuyo "número" es el correspondiente a la entrada.

Este tipo de diseño se traduce más fácilmente a un circuito. (Figura 5.4)

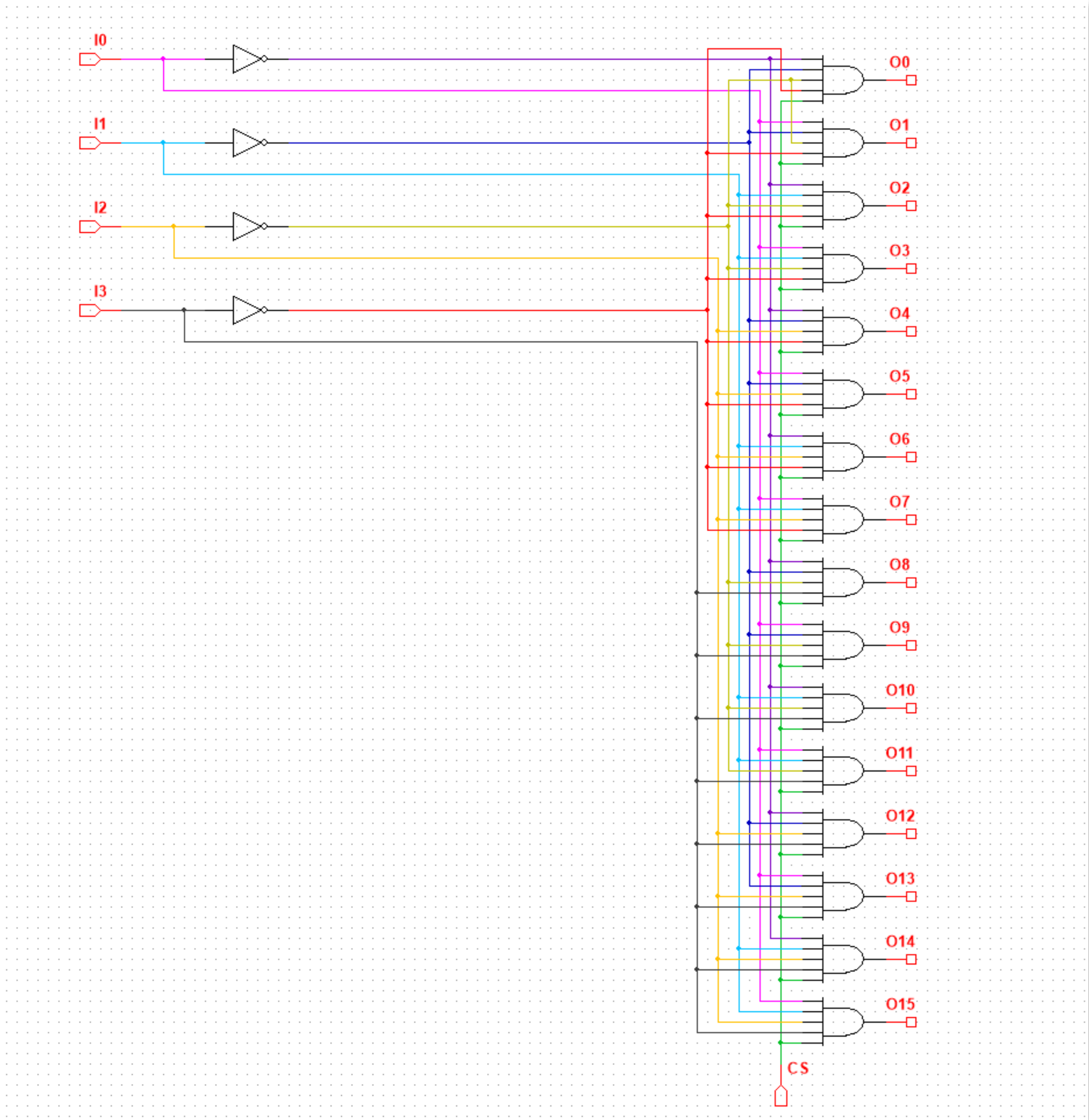

Figura 5.4 : Diseño decodificador

5.2.1. Banco de registros

Para poder dar la entrada a la ALU con los valores de los registros hay 2 entradas paralelas que representan el número o la dirección del registro y hay 2 salidas paralelas de datos. Para poder usar sólo un registro de los 2 disponibles en todo momento, se ha añadido una segunda entrada “chip select”. (Figura 5.5)

```
architecture registerBank of registerbank is

    type registers_output_t is array (0 to 15) of std_logic_vector (31 downto 0);
    signal decoded_cs,decoded_cs2,cs_together,initial_vector : std_logic_vector (15 downto 0);
    signal registers_output: registers_output_t;


begin

decoder: entity four_bit_decoder port map(cs => cs, input => registerN, output => decoded_cs);
decoder2: entity four_bit_decoder port map(cs => cs2, input => registerN2, output => decoded_cs2);

cs_together <= decoded_cs or decoded_cs2;

register0 : entity register32bits port map(
            dataIn => x"00000000",
            reset => reset,
            read_write => read_write,
            cs => cs_together(0),
            clk => clk,
            dataOut => registers_output(0));

bank:
    for i in 1 to 15 generate
        registers32bits : entity register32bits port map(
            dataIn => data_in,
            reset => reset,
            read_write => read_write,
            cs => cs_together(i),
            clk => clk,
            dataOut => registers_output(i));
    end generate;

r1_out <= registers_output(to_integer(unsigned(registerN)));
r2_out <= registers_output(to_integer(unsigned(registerN2)));

end registerBank;
```

Figura 5.5 : Código de banco de registros

Los parámetros “rs1” y “rs2” de las instrucciones son introducidos por “registerN” y “registerN2” respectivamente, y el parámetro “rd” también va a “registerN”. Los dos parámetros de dirección se introducen a 2 decodificadores de 4 bits, y la salida determina la entrada de “chip select” de los registros.

Se ha juntado las salidas de los decodificadores para simplificar la lógica. Por último, dependiendo de la entrada de “registerN” y “registerN2”, la salida del banco de “r1” y “r2”, es la salida de los registros con dicha dirección.

El registro *x0* se ha separado en el código y se ha creado fuera del *generate* para poder asegurar que su valor sea 0 siempre, fijando su entrada de datos a tierra. (Figura 5.6) Una forma alternativa de asegurar el valor 0 en el registro es simplemente dejar la entrada reset de ese registro como activado.

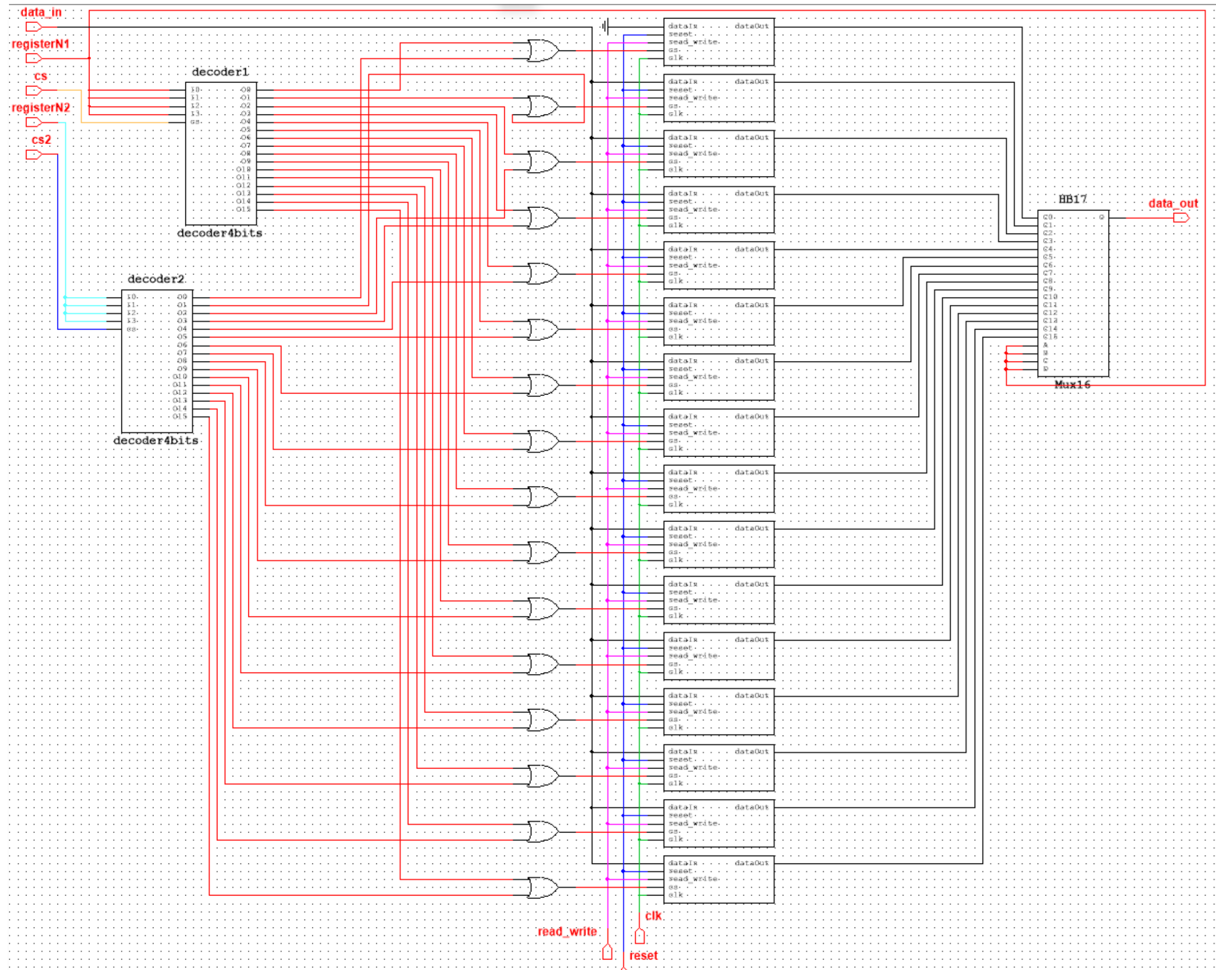

Figura 5.6 : Diseño banco de registros

Cabe destacar que, de la misma forma que se hizo con el diseño del registro (Figura 5.2), se ha decidido simplificar el diseño. En las entradas de los decodificadores y del multiplexor se han introducido las señales de “registerN” y “registerN2” bit a bit, no significa que a todas las entradas vaya la misma señal.

5.2.2. Contador de programa

El contador de programa en un principio se había implementado como un registro más en el banco de registros y se modificaba su valor cuando fuese necesario. En la última implementación se parece más a un contador básico con una entrada que permite modificar el valor inicial.

Por motivos de simplificar la lógica del MISC se ha decidido separar el contador de programa y me he basado en el registro de 32 bits básico para hacer el contador. (Figura 5.7)

```
entity pc is
    Port ( addr : in STD_LOGIC_VECTOR (7 downto 0);
           reset : in STD_LOGIC := '0';
           read_write : in STD_LOGIC;
           clk : in STD_LOGIC;
           cs : in STD_LOGIC;
           addr_out : out STD_LOGIC_VECTOR (7 downto 0));
end pc;

architecture register_pc of pc is

signal new_addr : std_logic_vector(7 downto 0) := (others => '0');

begin

new_addr <= "00000000" when reset = '1' else
            addr when cs = '1' and read_write = '1' else
            new_addr + 4 when rising_edge(clk) and cs = '1' and read_write = '0';

addr_out <= new_addr;

end register_pc;
```

Figura 5.7 : Código de contador de programa

El contador cuenta con una señal de 32 bits que es donde se calcula y almacena la siguiente dirección de la instrucción a la que se va a acceder.

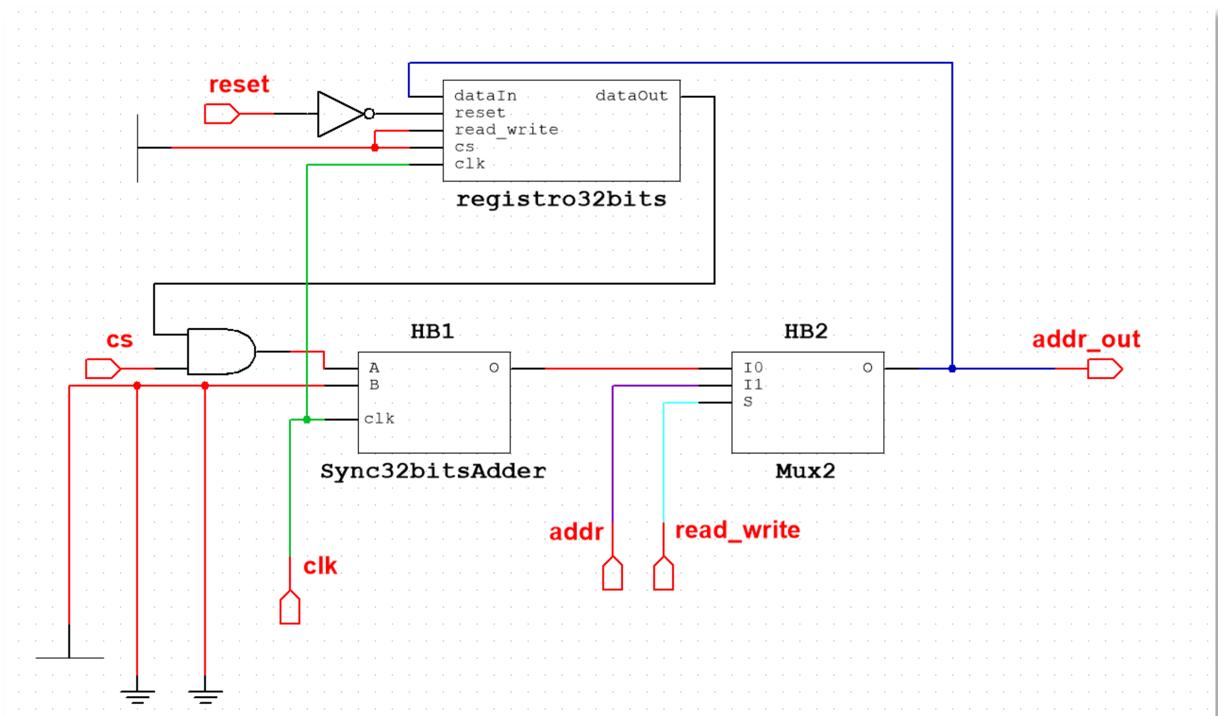

Figura 5.8 : Diseño contador de programa

De nuevo, el diseño se ha simplificado (Figura 5.8), ya que "addr", "A" y "B" del sumador, "dataIn" y "dataOut" del registro y la salida del multiplexor son buses de 32 bits. La entrada "B" simboliza un valor de *x00000004*, con los dos bits menos significativos unidos a tierra, y el tercer bit a la entrada positiva.

5.2.3. Memoria

En un principio la memoria se diseñó unificada la memoria de instrucciones y la memoria de datos, pero más adelante se modificó y se optó por la arquitectura Harvard, ya que simplificaba las conexiones entre la memoria y la unidad de control. De todas formas, las dos memorias usan el mismo chip. (Figura 5.9)

```
    type registers_output_t is array (0 to 63) of std_logic_vector (31 downto 0);
    signal decoded_cs : std_logic_vector (63 downto 0);
    signal registers_output: registers_output_t;
    signal cs0 : std_logic;
    signal cs1 : std_logic;
    signal cs2 : std_logic;
    signal cs3 : std_logic;
    signal initial_data : std_logic_vector (31 downto 0);
    signal initial_write : std_logic;
    signal initial_cs : std_logic_vector (63 downto 0);
begin

cs0<=(not addr(7)) and (not addr(6)) and cs;
cs1<=(not addr(7)) and (addr(6)) and cs;
cs2<=(addr(7)) and (not addr(6)) and cs;
cs3<=(addr(7)) and (addr(6)) and cs;

initial_cs <= decoded_cs when initial = '0' else x"FFFFFFFFFFFFFFFF" when initial = '1';

initial_data <= data_in when initial = '0' else x"10101010" when initial = '1';
initial_write <= initial or read_write;

--Vamos a ignorrar los dos últimos bits de la dirección, porque siempre vamos a tratar words de 32 bits
decoder0 : entity four_bit_decoder port map(cs=>cs0,input=>addr(5 downto 2), output=>decoded_cs(15 downto 0));
decoder1 : entity four_bit_decoder port map(cs=>cs1,input=>addr(5 downto 2), output=>decoded_cs(31 downto 16));
decoder2 : entity four_bit_decoder port map(cs=>cs2,input=>addr(5 downto 2), output=>decoded_cs(47 downto 32));
decoder3 : entity four_bit_decoder port map(cs=>cs3,input=>addr(5 downto 2), output=>decoded_cs(63 downto 48));

--Sólo vamos a usar los últimos 5 bits dirección
memory:
    for i in 0 to 63 generate
        registers32bits : entity register32bits port map(
            dataIn => initial_data,
            reset => reset,
            read_write => initial_write,
            cs => initial_cs(i),
            clk => clk,
            dataOut => registers_output(i));
    end generate;

output <= registers_output(to_integer(shift_right(unsigned(addr),2))) when cs = '1' else x"00000000";
end memory;
```

Figura 5.9: Código de módulo de memoria

Se ha decidido que para la demostración del funcionamiento se crearán 256 bytes de memoria, para eso se han creado 64 registros de 32 bits. También se ha aportado una entrada de inicialización, que cuando se activa proporciona un valor predeterminado inicial a todos los registros, y cuando se desactiva funciona de forma normal. Una vez acabadas las pruebas, se eliminará esta entrada.

Como sólo se accede de palabra en palabra, y no hay posibilidad de acceder a los bytes independientes, se ha codificado para ignorar los dos últimos bits de la dirección de memoria, por lo que se redondea a la dirección menor que es múltiplo de 4 más cercana.

En las memorias se ha seguido la misma estrategia que en el banco de registros (ver Figura 5.5 y Figura 5.10), la dirección se ha introducido, en este caso a 4 decodificadores debido al número de registros en la memoria.

Como entrada de "chip select" de estos decodificadores se tienen los dos primeros bits de la dirección de memoria, y así se consigue no tener que hacer un decodificador de 6 bits y se reutiliza el decodificador de 4 bits.

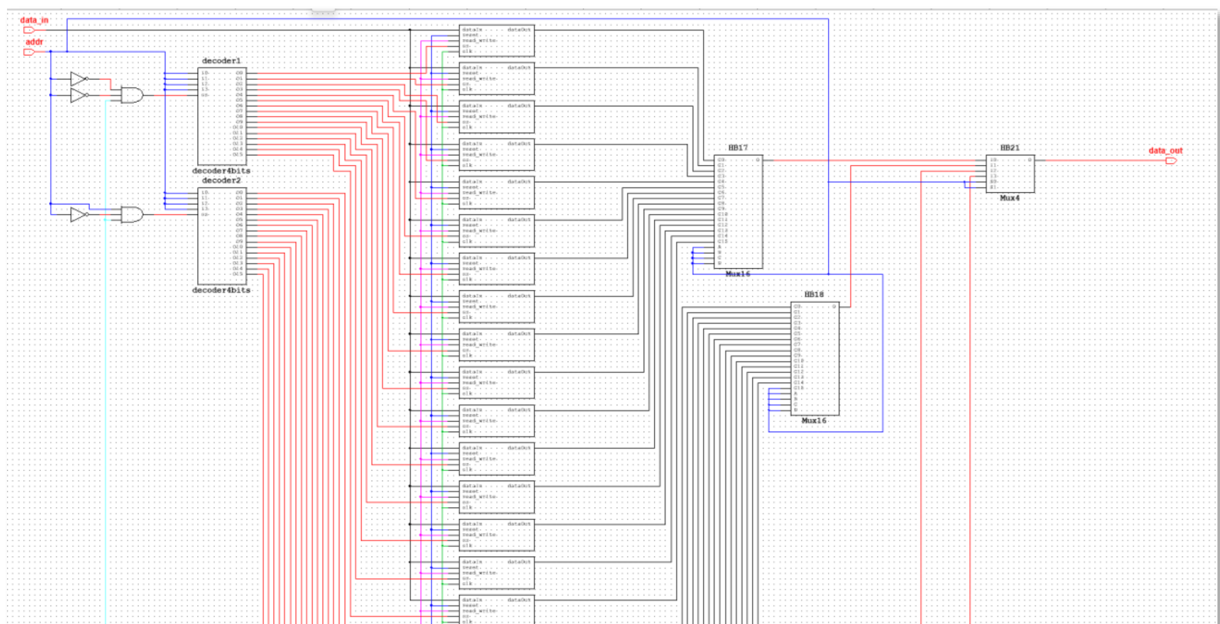

Figura 5.10 : Diseño parcial de la memoria

En el diseño se han usado las mismas simplificaciones que en el banco de registros (ver Figura 5.6). Las entradas de "chip select" de los decodificadores son las operaciones de *NOT* y *AND* de los dos bits más significativos, y las entradas a decodificar, los 4 bits siguientes.

Algo similar se ha diseñado con los multiplexores de la salida: los 5 bits menos significativos recogen una entrada por multiplexor de las 16 que tenía cada multiplexor, resultando en 4 entradas del último multiplexor, que, usando los 2 bits más significativos, escoge una salida.

El diseño no contiene los 64 registros que tiene la implementación porque es una referencia de cómo se ha diseñado.

5.2.4. ALU

La ALU tiene como enterada los dos registros y la instrucción completa para saber qué operación debe hacer. Para esto se podría haber puesto como entrada los parámetros “funct3”, “funct7” y “opcode” de la instrucción, pero se ha optado por introducir la instrucción completa para poner menos entradas de datos. (Figura 5.11)

```
architecture Behavioral of ALU is

signal result : std_logic_vector (31 downto 0);
signal funct3 : std_logic_vector(2 downto 0);

begin

funct3 <= instruction(14 downto 12) or "000";

--  SLT, SUB, AND, XOR, ADD, SLR, SLL
result <=   (r0 and r1) when funct3 = "111" else --AND
            (r0 xor r1) when funct3 = "100" else --XOR
            std_logic_vector(shift_right(unsigned(r0), to_integer(unsigned(r1(4 downto 0))))) when funct3 = "101" else -- SLR
            std_logic_vector(shift_left(unsigned(r0), to_integer(unsigned(r1(4 downto 0))))) when funct3 = "001" else -- SLL
            (r0 + r1) when ((funct3 = "000") and (instruction(30) = '0')) else --ADD
            (r0 - r1) when ((funct3 = "000") and (instruction(30) = '1')) else --SUB
            x"00000001" when funct3 = "010" and r0 < r1 else --SLT 1 output
            x"00000000" when funct3 = "010" and (r1 < r0 or r0 = r1); --SLT 0 output

output <= result;

zero <= '1' when result = x"00000000" else '0';

ow <=   '1' when r0(31) = '0' and r1(31) = '0' and result(31) = '1' and ((funct3 = "000") and (instruction(30) = '0')) else --(r0 + r1) = -
        '1' when r0(31) = '1' and r1(31) = '1' and result(31) = '0' and ((funct3 = "000") and (instruction(30) = '0')) else ---r0 + (-r1) = +
        '1' when r0(31) /= r1(31) and r1(31) = result(31) and ((funct3 = "000") and (instruction(30) = '1')) else '0'; --r0 - (-r1) = - || (-r0) - r1 = +

end Behavioral;
```

Figura 5.11: Código de ALU

Se han incluido también las salidas “zero” y “ow”, que se activan con una salida de 0 y con *overflow* respectivamente. En el MISC no se llegan a utilizar, pero se han implementado en caso de que en un desarrollo futuro se use.

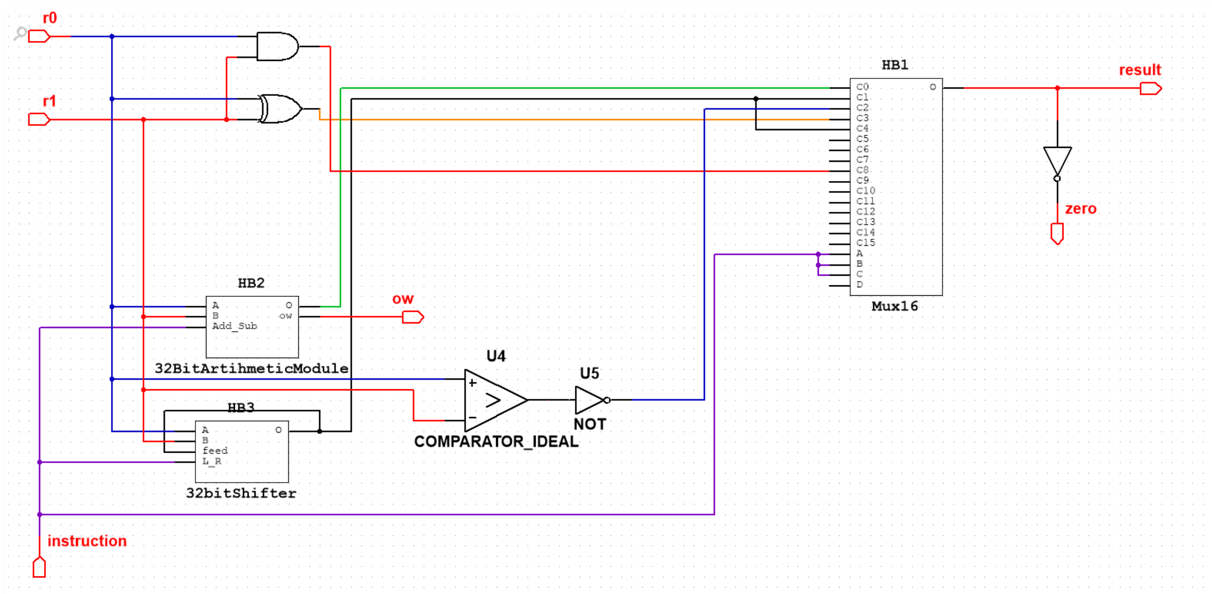

Figura 5.12 : Diseño ALU

En el diseño de la ALU (Figura 5.12) se ha simplificado las entradas “r0”, “r1”, instrucción y la salida “result”. La operación *AND* se hace bit a bit, de la misma forma se hace la operación *XOR*.

Por otra parte, en el sumador/restador, la operación se hace usando el valor entero de en una sola operación. Su entrada “Add_Sub” señaliza la operación a hacer: *ADD* si el bit 30 de la instrucción es 0, y *SUB* en caso contrario.

También se puede encontrar un módulo de desplazamiento bit a bit con entrada de realimentación y selección de modo izquierda/derecha dado por el bis más significativo de “funct3”.

El comparador que se encuentra en el diseño simplemente compara el valor de la entrada positivo con la entrada negativa, y si es mayor, la salida se activa, es decir, la salida se activa cuando el positivo es mayor. Como queremos exactamente lo contrario, simplemente se añade una puerta *NOT* después.

Por último, los valores de “funct3” son la entrada que determina la salida del multiplexor de salida.

5.2.5. Unidad de control

La unidad de control se puede entender como que tiene 2 partes: las salidas, y la generación de los inmediatos.

En la primera parte se encuentra un contador de 3 bits, cuyos bits se han usado para marcar las distintas fases de las instrucciones siendo el bit más significativo el que marca que la instrucción todavía está en ejecución, y el segundo bit es el que marca la fase de escritura. El bit menos significativo marca los 2 ciclos que cuesta hacer cada fase de lectura/escritura. (Figura 5.13)

El contador inicia en un valor de “111”, y cuenta de uno en uno de forma incremental cuando hay un flanco de subida en el reloj y se están ejecutando instrucciones del tipo R, I o S, es decir, únicamente no se usa el contador cuando se está ejecutando un *JAL*.

Como el bit más significativo del contador marca el estado de ejecución de la instrucción, se puede usar como la señal de “chip "” del contador de programa, ya que mientras se ejecuta una instrucción no queremos que avance el contador de programa porque no tenemos un pipeline.

Como el segundo bit más significativo marca la operación de escritura, se puede usar para la señal de lectura escritura de los registros y de la memoria, así como su señal de "chip select" y la salida "registerN1".

En la primera parte, también se recoge la instrucción que llega desde la memoria de instrucciones en el momento que se activa el contador de programa. Se envía esa instrucción almacenada a la *ALU*, y un ciclo antes de la etapa de almacenamiento, se guarda el resultado que llega de la *ALU* si es una instrucción tipo R.

El "chip select" del contador de programa se activa con el bit 2 del contador o con el *flag* de escritura de la memoria de instrucciones y así poder programar el procesador.

La salida "addr_out", se establece como la dirección de memoria a la que se va a acceder o se va a saltar dependiendo de la instrucción a ejecutar. Si es una instrucción tipo I, la dirección es la especificada en la instrucción, si es tipo S, la dirección es la del inmediato S, y si es tipo J, es la del inmediato J.

El "chip select" de la memoria se activa en la fase de escritura de las instrucciones de tipo I y tipo S.

Los modos de lectura escritura en el contador de programa, en la memoria y en los registros se establecen con una instrucción tipo J; cuando el *flag* de escritura de la memoria de instrucciones está activo o es una instrucción tipo S; y cuando se está en fase de escritura y es una instrucción tipo R o tipo I, respectivamente.

La salida "resgisterN1" aporta el valor de rd cuando está en la fase de escritura, el valor de "rs1" cuando está en la fase de obtención de datos, y 0 en cualquier otro caso. El "chip select" del registro 1 está activo cuando la instrucción a ejecutar es de tipo R o tipo I.

Mientras tanto, la salida "registerN2" proporciona rs2 si la instrucción almacenada es tipo R o tipo S, y en otro caso es 0. Su "chip select" es 1 únicamente en la fase de recuperación de datos o durante la ejecución de una instrucción tipo S.

Se envía el resultado de la ALU al banco de registros cuando la instrucción es del tipo R, y si es de tipo I o J se envían los datos recibidos de memoria.

```vhdl
instruction_reg <= instruction when counter(2) = '1' and program_memory = '0' else instruction_reg;

instruction_out <= instruction_reg;
result <= alu_result when instruction_reg(6 downto 0) = "0110011" and counter(0) = '1' and counter(1) = '0' else result;


addr_out <= instruction_reg(27 downto 20) when instruction_reg(6 downto 0) = "0000011" else
            inm_s(7 downto 0) when instruction_reg(6 downto 0) = "0100011" else
            inm_j(7 downto 0) when instruction_reg(6 downto 0) = "1101111" else x"00";

cspc<= counter(2) or program_memory;
cs_mem <= '1' when counter(1) = '1' and counter(2) = '0' and (instruction_reg(6 downto 0) = "0000011" or instruction_reg(6 downto 0) = "0100011") else '0';


read_write_pc <= '1' when instruction_reg(6 downto 0) = "1101111" else '0';
read_write_mem <= '1' when program_memory = '1' or instruction_reg(6 downto 0) = "0100011" else '0';
read_write_regs <= '1' when counter(0) = '0' and counter(1) = '1' and (instruction_reg(6 downto 0) = "0110011" or instruction_reg(6 downto 0) = "0000011") else '0';

registerN1 <= instruction_reg(10 downto 7) when counter(1) = '1' and counter(0) = '0' else instruction_reg(18 downto 15) when counter(1) = '0' else "0000";
cs_register1 <= '1' when instruction_reg(6 downto 0) = "0110011" or instruction_reg(6 downto 0) = "0000011" else '0';

registerN2 <= instruction_reg(23 downto 20) when instruction_reg(6 downto 0) = "0110011" or instruction_reg(6 downto 0) = "0100011" else "0000";
cs_register2 <= '1' when (counter(1) = '0' and counter(2) = '0') or instruction_reg(6 downto 0) = "0100011"  else '0';

data_for_register <= result when instruction_reg(6 downto 0) = "0110011" else
                     data_from_memory when instruction_reg(6 downto 0) = "0000011" or instruction_reg(6 downto 0) = "1101111" else
                     (others=> '0');

counter <= "111" when rising_edge(counter(2)) or
           not ((instruction_reg(6 downto 0) = "0110011") or (instruction_reg(6 downto 0) = "0000011") or (instruction_reg(6 downto 0) = "0100011")) else
           counter + 1 when rising_edge(clk);
```

Figura 5.13 : Señales de la unidad de control

Como se puede observar, se ha decidido que las operaciones de memoria (*JAL*, *SW* y *LW*) se ejecuten en la unidad de control, debido a que no se hace ninguna operación aritmética o lógica y simplificaba significativamente la lógica.

La segunda parte de la unidad de control se compone de la construcción de los dos inmediatos utilizados en el MISC según su especificación en la ISA. (Figura 5.14) (ver Tabla 5.1)

```
inm_j<= (19 => instruction_reg(31),
         18 => instruction_reg(19),
         17 => instruction_reg(18),
         16 => instruction_reg(17),
         15 => instruction_reg(16),
         14 => instruction_reg(15),
         13 => instruction_reg(14),
         12 => instruction_reg(13),
         11 => instruction_reg(12),
         10 => instruction_reg(20),
         9 => instruction_reg(30),
         8 => instruction_reg(29),
         7 => instruction_reg(28),
         6 => instruction_reg(27),
         5 => instruction_reg(26),
         4 => instruction_reg(25),
         3 => instruction_reg(24),
         2 => instruction_reg(23),
         1 => instruction_reg(22),
         0 => instruction_reg(21));

inm_s <= (11 => instruction_reg(31),
          10 => instruction_reg(30),
          9 => instruction_reg(29),
          8 => instruction_reg(28),
          7 => instruction_reg(27),
          6 => instruction_reg(26),
          5 => instruction_reg(25),
          4 => instruction_reg(11),
          3 => instruction_reg(10),
          2 => instruction_reg(9),
          1 => instruction_reg(8),
          0 => instruction_reg(7));
```

Figura 5.14: Construcción de inmediatos en la unidad de control

5.3. Diseño MISC

El chip tiene 5 entradas que permiten programar y reiniciar la memoria y los datos de los registros.

Se recomienda reiniciar el contador, poniendo un flanco de subida en la entrada "reset_pc", antes y después de programar la memoria y así conseguir una facilidad mayor de programación. También conviene antes de programar el dispositivo, borrar los datos de memoria, esto se hace activando la entrada de "reset". (Figura 5.15)

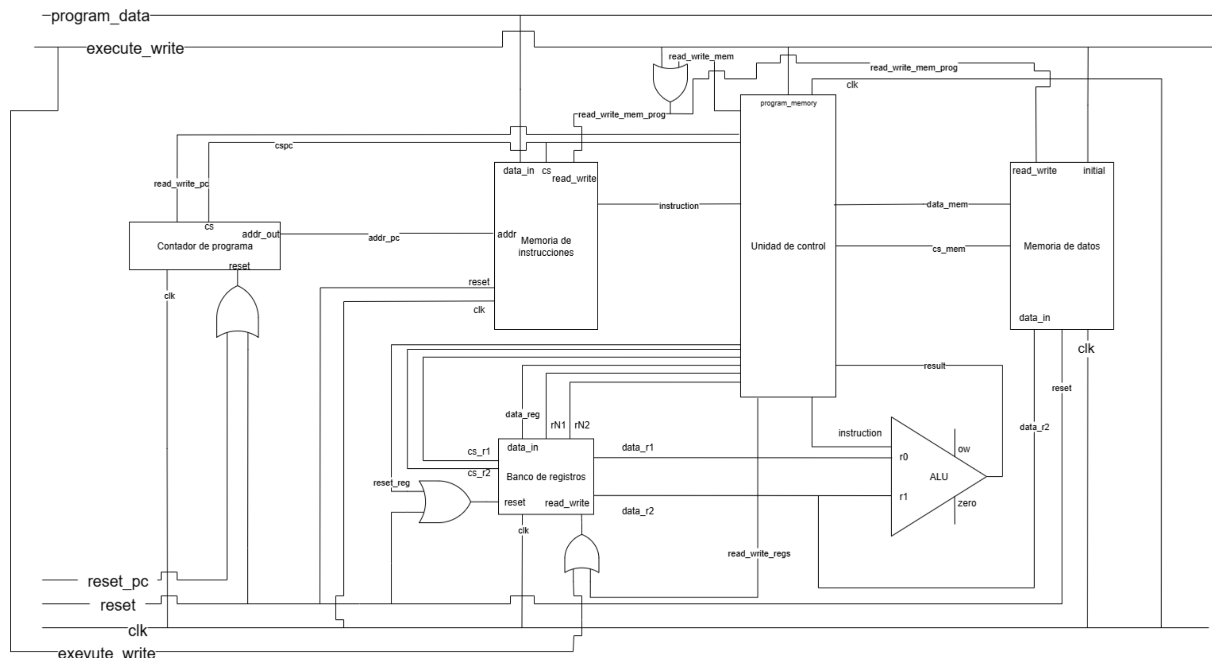

Figura 5.15 : Conexión de los módulos

La entrada "program_data" es por donde se introducen las instrucciones a ejecutar, y van directamente al banco de memoria de instrucciones a la dirección de memoria dada por el contador de programa.

También es necesario que cuando se estén introduciendo las instrucciones se active el *flag* de "execute_write", para así indicar que se está programando el dispositivo. Al desactivar el *flag* "execute_write" el procesador pasará a la fase de ejecución.

Al entrar en la fase de ejecución, el contador de programa pondrá en su salida de "addr_out" la dirección de memoria a la que accede en la memoria de instrucciones (por esta razón se recomienda reiniciar el contador, aunque se puede programar en la dirección *x00* un *JAL* a la dirección donde se va a escribir el programa).

Esta memoria pondrá en su salida de datos la instrucción que está en dicha dirección y se introducirá a la unidad de control.

En la unidad de control se guarda esta instrucción y dependiendo de qué tipo de instrucción es se hace una cosa u otra:

- Si es una instrucción tipo I, se activa la memoria de datos y se introduce la dirección de memoria donde está la palabra a recuperar. También se activa el contador para sincronizar toda la acción. Una vez recuperado el dato de memoria, se establece la salida "registerN1" que determinará qué registro es el destino.
- Si es tipo S, se activa la memoria de datos y se introduce la dirección de memoria del inmediato en la salida de dirección a memoria, a la vez que se activa el contador para sincronizar la operación. Se activa el "registro2" y se establece cuál es el "registro2" a partir de la instrucción. Por último, se establece el modo de escritura a la memoria.
- Si el tipo de instrucción es J, se activa el *flag* de escritura del contador de programa, y la dirección de salida al contador es la introducida como inmediato.
- Por último, si es tipo R, se activan los 2 registros, se ponen en modo lectura, la salida de los dos registros se envía a la ALU, y se activa el contador para la sincronía. El resultado de la ALU posteriormente vuelve a la unidad de control, que posteriormente se desactiva el registro 2 y se establece el "registro1" como el valor que estaba en el parámetro "rd" de la instrucción.

6. Resultados

6.1. Especificaciones

El procesador diseñado cuenta con:

- 16 registros generales, de los cuales el valor del registro *x0* está fijado a 0.
- 2048 bits de almacenamiento de instrucciones y datos, ambos cuentan con 64 registros con acceso de palabra completa.
- Entrada de reset general, y otra de reset del contador de programa.
- Entrada de datos del programa.
- 10 instrucciones de las cuales:
    - 5 instrucciones lógicas: *AND*, *XOR*, *SLR*, *SLL*, *SLT*
    - 2 instrucciones aritméticas: *ADD*, *SUB*
    - 2 instrucciones de gestión de memoria: *LW, SW*
    - 1 instrucción de salto: *JAL*
- La velocidad máxima de ejecución está todavía por medirse.

6.2. Pruebas unitarias

Con la finalidad de comprobar el correcto funcionamiento del proyecto, se han ido implementando una serie de pruebas a los distintos elementos diseñados y programados del procesador.

Como se hacían pruebas a cada elemento creado conforme se iban implementando, se ha seguido una filosofía *bottom-up*.

6.2.1. Prueba registro básico

El primer elemento del que se ha comprobado el correcto funcionamiento ha sido del registro básico de 32 bits. (Figura 6.1)

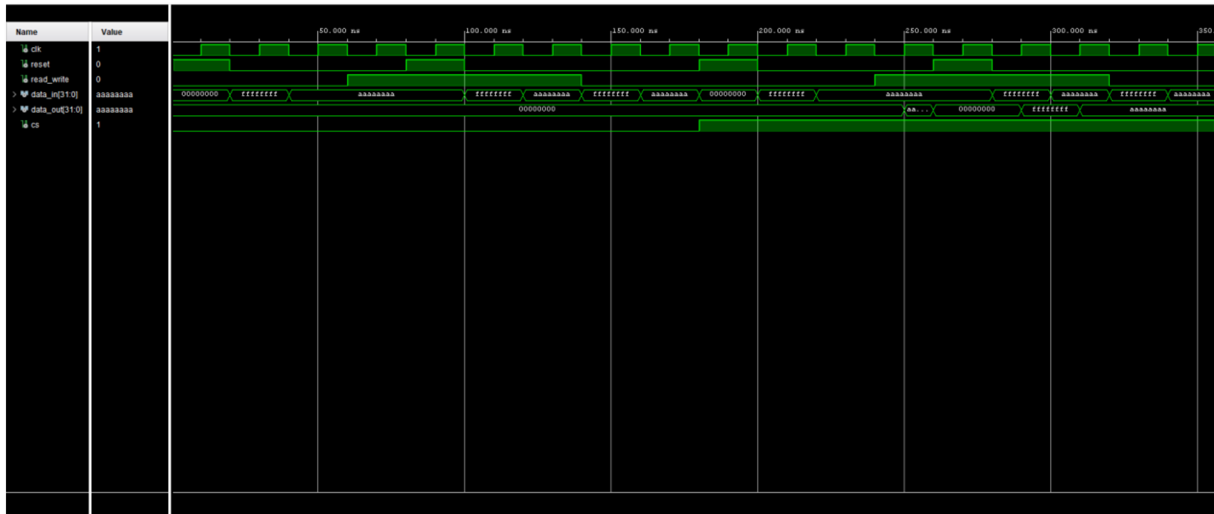

Figura 6.1: Prueba de registro básico

En este test se ha querido comprobar que hasta que no se activa el *flag* "chip select", el registro no aporta ninguna salida. También se quería comprobar que mientras el reset esté activo, la salida cambie a un valor de 0 de forma asíncrona. (Figura 6.3)

Por último, se ha comprobado que la salida de datos sólo cambia cuando se introduce un valor distinto por la entrada de datos, está en modo escritura, el *flag* de "reset" está desactivado, el chip está activado y hay un flanco de subida en el reloj.

```
clk_generation: process
begin
    clk <= not clk after 10ns;
    wait for 10ns;
end process;
```

Figura 6.2 : Código generación de reloj

```
read_write <= '0';
reset <= '1';
data_in <= (others => '0');
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

read_write <= '1';
wait for 20 ns;
reset <= '1';
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

read_write <= '0';
data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

cs <= '1';
read_write <= '0';
reset <= '1';
data_in <= (others => '0');
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;
```

Figura 6.3: Código prueba registro 32 bits

6.2.2. Prueba decodificador 4 bits

El siguiente componente básico para el que se ha diseñado una prueba ha sido el decodificador de 4 bits. (Figura 6.4)

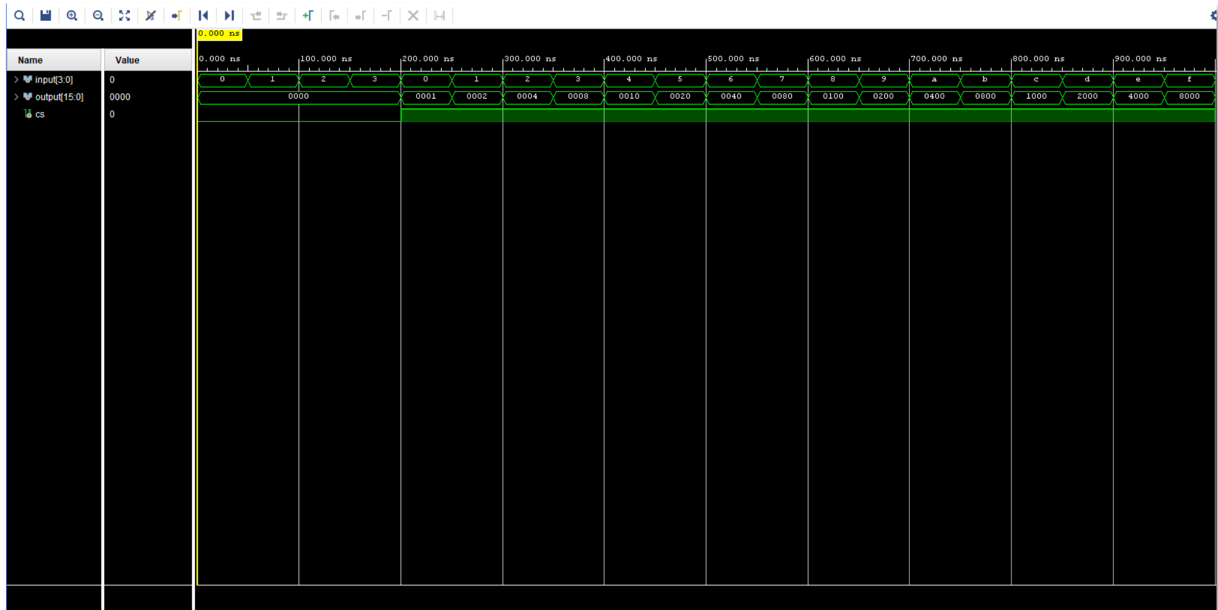

Figura 6.4: Prueba de decodificador de 4 bits

En este test se ha buscado comprobar que el codificador hace su trabajo de “seleccionar” la salida correcta a partir de la entrada especificada, y que únicamente lo hace cuando el “chip select” está activo. (Figura 6.5)

```
input <= x"0";
wait for 50 ns;
input <= x"1";
wait for 50 ns;
input <= x"2";
wait for 50 ns;
input <= x"3";
wait for 50 ns;

cs <= '1';

input <= x"0";
wait for 50 ns;
input <= x"1";
wait for 50 ns;
input <= x"2";
wait for 50 ns;
input <= x"3";
wait for 50 ns;
input <= x"4";
wait for 50 ns;
input <= x"5";
wait for 50 ns;
input <= x"6";
wait for 50 ns;
input <= x"7";
wait for 50 ns;
input <= x"8";
wait for 50 ns;
input <= x"9";
wait for 50 ns;
input <= x"A";
wait for 50 ns;
input <= x"B";
wait for 50 ns;
input <= x"C";
wait for 50 ns;
input <= x"D";
wait for 50 ns;
input <= x"E";
wait for 50 ns;
input <= x"F";
wait for 50 ns;
```

Figura 6.5: Parte del código del banco de prueba del decodificador

6.2.3. Prueba banco registros

Tras comprobar el corroborar el correcto funcionamiento del registro básico y el decodificador, se ha probado el banco de registros. (Figura 6.6)

Figura 6.6: Prueba del banco de registros

El objetivo de este test es comprobar que se puede acceder de forma independiente a 2 registros, e incluso que se puede acceder al mismo registro desde la entrada "registerN" y "registerN2" y que el dato resultante sea el correcto. (Figura 6.7)

También se han hecho unas pequeñas pruebas para comprobar el correcto funcionamiento de las entradas de "chip select", "reset", y "read_write", y como ya se comprobó el funcionamiento en la prueba del registro básico (Figura 6.1), no se considera necesario mencionarlo en esta parte.

```
reset <= '1';
data_in <= (others => '0');
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

cs <= '1';

wait for 20 ns;
reset <= '1';
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

read_write <= '0';
data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

registern <= x"1";
read_write <= '0';
data_in <= x"FFFFFFFF";
wait for 20 ns;

read_write <= '1';
data_in <= x"11111111";
wait for 20 ns;

registern <= x"0";
read_write <= '0';
wait for 20ns;
```

Figura 6.7 : Parte del código de las pruebas del banco de registros

6.2.4. Prueba memoria

El siguiente módulo para el que se ha diseñado una prueba ha sido la memoria. Como la memoria de instrucciones y la memoria de datos se basan en la misma entidad, sólo se ha tenido que diseñar un banco de pruebas

Por naturaleza de la memoria, su testeo es similar al del banco de registros, sólo que con una entrada de elección de registro. (Figura 6.9)

La mayor diferencia es que, como se ha explicado anteriormente, se ha diseñado la memoria para que únicamente se pueda acceder a las palabras enteras registradas en cada dirección, por lo que, al introducir, por ejemplo, la dirección *x01*, en realidad se está accediendo a la dirección *x00*.

Esto se puede ver reflejado en el test, ya que, al acceder por primera vez a esa dirección, el valor almacenado es el que se había guardado anteriormente en *x00*. (Figura 6.8)

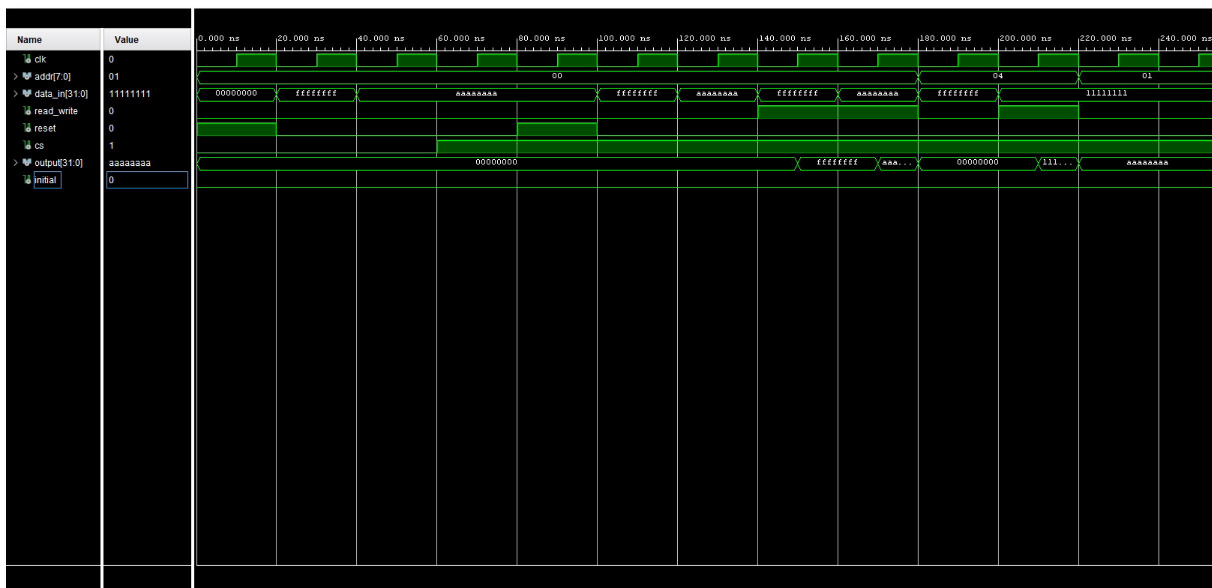

Figura 6.8: Prueba de memoria

Cabe destacar que en el acceso a *x01*, en un principio es incorrecto porque la salida del registro es *x00000000*, pero hay que recordar que el registro aporta su salida con un flanco de subida, y por lo tanto mientras tanto, al cambiar la dirección de acceso, la salida de memoria es *0x00000000*.

```
reset <= '1';
data_in <= (others => '0');
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

cs <= '1';

wait for 20 ns;
reset <= '1';
wait for 20 ns;
reset <= '0';

data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

read_write <= '1';
data_in <= x"FFFFFFFF";
wait for 20 ns;

data_in <= x"AAAAAAAA";
wait for 20 ns;

addr <= x"04";
read_write <= '0';
data_in <= x"FFFFFFFF";
wait for 20 ns;

read_write <= '1';
data_in <= x"11111111";
wait for 20 ns;

addr <= x"01";
read_write <= '0';
wait for 20ns;
```

Figura 6.9 : Parte del código del banco de pruebas de la memopria

6.2.5. Prueba ALU

Una vez corroborado que la memoria funciona correctamente, se ha diseñado el test de la ALU. (Figura 6.10)

La prueba se ha subdividido en 7, correspondiendo una parte a cada instrucción que se ejecuta en la ALU, siendo estas las instrucciones *AND*, *XOR, SRL, SLL, SLT, ADD* y *SUB*, siendo ese el orden de ejecución de instrucciones en el test. (Figura 6.11)

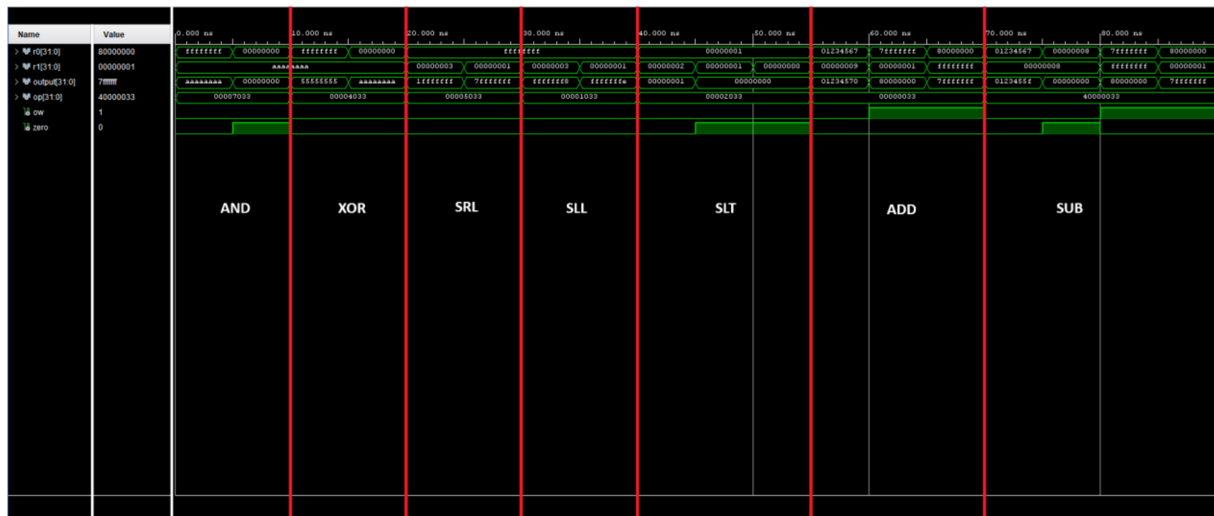

Figura 6.10: Prueba de ALU

Se ha querido comprobar que las operaciones lógicas se hacían correctamente, para eso no ha sido necesario hacer muchas pruebas porque son operaciones sencillas, sin embargo, con las operaciones aritméticas se ha querido comprobar que los *flags* de “overflow” y “zero” están bien implementados.

Se ha comprobado el correcto funcionamiento de estos *flags* con los 2 casos que cada instrucción produce un “overflow”, que es con la suma de dos positivos dando uno negativo, y la suma de dos negativos resultando en un positivo en el caso de *ADD*; mientras que para *SUB* se resta un negativo a otro negativo y resulta en un positivo, y la resta de positivo a un negativo, dando como resultado un positivo.

Por último, se ha restado un número a sí mismo, activándose el *flag* de “zero”, demostrando que el *flag* de “zero” funciona correctamente, aunque con las instrucciones *AND* y *SLT* ya se había visto que funcionaba correctamente.

```
op <= x"00007033"; -- AND
r0 <= x"FFFFFFFF";
r1 <= x"AAAAAAAA";
wait for 5 ns;

r0 <= x"00000000";
wait for 5 ns;

op <= x"00004033"; -- XOR
r0 <= x"FFFFFFFF";
r1 <= x"AAAAAAAA";
wait for 5 ns;

r0 <= x"00000000";
wait for 5 ns;

op <= x"00005033"; -- SRL
r0 <= x"FFFFFFFF";
r1 <= x"00000003";
wait for 5 ns;

r1 <= x"00000001";
wait for 5 ns;

op <= x"00001033"; -- SLL
r0 <= x"FFFFFFFF";
r1 <= x"00000003";
wait for 5 ns;

r1 <= x"00000001";
wait for 5 ns;

op <= x"00002033"; -- SLT
r0 <= x"00000001";
r1 <= x"00000002";
wait for 5 ns;

r1 <= x"00000001";
wait for 5 ns;

r1 <= x"00000000";
wait for 5 ns;
```

Figura 6.11 : Parte del código de banco de pruebas de la ALU

6.3. Prueba MISC

A la hora de diseñar la prueba del MISC, se ha modificado el código del banco de registros con la finalidad de que todos los registros tengan un valor por defecto.

Se han separado 3 registros que servirán de almacenamiento. (Figura 6.12)

```
register1 : entity register32bits port map(
            dataIn => data_in,
            reset => reset,
            read_write => read_write,
            cs => cs_together(1),
            clk => clk,
            dataOut => registers_output(1));

register2 : entity register32bits port map(
            dataIn => data_in,
            reset => reset,
            read_write => read_write,
            cs => cs_together(2),
            clk => clk,
            dataOut => registers_output(2));

register3 : entity register32bits port map(
            dataIn => data_in,
            reset => reset,
            read_write => read_write,
            cs => cs_together(3),
            clk => clk,
            dataOut => registers_output(3));

bank:
    for i in 4 to 15 generate
        registers32bits : entity register32bits port map(
            --dataIn => data_in,
            dataIn => std_logic_vector(to_unsigned(i, 32)),
            reset => reset,
            read_write => read_write,
            cs => cs_together(i),
            clk => clk,
            dataOut => registers_output(i));
    end generate;
```

Figura 6.12: Modificación del banco de registros para las pruebas

El banco de pruebas implementa la entidad que representa el MISC e incluye 2 procesos paralelos: la creación de la señal del reloj y generación de estímulos. (Figura 6.13 y Figura 6.16)

En el proceso de creación del reloj se invierte el estado del reloj cada 10ns, durante 10ns.

El proceso de generación de estímulos se subdivide en 2 partes: la programación del chip, y la ejecución del programa.

6.3.1. Ejecución de todas las instrucciones

```
miscriscv : entity misc_riscv port map
(clk=>clk, reset=>reset, reset_pc=>reset_pc, execute_write=>execute_write, program_data=> program_data);
clk_generation: process
begin
    clk <= not clk after 10ns;
    wait for 10ns;
end process;
stimulus_generation: process
begin
    program_data <= x"00000000";
    reset <= '1';
    reset_pc <= '1';
    execute_write <= '1';
    wait for 10ns;

    reset <= '0';
    reset_pc <= '0';
    program_data <= x"0084F0B3"; --AND r9,r8,r1
    wait for 20ns;
    program_data <= x"00A0C133"; --XOR r1,r10,r2
    wait for 20ns;
    program_data <= x"005110B3"; --SLL r2,r5,r1
    wait for 20ns;
    program_data <= x"0040D1B3"; --SRL r1,r4,r3
    wait for 20ns;
    program_data <= x"402180B3"; --SUB r2,r3,r1
    wait for 20ns;
    program_data <= x"003101B3"; --ADD r2,r3,r3
    wait for 20ns;
    program_data <= x"003120B3"; --SLT r2,r3,r1
    wait for 20 ns;
    program_data <= x"00300023"; --SW r3, 00
    wait for 20 ns;
    program_data <= x"00002103"; --LW r2, 00
    wait for 20 ns;
    program_data <= x"000000EF"; --JAL 00
    wait for 30ns;

    reset_pc <= '1';
    execute_write <= '0';
    wait for 5 ns;
    reset_pc <= '0';
    wait;
end process;
```

Figura 6.13: Banco de pruebas del procesador

En este caso se ha programado con una instancia de cada una de las instrucciones de la ISA, empezando por las instrucciones lógicas, luego las aritméticas, después las de gestión de memoria y por último la instrucción de salto.

El programa escrito simplemente toma los valores de los registros especificados, opera con ellos en las operaciones lógicas y aritméticas almacenándolo en un registro al final, que en este caso es el registro 1, para luego almacenar un valor de un registro, en este caso el 3, y a continuación recuperar ese mismo valor de la memoria de datos y almacenarlo en otro registro distinto, en este caso el 2. Por último, se salta al inicio, creando así un bucle infinito.

Como se puede comprobar, no es un programa complejo, pero se logra demostrar el funcionamiento de la arquitectura con la simulación. (Figura 5.14 y Figura 6.15)

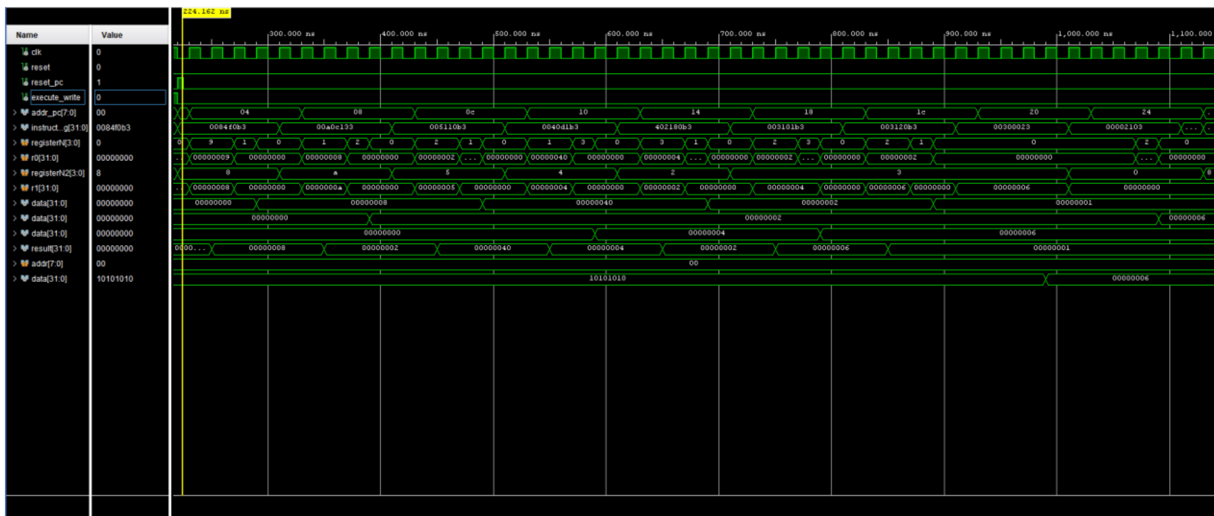

Figura 6.14: Simulación del banco de pruebas con 1 iteración

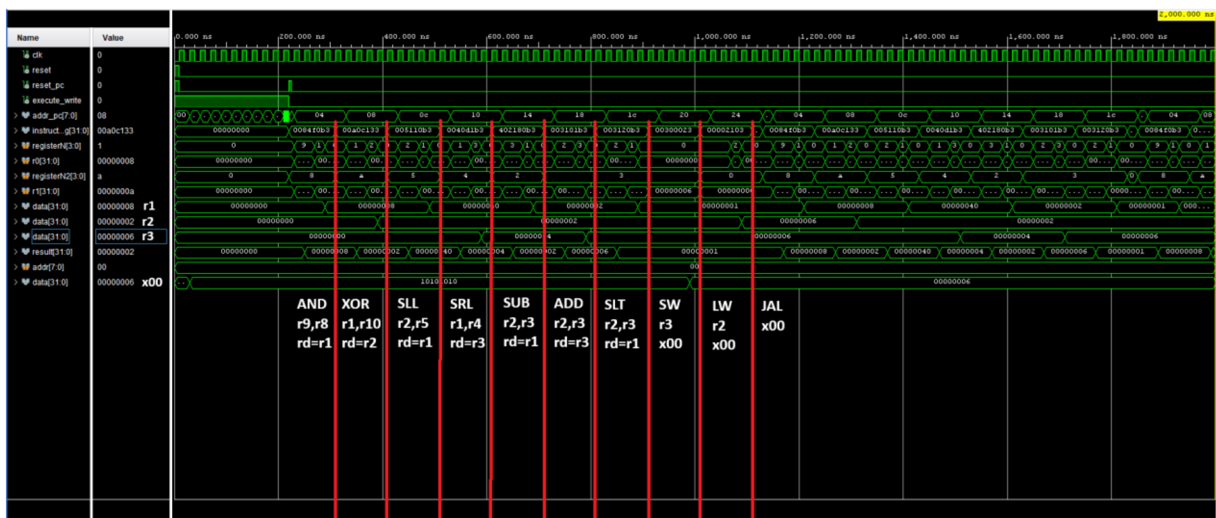

Figura 6.15: Simulación del banco de pruebas con 2 iteraciones

En la simulación podemos ver que la dirección del pc va cambiando (aumentando de 4 en 4 empezando desde 0, es decir, aumentando en una palabra por cada ciclo) tras el reseteo inicial. Ese cambio de dirección es utilizado para escribir las instrucciones en memoria.

Tras escribir la última instrucción, se reinicia de nuevo el contador de programa y se pone el MISC en modo ejecución.

En la simulación, además de las variables presentes en la entidad del MISC, se han añadido las variables: dirección almacenada en el contador de programa, el registro de la instrucción de la unidad de control, las entradas del banco de registros de “registerN” y “registerN2”, las entradas de la ALU de “r0” y “r1”, la señal data de los registros 1,2 y 3, la señal que almacena el resultado de la ALU en la unidad de control, la entrada de la dirección de acceso en la memoria de datos y la señal data de la dirección de memoria de datos accedida.

Podemos observar que una vez que se almacena la instrucción en la unidad de control, que en la simulación se ve en la variable “instrution_reg”, el contador de programa se para debido al contador de la unidad de control.

Tras eso, se pueden ver las 2 etapas de obtención de datos, por ejemplo, en las instrucciones de tipo R, donde el valor de la variable “registerN” en la primera etapa es el número del registro codificado en “rs1”, mientras que en la segunda etapa es el valor codificado en “rd”.

Otro aspecto que se puede apreciar, es el cómo se almacenan los datos en los registros o memoria en la etapa de escritura de las instrucciones.

Por último, cabe destacar que como *JAL* es la única instrucción que no usa el contador, una vez almacenada la instrucción, se ejecuta en el siguiente flanco de subida del reloj.

6.3.2. Programa de comprobación de overflow

Se ha creado un segundo banco de pruebas en el que el objetivo del programa es comprobar que el *flag* de "overflow" funciona correctamente. (Figura 6.16). Este banco de pruebas usa la misma generación de reloj y programa el procesador de la misma forma que el anterior (ver Figura 6.13), tan sólo cambia el código introducido a la memoria de instrucciones.

```
stimulus_generation: process
begin
    program_data <= x"00000000";
    reset <= '1';
    reset_pc <= '1';
    execute_write <= '1';
    wait for 10ns;

    reset <= '0';
    reset_pc <= '0';
    program_data <= x"0084F1B3"; --AND r9,r8,r3
    wait for 20ns;
    program_data <= x"00B44133"; --XOR r8,r11,r2
    wait for 20ns;
    program_data <= x"0021D133"; --SRL r3,r2,r2
    wait for 20ns;
    program_data <= x"402000B3"; --SUB r0,r2,r1
    wait for 20ns;
    program_data <= x"0020D0B3"; --SRL r1,r2,r1
    wait for 20ns;
    program_data <= x"002081B3"; --ADD r1,r2,r3
    wait for 20ns;
    program_data <= x"0021A1B3"; --SLT r2,r3,r3
    wait for 20 ns;
    program_data <= x"00300023"; --SW r3, 00
    wait for 30ns;

    reset_pc <= '1';
    execute_write <= '0';
    wait for 5 ns;
    reset_pc <= '0';
    wait;
end process;
```

Figura 6.16 : Banco de pruebas de programa de comprobación de overflow

Para la simulación, de nuevo, se ha usado el banco de registros modificado (ver Figura 6.12).

A partir de los valores en los que se inicializan, se ha pretendido primero obtener el valor 1 mediante la operación "AND r8,r9", dando como resultado *x8*, y paralelamente "XOR r8,r11", para obtener *x3*, y con ese valor hacer un desplazamiento lógico a la derecha al valor obtenido anteriormente (*x8*), dando como resultado *x1*; segundo obtener el valor positivo más alto, que se ha conseguido restando 1, que se ha obtenido con las instrucciones anteriores, al registro 0, cuyo valor es siempre *x0*, y luego desplazarlo un bit a la derecha para establecer el bit más significativo a un valor de 0, haciendo el número positivo; por último, sólo falta comprobar que el resultado de la suma es, efectivamente, menor que 0, porque al sumar al máximo de un entero 1, resulta en el valor mínimo de un entero, y luego se ha almacenado el resultado de la comparación en la dirección *x00*.

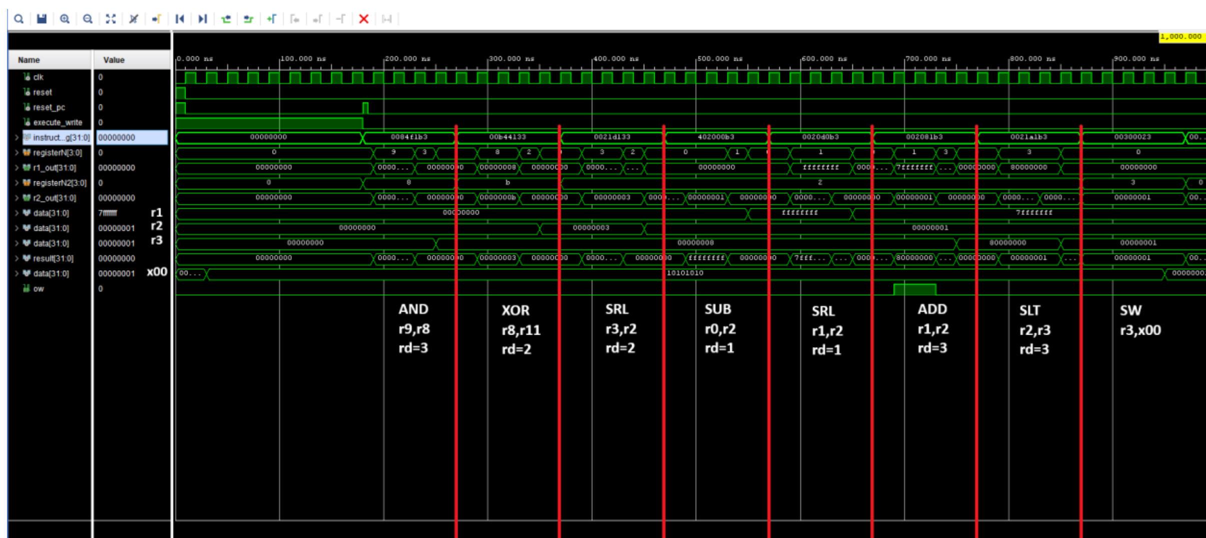

Figura 6.17 : Simulación del programa de comprobación de overflow

En la simulación (Figura 6.17) se puede ver cómo en la ejecución de la instrucción "ADD r1,r2,r3", codificado por *x002081B3* y apreciable en la variable "instruction_reg" en la simulación, el *flag* "ow" (en la parte inferior de la figura) se activa en el momento en el que se hace la suma y se desborda, observable en la variable "result".

7. Aspectos Sociales, Éticos, Legales y Ambientales

El MISC desarrollado no tiene un gran impacto social, al contrario que la arquitectura en la que se basa, que al ser la arquitectura RISC-V una arquitectura *open-source*, ayuda a que proyectos como este de ámbito académico se puedan desarrollar gracias al libre acceso a la información que permiten este tipo de proyectos.

Por esta razón se ha decidido no proteger el proyecto con una licencia restrictiva, y así ayudar con la formación académica, ayudando en el esfuerzo de cumplir el objetivo 4 de los ODS (Objetivos de Desarrollo Sostenible).

En la misma línea del objetivo 4 de los ODS, tenemos que un proyecto como éste, en el que se simplifica lo máximo posible la arquitectura, puede ayudar a hacer la computación más accesible, porque al ser más simple, es más fácil de entender que, por ejemplo, una arquitectura CISC o una RISC más compleja.

Al ser una arquitectura tan sencilla, se puede desarrollar en una FPGA, sin la necesidad usar la industria especializada en fabricación de chip, cuyas plantas precisan de una inversión económica considerable, aportando al objetivo 9.b de los ODS.

Por último, el consumo energético es un tema que ha tomado más relevancia desde el inicio de la guerra de Ucrania, dando más importancia a la procedencia de la energía y cuánto se consume, por lo que se ha considerado importante usar un procesador tipo RISC, que como se ha mencionado anteriormente, consumen menos energía al ser más sencillos, y en muchas ocasiones, ser diseñados para ser usados en sistemas empotrados conectados a una batería.

8. Discusión

Tras la implementación de la ISA, se han podido observar sus limitaciones, las cuales se exponen en este capítulo.

8.1. Banco de registros

Uno de los puntos que más han complicado el diseño del banco de registros y su uso ha sido el acceso simultáneo de sus registros.

Desde un principio se ha pretendido minimizar la cantidad de entradas que tenía el banco de registros con el objetivo de simplificar su implementación, y esto ha sido un grave error porque ha provocado que se haya tenido que implementar el contador en la unidad de control, haciendo así que la ejecución de las instrucciones se divida en 2 partes, tampoco se conseguía de esta forma llegar a ejecutar una instrucción por ciclo.

Añadiendo un conjunto de entradas que permitan manejar el registro de destino se conseguiría tanto eliminar el contador de la unidad de control, como ejecutar al menos una instrucción por ciclo.

8.2. PAI

8.2.1. Inicialización

Durante las pruebas del MISC se ha observado una limitación considerable: a la hora de programar con el juego de instrucciones aportado, no hay forma de insertar un valor a los registros o a memoria si no hay nada almacenado.

Esto quiere decir que, durante una ejecución ya iniciada, o con los registros y/o memoria iniciada con algún valor, se puede almacenar y utilizar cualquier valor, pero si los registros están vacíos y la memoria de datos está vacía, no se puede operar con otro valor que no sea 0.

Este problema se solucionaría con una instrucción que opere con inmediatos como *ADDI* o *LUI*. De esta forma se podría cargar un inmediato, por ejemplo, con "ADDI r0, r1, xFFF", y para cargar el resto del inmediato de 32 bits, se hace un desplazamiento lógico, y se repiten estas dos instrucciones hasta tenerlo completo.

8.2.2. Ramificación

A la hora de diseñar un banco de pruebas para el MISC, ha sido evidente que la elección inicial de instrucción de ramificación no ha sido correcta, ya que no hay ninguna instrucción que permita que antes de llegar a la instrucción de ramificación, no se proceda con el salto, porque, aunque sí que es cierto que SLT compara los valores de los registros aportados, por cómo se describe a *SLT* y a *JAL*, no hay ninguna interacción posible entre las dos instrucciones.

Tras una revisión del conjunto de instrucciones del RISC-V, se ha demostrado que con una instrucción de salto condicional como, por ejemplo, *BEQ* se podría solventar este problema.

Con la instrucción *BEQ* se puede simular la instrucción *JAL* comparando "r0" con "r0" porque "r0" está fijado a 0, por lo que al llegar a esa instrucción se saltará siempre. Esta simulación de *JAL* tiene su limitación, que es que *BEQ* no puede saltar a todas las direcciones a las que te permite saltar *JAL*, ya que el inmediato de *BEQ* es de 12 bits y el de *JAL* es de 20 bits.

8.3. Anotación de diseño

Por último, el contador de programa se podría haber diseñado únicamente con 6 bits, debido a que en el diseño de pruebas construido cuenta con únicamente 64 registros, pero se ha decidido usar el registro básico ya construido. Esto no es tanto un error, si no una anotación de la elección de diseño realizada.

9. Conclusiones

Si consideramos a como referente de los RISC, la arquitectura RISC-V, y el representante de los MISC la arquitectura diseñada. Con estas consideraciones, el conjunto de instrucciones entre el MISC diseñado con su expansión y el juego de instrucciones del RISC-V resultarían en una arquitectura bastante similar.

A falta de comprobarlo de forma práctica, sólo nos podemos basar en la arquitectura para afirmar que, como el consumo de los dos procesadores es similar, por lo que la mayor diferencia entre las dos especificaciones, es que el conjunto de instrucciones de RISC aporta una variedad de instrucciones más amplia.

Una vez contempladas las limitaciones de la arquitectura, y que sus soluciones pasan por incluir más instrucciones, surge la pregunta de sus beneficios suplen sus inconveniencias.

Sin poder comparar el consumo y velocidad de las dos arquitecturas, es imposible dar unas conclusiones definitivas.

Una vez comprobada la eficiencia, el siguiente objetivo sería optimizar lo máximo posible el diseño para intentar aumentar la cantidad de instrucciones ejecutadas por segundo, objetivo que debería ser más sencillo que en otros tipos de procesador debido a que es más sencillo.

Por último, dependiendo de los requisitos de los proyectos donde se pretende implementar el RISC-V, y si se demuestra la eficiencia energética, se podría recomendar usar el MISC si, entre otros parámetros, el programa que se quiere ejecutar en el procesador no requiere de paralelismo.

Referencias

[1] Espressif Systems. *ESP32 Technical Reference Manual*, version 5.4. (2025). Accessed: Jul. 11, 2025. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

[2] Microchip Technologies. *Low-Power, 32-bit Cortex-M0+ MCU with Advanced Analog and PWM*, version 45.19 Rev.A. (2014). Accessed: Jul. 11, 2025. [Online]. Available: https://content.arduino.cc/assets/mkr-microchip_samd21_family_full_datasheet-ds40001882d.pdf

[3] M. J. Flynn, "Historical Development of Computers", *Computer Architecture: Pipelined and Parallel Processor Design*, J. C. Browne and J. S. Werth, Eds., Boston, MA : Jones and Bartlett, 1995, pp. 54.

[4] G. Radin, "The 801 minicomputer", International Business Machines Corporation, New York, New York, United States. Accessed: Jul. 11, 2025. doi: 10.1145/800050.801824 [Online]. Available: https://nanopdf.com/download/the-801-minicomputer_pdf

[5] A. S. Tanenbaum, "Implications of Structured Programming for Machine Architecture", Vrije Universitet Amsterdam, Amsterdam, The Netherlands. Accessed: Jul. 11, 2025. doi: 10.1145/359361.359454 [Online]. Available: https://research.vu.nl/ws/files/110789436/11056

[6] D. A. Patterson and C. H. Sequin, "RISC I: a reduced instruction set VLSI computer", University of California, Berkeley, California, United States. Accessed: Jul. 11, 2025. doi: 10.1145/285930.285981 [Online]. Available: https://www.cse.iitk.ac.in/users/biswap/CS422/RISC-ISCA81.pdf

[7] C. Chen, G. Novick y K. Shimano. "MIPS". Accessed: Jul. 11, 2025. [Online]. Available:
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html

[8] Media.ccc.de, Leipzig. "36C3 - The Ultimate Acorn Archimedes talk" (Dec. 27, 2019). Accessed: Jul. 11, 2025. [Online Video]. Available: https://youtu.be/Hf67JYkUCHQ?t=1410

[9] C. H. Ting and C. H. Moore. “MuP21--A High Performance MISC Processor”. Accessed: Jul. 11, 2025. [Online]. Available: https://www.ultratechnology.com/mup21.html

[10] Ultra Technologies. “F21 CPU”. Accessed: Jul. 11, 2025. [Online]. Available: https://www.ultratechnology.com/f21cpu.html

[11] R.A. Mewaldt , C.M.S. Cohen, W.R. Cook *et al.* “The Low-Energy Telescope (LET) and SEP Central Electronics for the STEREO Mission” in *Space Science Reviews*, vol. 136, Hans Bloemen, Ed., Berlin, Germany: Springer Nature B.V. (2007), pp. 285–362

[12] Eforth. “9. P24 CPU Architecture” P24 Microprocessor User's Manual. Accessed: Jul. 11, 2025. [Online]. Available: https://eforth.com.tw/academy/sutra/chapter9.htm

[13] RISC-V International. “About RISC-V International”. Accesed: Jul. 11, 2025. [Online]. Available: https://riscv.org/about/

[14] A. Waterman, Y. Lee, D. A. Patterson and K. Asanovic. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA.* (2011). Accessed: Jul. 11, 2025. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf

[15] RISC-V Int., Switcherland. *The RISC-V Instruction Set Manual Volume I*, version 20250508. (2025). Accessed: Jul. 11, 2025. [Online]. Available: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

Anexo

Anexo 1: Código

El código del proyecto se ha ubicado en la plataforma gitlab, más en concrete en la instancia de la escuela. Se puede acceder iniciando session con un ususario válido en el siguiente enlace:

https://gitlab.etsisi.upm.es/bs0216/misc_a_partir_de_risc-v

Se han subido únicamente los archivos que contiene el código que implementan los módulos y sus componentes, y las pruebas realizadas en VHDL, evitando subir cualquier archivo de Vivado.