GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

DISEÑO DE UN SOFT CORE RISC-V SEGMENTADO

Autor: Francisco Labora Gómez

Director: José Daniel Muñoz Frías

Co-Director: Fermín Zabalegui Sanz

Madrid

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

Diseño de un soft core RISC-V segmentado

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

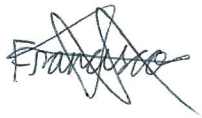
curso académico 2020/21 es de mi autoría, original e inédito y

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: Francisco Labora Gómez Fecha 25/08/2021



Autorizada la entrega del proyecto

LOS DIRECTORES DEL PROYECTO

Fdo.: José Daniel Muñoz Frías Fecha: ……/ ……/ ……

Firmado por MUÑOZ FRIAS JOSE DANIEL - 52573841G el día 25/08/2021 con un certificado emitido por AC FNMT Usuarios

Fdo.: Fermín Zabalegui Sanz Fecha: ……/ ……/ ……

ZABALEGUI SANZ FERMIN - 50202045Z

Firmado digitalmente por ZABALEGUI SANZ FERMIN - 50202045Z
Fecha: 2021.08.25 10:06:32 +02'00'

DISEÑO DE UN SOFT CORE RISC-V SEGMENTADO

Autor: Labora Gómez, Francisco
Director: Zabalegui Sanz, Fermín, José Daniel Muñoz Frías.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Se ha desarrollado una CPU softcore segmentado en 5 etapas. Se ha utilizado la microarquitectura RV32I de RISC-V. Se ha diseñado mediante la descripción a nivel RTL en el lenguaje VHDL. Para comprobar su funcionamiento se han instalado y utilizado Quartus II Lite Edition, ModelSim-Altera, RARS y hex2vhdl.

Palabras clave: RISC-V, FPGA, RV32I, CPU

1. Introducción

Hoy en día es difícil encontrar algún dispositivo electrónico que no cuente con un ordenador integrado. Ya sea una CPU (Central Processing Unit) pequeña integrada en un chip de control o una supercomputadora de alta tecnología, los microprocesadores forman una parte vital de la revolución tecnológica que se ha dado desde la segunda mitad del siglo XXI. Esto hace que su estudio sea de especial interés para aquellas personas que aspiran a diseñar nuevas tecnologías que pongan solución a problemas futuros.

La tendencia del mundo a la informatización causa a su vez que cada vez sean más comunes los planes de estudios que incorporan la electrónica y la informática como materias troncales. Estos currículos académicos de tipo STEM (Science Technology Engineering Mathematics) hacen necesaria la existencia de microprocesadores simples que ejemplifiquen como funciona un sistema de computación de manera didáctica y conceptual.

El proyecto RISC-V nace de este deseo de volver las CPU un elemento accesible para todo el público. El objetivo de este proyecto es la creación de una microarquitectura de uso completamente gratuito y de un nivel de complejidad lo más bajo posible. La iniciativa RISC-V ha conseguido revolucionar la industria y demostrar la relevancia del código abierto (Krste y Patterson, 2014).

Una CPU funciona ejecutando una lista de instrucciones escritas en un lenguaje llamado código máquina, el conjunto concreto de instrucciones que una CPU entiende se denomina ISA (Instruction set Architecture). La cantidad y complejidad de las instrucciones que conforman la ISA determina la complejidad de la CPU.

Las CPU RISC (Reduced Instruction Set Computer) han sido diseñadas para tener una ISA sencilla y reducida para así poder ser implementadas de manera eficiente mediante segmentación.

Además, son open source por lo que su uso es completamente gratuito y son implementables en tarjetas de lógica programable FPGA (*Field Programmable Gate*

Array). Esto permite que el gran público pueda diseñar y personalizar su propio ordenador.

2. Definición del proyecto

En este Proyecto se ha diseñado un microprocesador RISC-V softcore segmentado en 5 etapas. Para realizar el diseño de ha recurrido al lenguaje de descripción de hardware VHDL implementado mediante el software Quartus II Lite Edition de Altera. Para cargar instrucciones a la CPU se ha recurrido a la capacidad de Quartus de inicializar memorias a valores concretos junto a las herramientas RARS y hex2vhdl.

RARS es una herramienta open source de java que emula una CPU de RISC-V. Permite escribir instrucciones en ensamblador y simular su funcionamiento en la CPU. También permite para generar un archivo de texto plano con el código máquina codificado en hexadecimal. El programa de Python hex2vhdl permite tomar las instrucciones en hexadecimal y generar un archivo de VHDL que describe una memoria ROM con las instrucciones cargadas.

Utilizando Quartus para escribir las instrucciones, hex2vhdl y RARS para compilarlas y describirlas en VHDL y ModelSim-Altera para simular a nivel RTL la CPU se puede comprobar si el diseño funciona correctamente o no.

3. Descripción del microprocesador.

Para diseñar el microprocesador se ha diseñado primero el nivel RTL atendiendo a las necesidades de hardware de cada instrucción y teniendo en cuenta los efectos de la segmentación y la necesidad de sincronismo de los datos.

Este nivel RTL se ha descrito en VHDL generando por separado cada componente de la CPU. Los diferentes componentes se han incorporado a la etapa correspondiente y por último las etapas se han combinado con registros intermedios para formar el top level del microprocesador. La descomposición en niveles se muestra en la Ilustración 1.

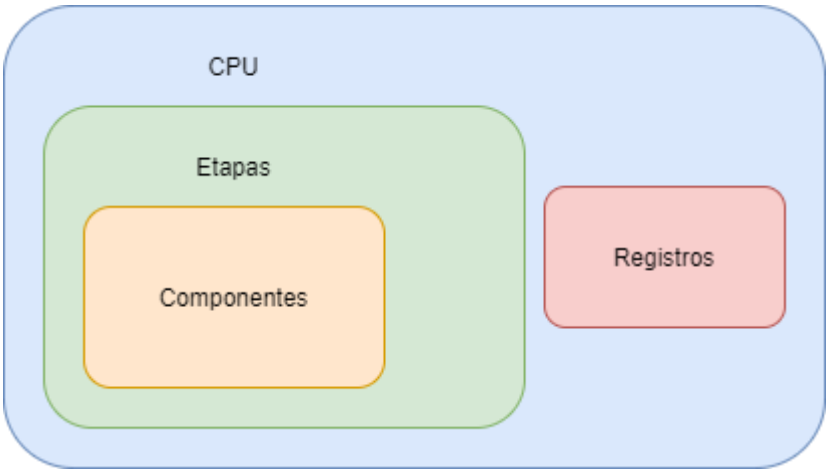


Ilustración 1. Niveles de diseño de la CPU

El microprocesador ha sido segmentado siguiendo el criterio habitual en RISC-V dando lugar a las etapas Instruction Fetching, Instruction Decode, Execution, Memory y Write Back. El diagrama RTL estructural del pipeline se muestra en la Ilustración 2.

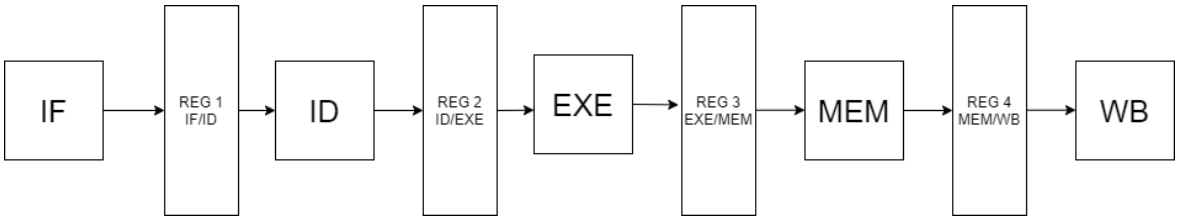


Ilustración 2. Diagrama RTL del pipeline.

4. Resultados

Se ha conseguido diseñar un microprocesador RV32I capaz de ejecutar las instrucciones de la ISA base de RISC-V. Calcula correctamente todas las operaciones, es capaz de controlar el flujo de programa, lee correctamente los datos de los registros y es capaz de almacenar y cargar datos desde una memoria RAM.

También se ha utilizado un sistema para codificar instrucciones de ensamblador en una memoria ROM descrita en VHDL que se puede implementar en el diseño para cargar las instrucciones al microprocesador. Este proceso se ha llevado a cabo gracias a las herramientas open source RARS y hex2vhdl.

Para comprobar el funcionamiento se ha utilizado la simulación RTL de Quartus. Esta simulación se realiza por medio de la herramienta ModelSim-Altera con un fichero de testbench programado para el diseño. La simulación se compara con la simulación que proporciona RARS

El resultado de una simulación exitosa para instrucciones de carga desde memoria se muestra en la Ilustración 3. El resultado de la simulación de RARS se muestra en la Ilustración 4.

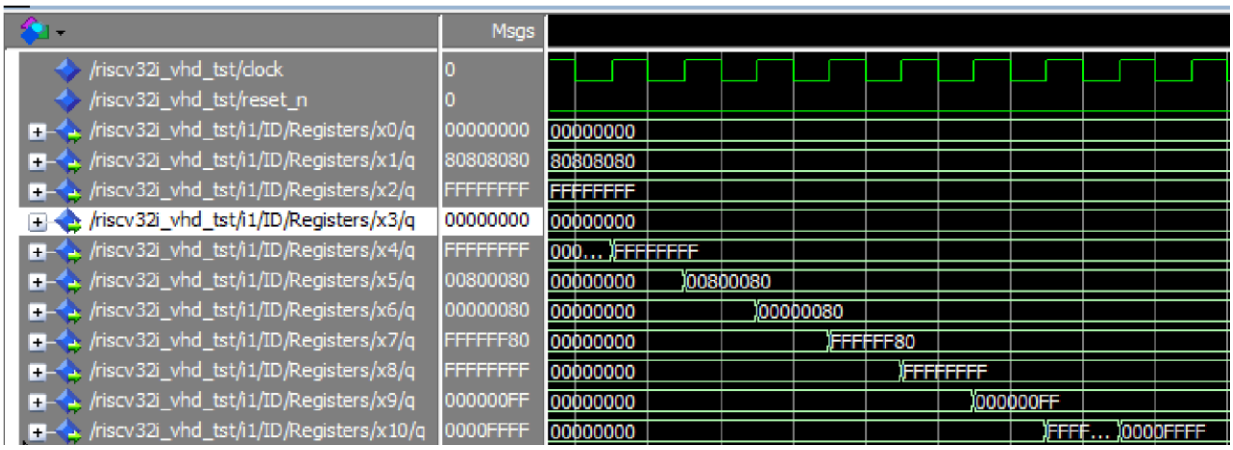


Ilustración 3. Simulación de instrucciones de carga de memoria

Number	Value
0	0x00000000
1	0x80808080
2	0xffffffff
3	0x00000000
4	0xffffffff
5	0x00800080
6	0x00000080
7	0xffffff80
8	0xffffffff
9	0x000000ff
10	0x0000ffff

Ilustración 4. Simulación de RARS

```
addi x1,x0,128
addi x2,x0,-1
sb x1,(x0)
sb x1,1(x0)
sb x1,2(x0)
sb x1,3(x0)
sb x2,4(x0)
sb x2,5(x0)
sb x2,6(x0)
sb x2,7(x0)
sh x1,8(x0)
sh x1,10(x0)
sw x1,12(x0)
lw x1,0
lw x4,4
lw x5,8
lw x6,12
lb x7,0
lb x8,4
lbu x9,4
lh x10,4
lhu x10,4
```

Ilustración 5. Código ensamblador de la simulación

Para comprobar que la simulación ha sido exitosa basta con comparar los valores de los registros en la simulación de RARS y la simulación de Quartus.

5. Conclusiones

La CPU diseñada presenta mucho potencial dado el alto grado de personalización que ofrece RISC-V junto a la sencillez de su diseño.

El microprocesador puede ser utilizado con fines didácticos y académicos debido al bajo coste que representa su adquisición, la facilidad para la modificación de la CPU y la cantidad de herramientas de soporte de terceros que existen.

Además, el RV32I puede modificarse añadiendo extensiones a la ISA y soporte para diversos periféricos creando una CPU a medida para el control de un proceso industrial determinado. No solo es posible modificar la lógica de control a nivel software, sino que también es posible configurar el hardware de tal forma que la ejecución sea lo más rápida y eficiente posible.

6. Referencias

[1] Krste, A.; Patterson A. (2014) *Instruction Sets Should Be Free: The Case For RISC-V.*

DESIGN OF A SOFT CORE RISC-V CPU.

Author: Labora Gómez Francisco.
Supervisor: Zabalegui Sanz, Fermín, José Daniel Muñoz Frías
Collaborating Entity: ICAI – Universidad Pontificia Comillas)

ABSTRACT

The RV32I ISA from RISC-V has been used to design a softcore CPU with 5 stage pipeline segmentation. The design has been made at RTL level in VHDL. The third-party programs Quartus II Lite Edition, ModelSim-Altera, RARS and hex2vhdl have been installed and used for verification purposes.

Keywords: RISC-V, FPGA, RV32I, CPU

1. Introduction

It is practically impossible nowadays to find an electronic device which does not have a CPU in it. The use of CPUs is a core aspect of the technological revolution of the 21st century. We can find CPUs ranging from small integrated control chips to latest technologies supercomputers. The extension in the use of microprocessors makes them a particularly interesting topic for those who aspire to design technologies that can solve the issues humanity may face in the future.

The world is tending towards computerization and therefore the amount of relevance given to STEM centered education is rising every day. This focus on IT based plans

of study generates the necessity of cheap accessible CPUs which can exemplify how a computer works in a didactic manner.

Due to this urge of turning CPUs into a widely available resource the RISC-V project was created to provide an open-source ISA with a low level of complexity. This not only made CPUs way cheaper and more accessible for those willing to learn. Industry was also “revolutionized by open standards and open-source software” (Krste & Patterson, 2014).

A CPU works by executing a series of instructions written in machine code, which is a list of 0s and 1s, the complexity and amount of instructions understood by a CPU determines the complexity of said CPU. Therefore RISC (Reduced Instruction Set Computer) CPUs have been designed with the objective of lowering as much as possible the complexity of the instruction set.

They are also compatible with FPGA technology which makes them widely available and grants a high degree of customization and design freedom.

2. Project Definition

The aim of this project is to design a RISC-V softcore CPU with 5 stage pipeline segmentation. For designing purposes, the hardware description language VHDL has been used in conjunction with Altera’s Quartus II Lite Edition. In order to program the ROM instruction memory, Quartus’s capability of initializing registers at a certain value has been combined with the software tools RARS and hex2vhdl.

RARS is an open-source java program for RISC-V, it is able to compile assembly instructions into hexadecimal machine language. This machine language is dumped into a plain text file so that it can be read by hex2vhdl, a python program which generates the RTL description of a ROM with the hexadecimal instructions written in it.

Using Quartus to physically write the instructions, hex2vhdl and RARS to compile and describe them and ModelSim-Altera to simulate their outcome it is possible to determine if the CPU works correctly or not.

3. Description of the CPU.

The microprocessor has been designed at a register transfer level to fulfill the hardware demands of the RV32I ISA and taking into account the effects of segmentation.

This RTL level has been described in VHDL generating each module separately and combining them at the stage corresponding to the module. After, the 5 stages have been combined with the intermediate registers and between themselves to form the top level RV32I. The hierarchy of the design is shown at Figure 1.

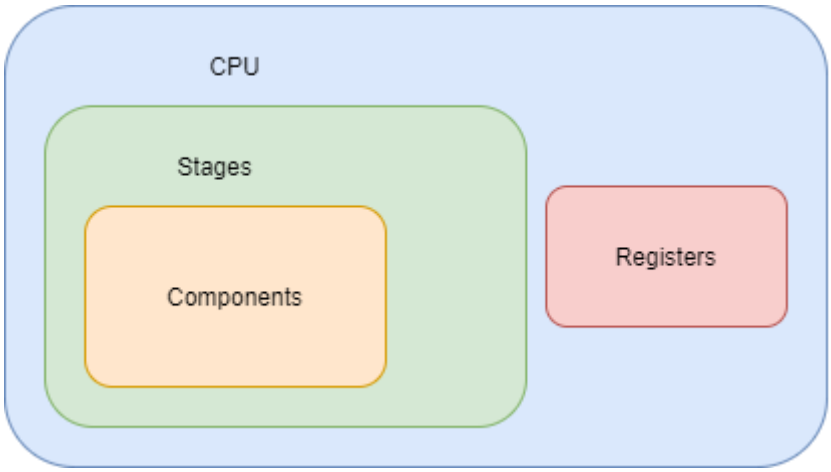


Figure 1. Hierarchy of the design

The CPU has been segmented following the RISC-V usual criteria, the stages are Instruction Fetch, Instruction Decode, Execution, Memory and Write Back. The RTL diagram of the CPU is shown at Figure 2.

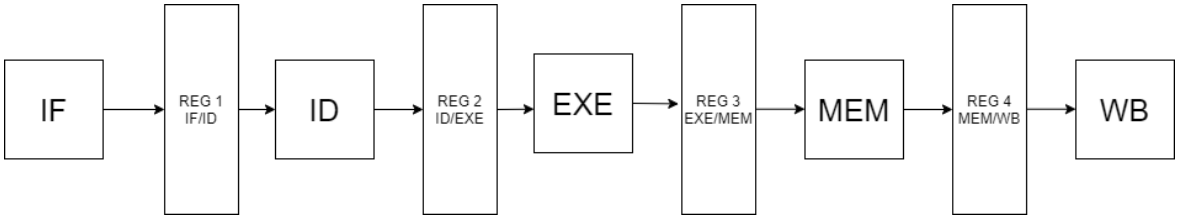


Figure 2. RTL Diagram of the CPU

4. Results

A RV32I CPU capable of executing the base RISC-V instructions has been successfully designed. It correctly calculates the output of all instructions, can control the flow of the program, reads and writes correctly register data and can load and store in a RAM memory.

A set of tools used for compiling and loading instructions into the program memory has also being installed. This is possible due to the open-source software development tools RARS and hex2vhdl.

The RTL simulation provided by Quartus using ModelSim-Altera simulation tool configured by a testbench file has been used to verify the results of the design.

The successful execution of load instructions is shown at Figure 3. The RARS simulation is shown at Figure 4 for comparison, the assembly code used is shown at Figure 5

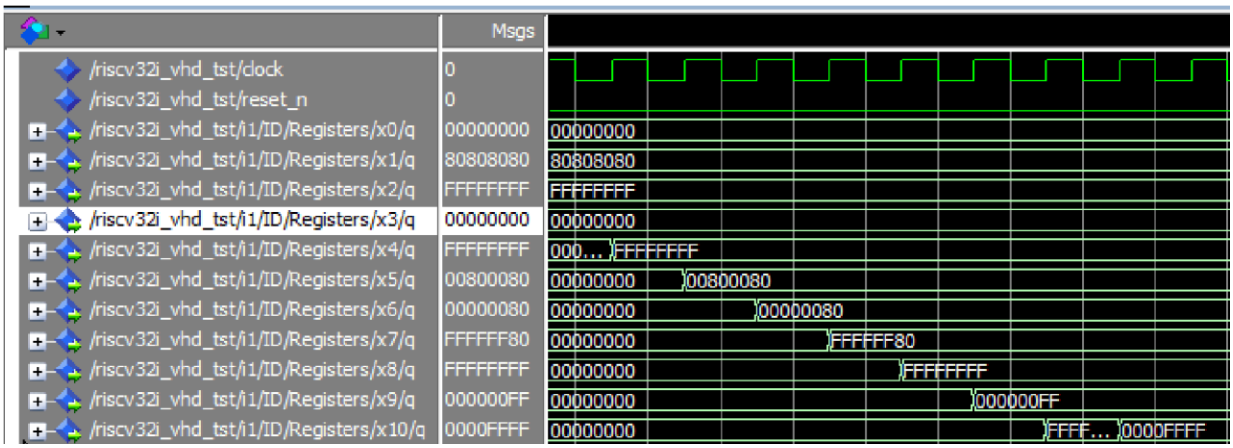


Figure 3. Quartus simulation

Number	Value
0	0x00000000
1	0x80808080
2	0xffffffff
3	0x00000000
4	0xffffffff
5	0x00800080
6	0x00000080
7	0xffffff80
8	0xffffffff
9	0x000000ff
10	0x0000ffff

Figure 4. RARS simulation

```
addi x1,x0,128
addi x2,x0,-1
sb x1,(x0)
sb x1,1(x0)
sb x1,2(x0)
sb x1,3(x0)
sb x2,4(x0)
sb x2,5(x0)
sb x2,6(x0)
sb x2,7(x0)
sh x1,8(x0)
sh x1,10(x0)
sw x1,12(x0)
lw x1,0
lw x4,4
lw x5,8
lw x6,12
lb x7,0
lb x8,4
lbu x9,4
lh x10,4
lhu x10,4
```

Figure 5. Assembly code

Para comprobar que la simulación ha sido exitosa basta con comparar los valores de los registros en la simulación de RARS y la simulación de Quartus.

5. Conclusions

The CPU which has been designed has a high potential due to the amount of personalization RISC-V provides in addition to its simplicity.

This CPU can be used as a learning tool which can show students computers work and how complex systems are built. This is possible due to the low cost of FPGAs and the amount of third-party tools available for use.

Also, the RV32I can be used as a industrial control CPU, due to its high grade of customization possibilities, a CPU with certain extensions and input and output support can be tailored for a certain application so that efficiency and speed can be maximized.

6. References

[1] Krste, A.; Patterson A. (2014) *Instruction Sets Should Be Free: The Case For RISC-V*.

Índice de la memoria

Índice de figuras

Índice de tablas

Capítulo 1. INTRODUCCIÓN

Este proyecto tiene su origen en la rápida expansión del uso de la tecnología FPGA desde su nacimiento hace cinco décadas y en la creciente importancia que están tomando las competencias STEM y de electrónica digital en los sistemas educativos.

La combinación de una tecnología joven con muchas posibilidades por explotar con un área de disciplinas que puede valerse de esta tecnología para expandirse genera un caldo de cultivo ideal para la investigación en implementaciones *softcore* de microprocesadores.

La capacidad de desarrollar nuevas tecnologías que afronten los retos a los que se enfrentará la humanidad durante los próximos años depende de manera directa de la calidad de la educación que se pueda aportar a las generaciones venideras. Y la calidad de esta educación depende a su vez de la calidad de los recursos que se pongan en manos de las nuevas generaciones.

Uno de los recursos de mayor necesidad para el mundo de la tecnología es sin duda el microprocesador, sin él la mayoría de las aplicaciones de electrónica no serían posibles tal y como las conocemos. Es por tanto que el acceso a un recurso tan preciado debe ser facilitado por todos los medios posibles

Con este mismo objetivo en mente comienza en 2010 el programa RISC-V para desarrollo de microprocesadores de propósito general altamente accesibles. Hoy en día cuenta con una comunidad internacional consolidada. El carácter abierto de RISC-V facilita que personas de todo el mundo y de distintas instituciones aporten su pequeño grano de arena al futuro de la tecnología. Por esto mismo cantidad de proyectos tanto de hardware como de software basados en RISC-V y completamente abiertos y gratuitos crece cada día.

Para un ingeniero industrial especializado en la rama de electrónica, la capacidad hacer un pequeño aporte a la comunidad de desarrolladores de RISC-V resulta un proyecto altamente atractivo y enriquecedor.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

2.1 ISA RISC-V

En primer lugar, el término ISA (Instruction Set Architecture) se refiere a un conjunto de instrucciones escritas en código máquina (secuencia binaria de unos y ceros) que un microprocesador es capaz de decodificar y ejecutar.

La arquitectura o microarquitectura constituye uno de los diferentes niveles de diseño o abstracción de un sistema electrónico. Estos niveles indican el grado de complejidad física del sistema y se ordenan de manera jerárquica por orden de complejidad, un nivel más alto implica un mayor grado de abstracción.

Cada nivel trata a los demás como un sistema de cajas negras del cual se conocen las entradas y salidas, pero se desconoce la lógica interna de funcionamiento. Los niveles de abstracción van desde el nivel físico, que describe el funcionamiento de los componentes electrónicos como sistemas eléctricos y atómicos, hasta el nivel de sistema operativo que utiliza todos los demás niveles para ejecutar instrucciones complejas. La jerarquía se muestra en la Figura 1.

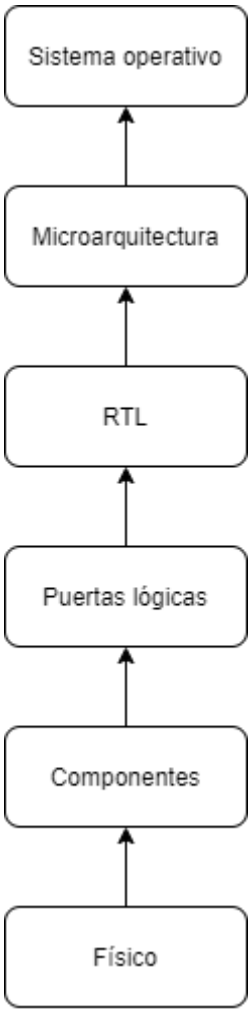


Figura 1. Niveles de abstracción de un sistema electrónico

Para el diseño del microprocesador se ha trabajado en el nivel de transferencia de registros o RTL (Register Transfer Level) que consiste en una combinación de lógicas combinacionales y secuenciales mediante las cuales se puede establecer un proceso lógico que genera señales electrónicas de salida en función de una serie de señales de entrada y el estado del sistema.

El sistema en nivel RTL toma un conjunto de instrucciones definidas en el nivel de microarquitectura, la ISA, e implementa un hardware capaz de ejecutarlas.

En este caso se ha trabajado con la ISA RISC-V perteneciente a las arquitecturas RISC (Reduced Instruction Set Computer), existen otras muchas arquitecturas como la x86 utilizada por Intel o la MIPS de MIPS technologies. Lo que diferencia RISC-V del resto de arquitecturas es su carácter *open source*, esto significa que es una ISA de libre acceso al público que no requiere del pago de *royalties* para su uso, lo que la convierte en una ISA idónea para ámbitos académicos.

RISC-V nace en 2010 de un proyecto de la universidad de California, Berkeley, con el objetivo de crear una ISA open source de uso generalizado con un alto grado de adaptabilidad y personalización (Krste y Patterson, 2014). La ISA busca ser sencilla, con un número de instrucciones reducido, pero fácilmente ampliable con la instalación de módulos adicionales que se añaden a la arquitectura base dando soporte a nuevas funcionalidades. Este carácter modular sumado a la naturaleza open source de RISC-V tiene como objetivo que cualquier persona pueda aportar su trabajo al proyecto para que otros usuarios se beneficien de el sin coste alguno (Krste y Patterson, 2014).

Todos estos factores han convertido a RISC-V en una arquitectura de uso generalizado en ámbitos no sólo académicos sino también industriales y de aplicaciones de baja potencia, ya que la simplicidad de RISC-V permite la ejecución eficiente de instrucciones sencillas.

El éxito del proyecto ha llevado a la creación de la RISC-V International, una organización cuyo objetivo es promover el uso de la ISA a nivel internacional y expandir su ecosistema de *hardware* y *software*.

2.2 TIPOS DE RISC-V

Como ya se ha indicado antes, RISC-V tiene una fuerte componente modular, existiendo una serie de arquitecturas base a las cuales se pueden añadir extensiones que generan soporte para diversas funcionalidades.

Toda arquitectura RISC se define en base a una nomenclatura que especifica las operaciones y tipos de variables para los cuales ofrece soporte dicha arquitectura, para explicar la nomenclatura de las ISA de RISC-V se pondrá como ejemplo la arquitectura base utilizada en el microprocesador diseñado en este proyecto, el RV32I:

- **RV** – Abreviación de RISC-V, todas las arquitecturas RISC-V comienzan su nombre así.
- **32** – Indica el ancho de bits de la arquitectura, define la longitud de las palabras en la arquitectura y el ancho de los registros
- **I** – Indica soporte para números enteros, forma parte de la arquitectura base.

Para indicar que aparte de la arquitectura base, la ISA utilizada tiene extensiones adicionales, se añaden letras al final del nombre que indican para que funcionalidad concreta tiene soporte la ISA.

Las letras y las funcionalidades están definidas como parte del proyecto RISC-V, aquellas extensiones estables cuyo desarrollo ya ha concluido se muestran en la *Tabla 1*.

Código	**Soporte de la extensión**
M	Multiplicación y división
A	Instrucciones atómicas
F	Coma flotante precisión simple (32 bits)
D	Coma flotante precisión doble (64 bits)
Zicsr	CSR (Control and Status Register)

Zifencei	Fence.I (Escritura en memoria de programa)
Q	Coma flotante precisión cuádruple (128 bits)
C	Instrucciones comprimidas

Tabla 1. Extensiones de RISC-V

Por tanto, un microprocesador de 64 bits con capacidad para trabajar con enteros y coma flotante de precisión simple y capaz de realizar aritmética y lógica básicas junto a multiplicaciones se denominaría RV64IMF.

Existe un caso particular en el cual si un microprocesador es de tipo IMAFD (enteros, multiplicación, coma flotante con precisión simple y doble) se sustituye este conjunto de letras por una G (General purpose). Esto se debe a que es una configuración muy habitual y una de las principales políticas de diseño de RISC-V es volver los casos habituales lo más simples posible.

2.2.1 RV32I. INSTRUCCIONES BASE

En la arquitectura RV32I en la que se basa este documento existen 37 instrucciones base. Estas instrucciones son las necesarias para crear un microprocesador capaz de ejecutar código de lenguajes de alto nivel compilado (Winans, 2021). Además de estas 37 instrucciones existen otras instrucciones base que no son estrictamente necesarias para el funcionamiento de un RV32I y que no son el objetivo de este documento.

Las instrucciones se codifican en una secuencia de 32 bits en la cual se encuentran todos los datos necesarios para la ejecución de la instrucción. Existen 4 grupos de instrucciones principales (R, I, S y U) y 2 subgrupos (B y J), cada grupo se codifica de una manera

determinada y única. Los dos subgrupos B y J son instrucciones de tipo S y U respectivamente con ligeras modificaciones sobre la codificación del inmediato.

Los grupos y su codificación en código máquina de 32 bits se muestran en la *Tabla 2*.

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

Tabla 2. Tipos de instrucciones

Los grupos y subgrupos de instrucciones son:

- **R:** Operaciones entre registros.
- **I:** Operaciones con inmediatos.
- **S:** Operaciones de almacenamiento en memoria (*Stores*).
- **U:** Saltos incondicionales.
- **B:** Saltos condicionales o *branches.*
- **J:** Salto relativo a *Program Counter* (instrucción JAL)

Además de por cómo se codifican las instrucciones, se pueden distinguir en base al soporte que ofrece cada grupo de instrucciones. Relativo a la funcionalidad nos encontramos instrucciones aritmeticológicas, de control y de memoria.

2.2.1.1 Instrucciones aritmeticológicas

Las instrucciones aritmeticológicas son aquellas encargadas de realizar operaciones aritméticas simples, lógicas bit a bit, desplazamientos y comparaciones. Pueden ser operaciones con inmediatos o entre registros, diferenciándose en que el segundo dato se encuentre almacenado en un registro o codificado en la instrucción en forma de inmediato. Aquellas operaciones que son de tipo *unsigned* toman los valores de los datos como números de 32 bits sin signo en vez de como datos en complemento a 2.

Las operaciones de tipo desplazamiento se codifican de una manera especial, teniendo en cuenta únicamente los 5 bits menos significativos del segundo operando. Esto se debe a que con 32 bits no tiene sentido desplazar más de 31 posiciones y 5 bits codifican 32 números. El número de bits a desplazar se denomina shamt (Shift Amount). Cuando la operación de desplazamiento es hacia la derecha se distinguen dos tipos de desplazamiento: desplazamiento lógico o sin signo y desplazamiento aritmético o con signo.

En la *Tabla 3* se muestran todas las instrucciones de este tipo junto a su codificación en ensamblador y la funcionalidad de la instrucción.

Instrucción	Nombre	Tipo	Ensamblador	Funcionalidad
addi	Add immediate	I	addi rd,rs1,imm	rd = rs1 + imm
slti	Set les than immediate	I	slti rd,rs1,imm	rd=(1|0) si rs1<imm

sltiu	Set less than immediate unsigned	I	sltiu rd,rs1,imm	rd=(1|0) si rs1<imm
xori	Exclusive OR immediate	I	xori rd,rs1,imm	rd = rs1 xor imm
ori	OR imediate	I	ori rd,rs1,imm	rd = rs1 or imm
andi	AND immediate	I	andi rd,rs1,imm	rd = rs1 and imm
slli	Shift left logical immediate	I	slli rd,rs1,shamt	rd = rs1 << shamt
srli	Shift right logical immedaite	I	srli rd,rs1,shamt	rd = rs1 >> shamt
srai	Shift right aritthmetic immediate	I	srai rd,rs1,shamt	rd = rs1 >> shamt
add	Add	R	add rd,rs1,rs2	rd = rs1 + rs2
sub	Subtract	R	sub rd,rs1,rs2	rd = rs1 - rs2
slt	Set les than	R	slt rd,rs1,rs2	rd = (1|0) rs1<rs2
sltu	Set les than unsigned	R	sltu rd,rs1,rs2	rd = (1|0) rs1<rs2

sll	Shift left llogical	R	sll rd,rs1,rs2	rd = rs1 << rs2
srl	Shift right logical	R	srl rd,rs1,rs2	rd = rs1 >> rs2
sra	Shift right arithmetic	R	sra rd,rs1,rs2	rd = rs1 >> rs2
xor	Exclusive OR	R	xor rd,rs1,rs2	rd = rs1 xor rs2
or	OR	R	or rd,rs1,rs2	rd = rs1 or rs2
and	AND	R	and rd,rs1,rs2	rd = rs1 and rs2
lui	Load upper immediate	I	Lui rd, imm	Rd =imm*2^12

Tabla 3. Instrucciones aritmeticológicas

2.2.1.2 ***Instrucciones de control***

Las instrucciones de control son las que distinguen a un microprocesador de un simple dispositivo de cálculo, permiten realizar saltos en el contador de programa y por tanto ejecutar o no conjuntos de instrucciones dependiendo de condiciones lógicas. Además de estos saltos condicionales, existen dos tipos de salto incondicional, uno para saltos relativos a registro y otro para saltos relativos al contador de programa. Estos saltos ofrecen soporte para la ejecución de subrutinas localizadas en una parte concreta del código.

De manera análoga a con las instrucciones aritmeticológicas se muestra en la *Tabla 4* el conjunto de instrucciones de control.

Instrucción	Nombre	Tipo	Ensamblador	Funcionalidad
Jal	Jump and link	J	jal rd,imm	rd=pc+4 pc=pc+imm
Jalr	Jump and link register	I	jalr rd,imm(rs1)	rd=pc+4 pc=rs1+imm
Beq	Branch if equal	B	rs1,rs2,imm	pc =(pc+imm|pc) (rs1=rs2|rs1≠rs2)
Bne	Branch if not equal	B	rs1,rs2,imm	pc= (pc+imm|pc) (rs1≠rs2|rs1=rs2)
Blt	Branch if les than	B	rs1,rs2,imm	pc= (pc+imm|pc) (rs1<rs2|rs1≥rs2)
Bge	Branch if greater than	B	rs1,rs2,imm	pc= (pc+imm|pc) (rs1>rs2|rs1≤rs2)
Bltu	Branch if les than unsigned	B	rs1,rs2,imm	pc= (pc+imm|pc) (rs1<rs2|rs1≥rs2)
Bgeu	Branch if greater than unsigned	B	rs1,rs2,imm	pc= (pc+imm|pc) (rs1≥rs2|rs1<rs2)

Tabla 4. Instrucciones de control

2.2.1.3 ***Instrucciones de memoria***

Las instrucciones de memoria permiten trabajar con una memoria RAM (Random Access Memory) donde almacenar datos para ser utilizados posteriormente. La arquitectura RISC-V es de tipo *Load/Store*, esto significa que los accesos a memoria son únicamente para cargar o almacenar datos, nunca se opera con datos desde la memoria, primero se cargan en registro se operan y se vuelven a almacenar. El direccionamiento a memoria es por bytes, pudiéndose cargar y almacenar bytes (8 bits), medias palabras (16 bits) y palabras (32 bits). Al tener disponibles 32 bits para direccionamiento RISC-V ofrece soporte para memorias de hasta 4 GiB (2^{32}=4.295.967.296).

En el caso del diseño para este proyecto se ha utilizado una memoria de 256 palabras ya que para los objetivos académicos y conceptuales que se han perseguido no es necesaria una memoria grande que simplemente volvería el diseño más lento y costoso a nivel de *hardware.* Además, un diseño con más memoria no podría implementarse en una FPGA ya que estas están limitadas a unos pocos KB de memoria.

RISC-V es una arquitectura de tipo *little endian* lo que significa que los bits menos significativos se almacenan en las posiciones de memoria más bajas. En la Tabla 5 se muestra la diferencia entre un formato *little endian* y *big endian* en 32 bits con el dato hexadecimal 0x01234567.

Little endian				
Dirección	3	2	1	0
Valor	01	23	45	67

Big endian				
Dirección	3	2	1	0
Valor	67	45	23	01

Tabla 5. Formatos little endian y big endian

Aunque RISC-V teóricamente ofrezca soporte para escrituras en memoria sin alinear (esto quiere decir que es posible escribir por ejemplo en las direcciones 1 y 2 una media palabra) el microprocesador diseñado sólo permite escrituras alineadas. Esto significa que las medias palabras solo pueden ser escritas en direcciones pares y las palabras solo pueden ser escritas en direcciones múltiplo de 4.

La escritura en direcciones no alineadas es una práctica poco común hasta el punto de que unidades de memoria para FPGA estándar no tienen soporte para esta operación a nivel de hardware. El diseño de la memoria y aspectos sobre la alineación se tratarán en más profundidad en el apartado de la etapa de memoria del microprocesador. En caso de haber un acceso no alineado se deberá resolver vía software mediante una excepción, lo que haría el acceso tremendamente lento.

Las instrucciones de RISC-V que dan soporte al acceso a memoria junto a su funcionalidad se muestran en la Tabla 6.

Instrucción	**Nombre**	**Tipo**	**Ensamblador**	**Funcionalidad**

Lb	Load byte	I	lb rd,imm(rs1)	rd= 8LSB address rs1+imm
Lh	Load half-word	I	lh rd,imm(rs1)	rd= 16LSB address rs1+imm
Lw	Looad word	I	lw rd,imm(rs1)	rd= 32LSB address rs1+imm
Lbu	Load byte unsigned	I	lbu rd,imm(rs1)	rd= 8LSB address rs1+imm
Lhu	Load half-word unsigned	I	lhu rd,imm(rs1)	rd= 16LSB address rs1+imm
Sb	Store byte	S	sb rs2,imm(rs1)	Store rs2 8LSB address rs1+imm
Sh	Store half-word	S	sh rs2,imm(rs1)	Store rs2 16LSB address rs1+imm

Sw	Store word	S	sw rs2,imm(rs1)	Store rs2 32LSB address rs1+imm

Tabla 6. Instrucciones de memoria

2.3COMPONENTES DE UN PROCESADOR RISC-V

Un microprocesador con arquitectura RISC-V tiene una serie de componentes necesarios para su funcionamiento. Estos componentes se dividen en dos grupos dependiendo de su función:

- **Data Path:** los componentes del data path o ruta de datos son los encargados de realizar los cálculos, seleccionar datos de entrada en componentes, almacenar valores, etc. Son componentes con funciones concretas que requieren de entradas de control para realizar su tarea. Estas señales de control determinan que operaciones hacer, que datos seleccionar como salida, si se debe escribir en un registro, etc.
- **Circuito de control:** El circuito de control está compuesto de los diferentes componentes de control del microprocesador, estos componentes son los encargados de generar las señales de control que el data path utiliza para funcionar. El principal componente es la unidad de control del Instruction Decode que se detallará más adelante.

Todos los componentes utilizados se explicarán más adelante en las secciones correspondientes a cada componente. En este punto sólo se pretende realizar una

explicación breve sobre el funcionamiento de ciertos componentes indispensables para entender la lógica de la ISA RISC-V. Dichos componentes básicos son:

- **Registros:** El banco de registros es un conjunto de 32 registros numerados desde x0 hasta x31. Los registros son el lugar del que el microprocesador obtiene datos para la mayoría de las operaciones. El número de registros en una arquitectura condiciona la velocidad y coste del acceso a los mismos (a mayor número de registro, lógica de direccionamiento más lenta y costosa) además de ocupar más bits para definir las direcciones en las instrucciones. Tener demasiados pocos registros implica necesitar constantemente accesos a memoria por falta de espacio.

  32 registros es una cantidad habitual en muchas arquitecturas siendo un buen compromiso entre consumo de recursos y espacio para guardar variables relevantes.

  Los registros a su vez están asignados para almacenar de manera estándar ciertos tipos de valores, por ejemplo, los registros x5, x6 y x7 son de propósito general temporales (no conservan valor entre funciones y dan soporte a variables locales) mientras que otros registros como el x10 y x11 son los registros de retorno de valores de función. En función de estos usos estandarizados cada registro recibe un nombre concreto que hace alusión a su propósito. El propósito de cada registro y su nombre se indica en la Tabla 7.

Registros	Nombre	Abreviatura	Propósito
x0	Zero	zero	Constante 0
x1	Return address	ra	Dirección de retorno de función

x2	Stack pointer	sp	Siguiente posición de memoria disponible en el stack
x3	Global pointer	gp	Puntero a la zona de variables globales
x4	Thread pointer	tp	Se usa para dar soporte a programas multihilo.
x5-7	Temporaries	t0-2	Propósito general volátil (no se conservan entre funciones)
x8	Saved register/Frame pointer	s0/fp	Propósito general global (se conserva entre funciones) / Puntero al área de activación (donde las funciones almacenan variables locales dentro de la pila)
x9	Saved Register	s1	Propósito general global
x10-11	Function arguments / Return values	a0-1	Paso de argumentos y retorno de valores
x12-17	Function arguments	a2-7	Paso de argumentos
x18-27	Saved registers	s2-s7	Propósito general global
x28-31	Temporaries	t3-t6	Propósito general volátil

Tabla 7. Nomenclatura y propósito de registros

El único registro con valor fijo es el x0 que está conectado a tierra y vale siempre 0, este registro tiene este valor ya que el 0 es el número más utilizado y es conveniente tener un registro de trabajo que valga constantemente 0.

- **Contador de programa:** Es un registro que almacena qué instrucción se va a leer a continuación, su valor es la dirección en la memoria de programa de la instrucción a ejecutar.
- **Memoria de instrucción:** Componente de memoria que almacena las instrucciones de código máquina en 32 bits. Este componente es una memoria ROM y la principal entrada del sistema y se programa cada vez que se quiere cargar *software*.
- **Memoria de datos:** Similar a la memoria de instrucción, pero encargada de almacenar aquellos datos que se escriben en memoria. Es una memoria RAM que permite almacenar una cantidad elevada de datos que no cabrían en los registros.
- **ALU:** La ALU (Arithmetic Logic Unit) es la calculadora del microprocesador y núcleo del *datapath*, se encarga de realizar todos los cálculos requeridos en las instrucciones.
- **Unidad de control:** Decodifica las instrucciones y genera las señales de control para el resto de los componentes.
- **Unidad de inmediatos:** Decodifica el inmediato de una instrucción en caso de haberlo.
- Además de estos componentes existen otros pequeños módulos sencillos de lógica combinacional que, aunque son necesarios para el funcionamiento del microprocesador no son necesarios para comprender su funcionamiento y serán explicados más adelante en las secciones correspondientes a cada caso concreto.

2.4 MICROPROCESADOR MULTICICLO. PIPELINE SEGMENTADO.

Combinando los elementos mencionados anteriormente es posible diseñar un microprocesador RV32I capaz de ejecutar código de manera completamente funcional, no obstante, existe una técnica para aumentar la velocidad del microprocesador sin aumentar prácticamente el coste en *hardware*, esta técnica es la segmentación del procesador.

Un microprocesador es un componente de electrónica digital que realiza una serie de tareas de manera secuencial para ejecutar una instrucción. Si se tiene en mente este concepto de microprocesador se puede descomponer una instrucción en una serie de etapas que pueden ser ejecutadas de manera simultánea e independiente en diferentes partes del microprocesador. RISC-V está especialmente diseñado para una segmentación en 5 etapas siendo las etapas las siguientes:

1. Instruction Fetch (IF): Se obtiene la instrucción a ejecutar de la memoria de programa.
2. Instruction Decode (ID): Se decodifica la instrucción, se genera el inmediato y las señales de control del datapath.
3. Execution (EXE): Se realizan los cálculos necesarios para la instrucción.
4. Memory (MEM): Se realizan accesos de lectura y escritura a memoria en caso de ser necesarios.
5. Write Back (WB): Se escribe en registro en caso de que la instrucción lo requiera.

El objetivo de segmentar las instrucciones es ser capaz de ejecutar secuencialmente cada etapa de una instrucción mientras la etapa anterior de la instrucción siguiente se ejecuta, permitiendo así que todo el hardware de microprocesador esté activo al mismo tiempo. Al resultado de esta segmentación se le denomina pipeline debido a su similitud con una tubería donde un fluido avanza por el interior. Las diferentes etapas del pipeline se encuentran unidas por 4 registros intermedios que almacenan las salidas de cada etapa que se convertirán en entradas de la siguiente.

En la Figura 2 se muestra el resultado de segmentar el pipeline.

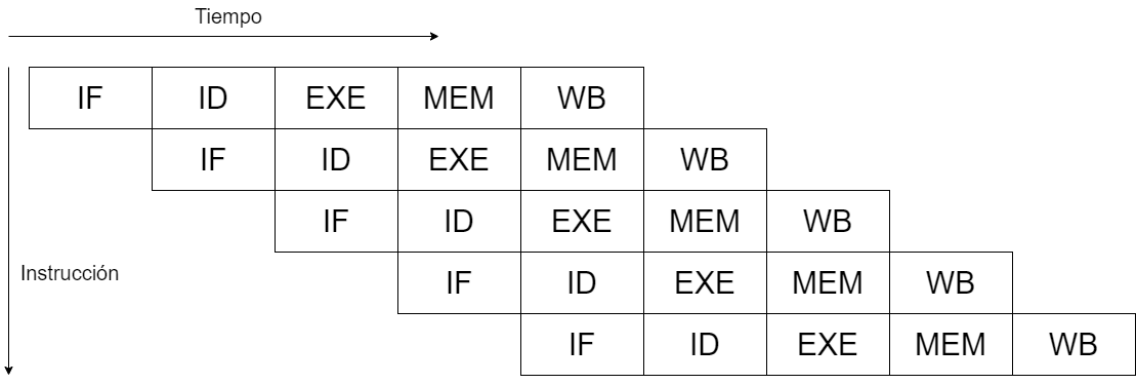


Figura 2. Ejecución de instrucciones en un pipeline segmentado de 5 etapas

Como se puede observar, la segmentación permite que en el caso ideal se ejecuten 5 instrucciones simultáneamente. Al estar la velocidad del microprocesador marcada por la velocidad del reloj, y la velocidad del reloj por la lógica más lenta, el segmentar el pipeline acelera notablemente la lógica combinacional permitiendo relojes más rápidos.

Idóneamente segmentar el *pipeline* en 5 etapas aumentaría en 5 la velocidad, esto no es cierto porque no todas las etapas son igual de rápidas y por tanto el aumento de velocidad no es tan alto. Además, con la segmentación aparecen una serie de inconvenientes que ralentizan notablemente el procesador y pueden hacer que no funcione correctamente, a estos problemas se les conoce como riesgos o *hazards*.

2.4.1 Riesgos asociados a la segmentación

Existen tres tipos principales de riesgos debidos a segmentación:

- Riesgos estructurales o *structural hazards.*
- Riesgos de datos o *data hazards.*

- Riesgos de control o *control hazards*

Los riesgos estructurales se generan cuando diferentes componentes del *hardware* intentan utilizar un mismo recurso de manera simultánea, causando que el componente no funcione adecuadamente.

Los riesgos de datos son causados cuando hay dependencias de datos entre instrucciones. A continuación, se detallan los tipos de riesgos.

Los riesgos de control se producen en los saltos, ya que hasta que no se decodifican no se sabe qué instrucción se ha de ejecutar, si la siguiente o la de destino del salto.

2.4.1.1 ***Structural Hazards***

Son problemas que surgen cuando un mismo recurso de *hardware* es necesario para ejecutar dos operaciones en el mismo ciclo de reloj. Esto causa que una de las dos operaciones deba esperar al siguiente ciclo de reloj para ser resuelta. La forma más sencilla de solucionar el problema es aumentando el número de recursos de *hardware* disponibles permitiendo así la ejecución en paralelo.

Esto podría ocurrir teóricamente en RISC-V, pero la arquitectura está diseñada de tal forma que hay hardware duplicado para realizar ciertas operaciones, dichas operaciones son:

- Almacenamiento dinámico de datos: Si en lugar de memorias separadas para datos e instrucciones se utilizase una memoria única, no podrían extraerse simultáneamente una instrucción y un dato. Esto podría solucionarse duplicando la lógica de lectura o, como se hace en RISC-V, duplicando las memorias y separándolas.
- Suma de números: Hay ciertas operaciones que requieren sumar simultáneamente dos números en la etapa de ejecución, esto podría causar que dichas instrucciones paralizasen el microprocesador durante un ciclo para ser resueltas. Se soluciona de

manera sencilla añadiendo un sumador en paralelo a la ALU para poder ejecutar estas instrucciones.

2.4.1.2 ***Data Hazards***

Los *data hazards* son una serie de inconvenientes que aparecen en un *pipeline* al segmentarlo ya que se establecen dependencias entre los datos dentro del *pipeline* donde puede requerirse para una etapa un dato que debe obtenerse en una etapa posterior. Los tipos de *data hazards* son:

- **Read after write (RAW):** Una instrucción intenta leer un dato antes de que este dato se haya escrito. Este es el tipo de *hazard* que aparece en la segmentación en 5 etapas de RISC-V.
  Este tipo de data hazard se ejemplifica en la Figura 3.

```
i1: ADD x3, x2, x1
i2: ADD x5, x4, x3
```

Figura 3. RAW data hazard

Cuando el microprocesador busca utilizar el dato del registro x3 para sumarlo con el dato del registro x4, el resultado de la primera instrucción aún está siendo calculado en la etapa de ejecución y por tanto la resolución será incorrecta.

Para solucionarlo se pueden utilizar dos técnicas diferentes: paralizar el microprocesador (*stall*) durante los ciclos necesarios para que el dato se escriba y continuar con la instrucción de lectura cuando la de escritura esté finalizada o hacer

que el dato que se necesite leer se envíe directamente desde la etapa en que se calcula a la etapa donde se necesita (*forwarding*).

El *stall* puede realizarse mediante software utilizando la pseudoinstrucción NOP. Las pseudoinstrucciones son instrucciones que no forman parte de la arquitectura, pero pueden formarse utilizando otras instrucciones. En el caso de la NOP, que es una pseudoinstrucción que no hace absolutamente nada simplemente se intenta escribir el valor 0 en x0 a través de la instrucción ADDI x0, x0, 0.

El uso de esta instrucción resulta en la introducción de una burbuja o *bubble* en el *pipeline*.

Se denomina burbuja a una o varias instrucciones que paralizan por completo el *hardware* de la etapa del procesador en la que se encuentran. Reciben este nombre dada la similitud a introducir una burbuja de aire en una tubería, lo que bloquea el paso de fluido en el punto donde se encuentra la burbuja.

El segundo método para evitar el *data hazard RAW* es utilizar un módulo de *forwarding* que detecte cuando instrucciones cercanas entre si tienen dependencia entre ellas y envíe el dato al elemento de *hardware* que lo necesite.

El *forward* requiere de un módulo *hardware* (*Forwarding Unit*) que detecte el riesgo de datos y lo solvente y por tanto es más caro de implementar además de poder reducir ligeramente la velocidad del reloj. El *stall* es menos eficiente pero mucho más sencillo de implementar ya que se puede hacer mediante software además de hardware.

- **Write after Read (WAR):** Una instrucción intenta escribir un dato en un registro mientras otra instrucción intenta leer un dato en ese registro. Este tipo de *hazard* se ejemplifica en la Figura 4.

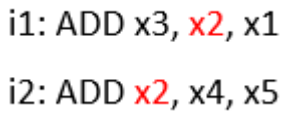


Figura 4. WAR Data Hazard

Esto ocurre en microprocesadores con arquitecturas donde se puede realizar la escritura en etapas anteriores a la lectura. En la CPU diseñada todas las instrucciones que interactúan con un mismo tipo de unidad de almacenamiento de datos (banco de registros o memoria) tienen la misma duración y la lectura se realiza antes que la escritura siempre. Por lo tanto, es imposible que se produzca un error relacionado con esta dependencia entre datos.

- **Write after write (WAW):** Una instrucción intenta escribir un dato en un registro antes de que la instrucción anterior haya escrito un dato en ese registro. Este tipo de *hazard* se ejemplifica en la Figura 5.

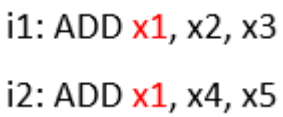


Figura 5. WAW Data Hazard

Esto sólo ocurre en arquitecturas con ejecución en paralelo donde dos instrucciones busquen utilizar el mismo registro simultáneamente para escritura. Esto no ocurre en la CPU diseñada dado que ninguna instrucción puede estar ejecutándose de manera simultánea a otra en una misma etapa.

2.4.1.3 ***Control hazards***

Son un tipo de riesgo debido a las instrucciones de control o salto. Se generan debido a que cuando el pipeline está segmentado y se decide que se va a realizar un salto, las etapas anteriores a donde se realiza esta decisión han empezado a ejecutar instrucciones incorrectas. Para evitar esto existen métodos como el introducir stalls por software en el programa después de cada salto o posible salto, utilizar predicción de saltos, saltos retardados o como en este caso, introducir stalls mediante hardware.

Este método consiste en que cuando se detecta que se va a producir un salto se elimina el trabajo realizado por las etapas anteriores a Execution y se ejecuta la instrucción correcta en el próximo ciclo.

2.5 FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Una FPGA es un circuito integrado diseñado específicamente para que la lógica interna del dispositivo pueda ser reconfigurada por el usuario. Para lograr esto se modifican físicamente las interconexiones entre los diferentes bloques de lógica contenidos dentro del circuito.

La tecnología FPGA nace en 1984 de manos de la empresa Xilinx al desarrollar la tecnología de dispositivos de lógica programable o PLD (Programmable Logic Devices) para conseguir prestaciones similares a las de la tecnología de circuitos integrados de aplicación específica o ASIC (Application Specific Integrated Circuits) sin incurrir en sus altos costes.

Hoy en día algunos de los principales fabricantes de FPGA son Altera (Intel), Xilinx, Microsemi (Microchip) y Lattice ofreciendo productos en un rango de precios que oscila

entre los pocos euros y los miles de euros. En cuanto a FPGA de bajo coste, la familia Cyclone de Altera es una de las opciones predilectas por su relación calidad precio.

La FPGA utilizada para el diseño del microprocesador ha sido la Cyclone II EP2C20F484C7.

Para establecer la estructura interna de una FPGA se utiliza un lenguaje de descripción de hardware o HDL (Hardware Description Language) el cual define de manera unívoca la estructura interna del circuito diseñado. Los dos principales lenguajes de uso más extendido son VHDL (*Very high speed integrated circuits Hardware Description Language*) y Verilog. Ambos son lenguajes normalizados por el IEEE (Institute of Electrical and Electronics Engineers) y por tanto la práctica totalidad de herramientas de diseño ofrecen soporte para uno o ambos de ellos.

Una vez se ha descrito la lógica que será implementada en una FPGA se debe recurrir a un sintetizador, programa encargado de traducir el diseño a nivel de transferencia de registros o RTL (Register Transfer Level) a un nivel de puertas lógicas para poder ser escrito en la FPGA.

2.6 QUARTUS II LITE EDITION

Quartus II conocida como Altera Quartus II antes de que Intel absorbiese Altera, es una herramienta de diseño de FPGA que cuenta con diversas funcionalidades para diseño e implementación de FPGA.

Quartus II tiene un entorno de desarrollo en el que diseñar componentes de FPGA tanto con lenguaje de descripción como con una interfaz gráfica interactiva, cuenta con sintetizador para transformar el diseño en HDL a un diseño a nivel de puerta lógica, herramienta de

análisis temporal para establecer el periodo del reloj, herramienta de simulación de circuitos y un sistema para implementación física del diseño en tarjetas FPGA.

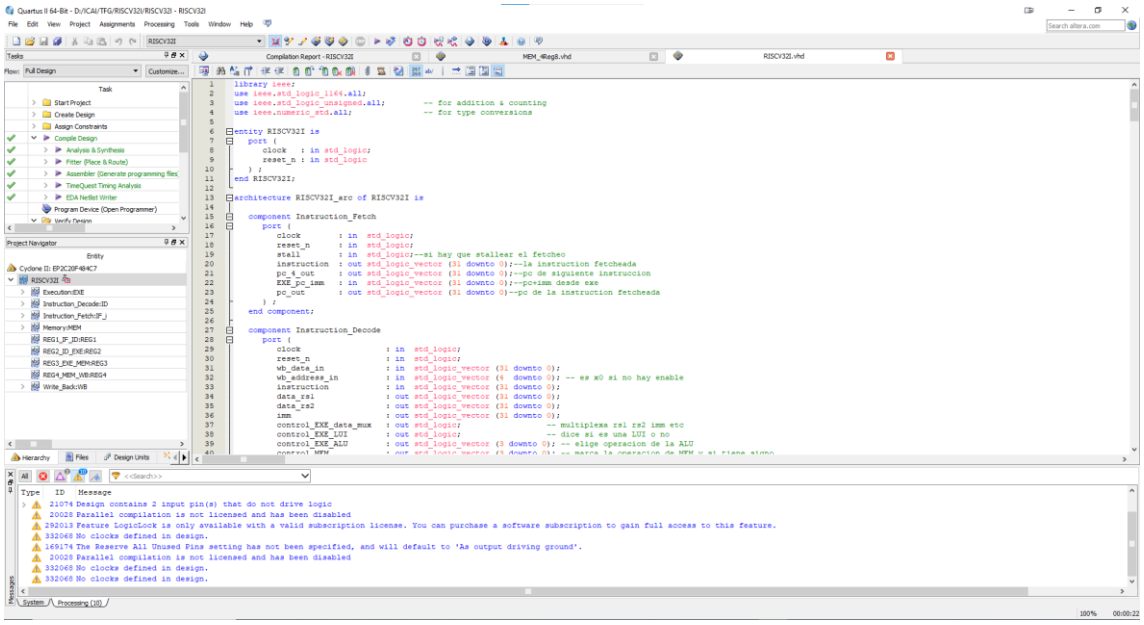


Figura 6. Entorno de desarrollo de Quartus II Lite Edition.

Además, Quartus es compatible con la herramienta de simulación ModelSim, que en su versión ModelSim-Altera se utiliza para simular diseños de FPGA implementables en tarjetas FPGA de Altera como la Cyclone II EP2C20F484C7 utilizada.

La versión de Quartus II utilizada para la realización de este proyecto ha sido Quartus II Lite Edition la cual cuenta con soporte para el lenguaje VHDL.

Quartus permite además la inicialización de memorias a un valor que el usuario puede elegir, esto servirá durante el diseño del microprocesador para escribir las instrucciones en código máquina que la CPU debe ejecutar. Para generar estas instrucciones se han utilizado dos programas open source externos: *RARS* y *hex2vhdl*.

2.7 RARS

RARS es una herramienta open source de simulación de instrucciones de ensamblador de RISC-V, permite generar código en ensamblador en archivos con extensión *.asm.* El lenguaje ensamblador es una forma de representación unívoca del código máquina que utilizan las CPU para codificar instrucciones. Se utiliza para evitar trabajar con instrucciones codificadas en binario a la hora de programar.

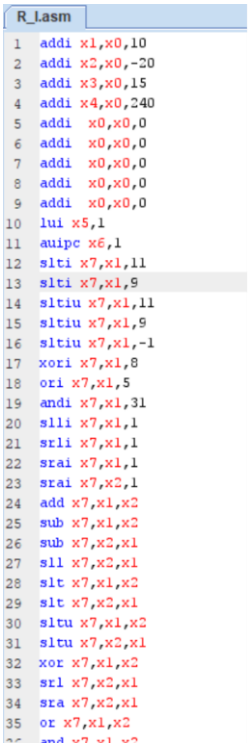


Figura 7. Entorno de desarrollo de RARS.

RARS cuenta además con una herramienta de simulación de la CPU que permite observar la evolución temporal de las variables de un programa instrucción a instrucción.

Name	Number	Value
zero	0	0x00000000
ra	1	0x0000000a
sp	2	0xffffffec
gp	3	0x0000000f
tp	4	0x000000f0
t0	5	0x00001000
t1	6	0x00401028
t2	7	0x00000008
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400094

Figura 8. Herramienta de simulación de rars

Otra de las funcionalidades de RARS es la capacidad de generar archivos de texto plano con extensión *.txt* que contienen la lista de instrucciones de un programa de ensamblador traducido a código máquina.

2.8 HEX2VHDL

Se trata de un programa open source escrito en lenguaje Python que genera una memoria ROM descrita en VHDL y que contiene las instrucciones codificadas en hexadecimal que extrae de un archivo de texto plano. En combinación con RARS permite traducir código ensamblador a, una memoria ROM que contiene código máquina descrito en VHDL.

Para utilizar esta herramienta se recurre a la consola de comandos del ordenador, en este caso la consola de comandos de Windows, y se ejecuta *hex2vhdl* utilizando el archivo que contiene las instrucciones, en este caso el archivo es *asmrom.txt*. Para facilitar el proceso se ha creado un pequeño programa de comandos de Windows llamado *ROMGenerator.bat* el cual simplemente hay que ejecutar.

El contenido del programa es:

```
D:
cd D:\ICAI\TFG\ensamblador
python hex2vhdl.py asmrom.txt
pause
```

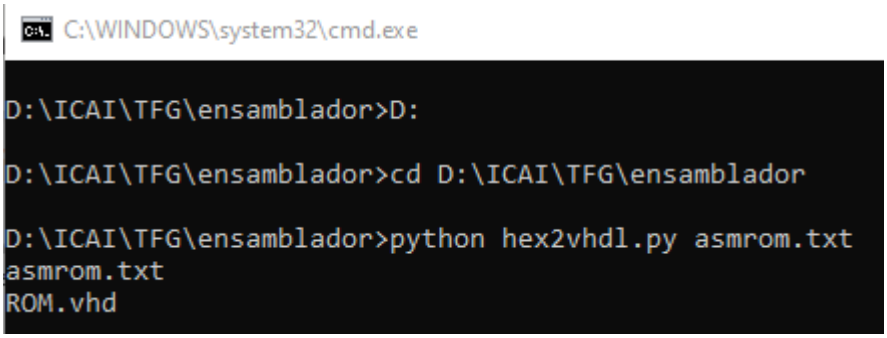


Figura 9. Resultado de la ejecución de ROMGenerator.bat

Una vez ejecutado el programa se genera un archivo ROM.vhd que contiene el diseño de la ROM con las instrucciones del programa de ensamblador.

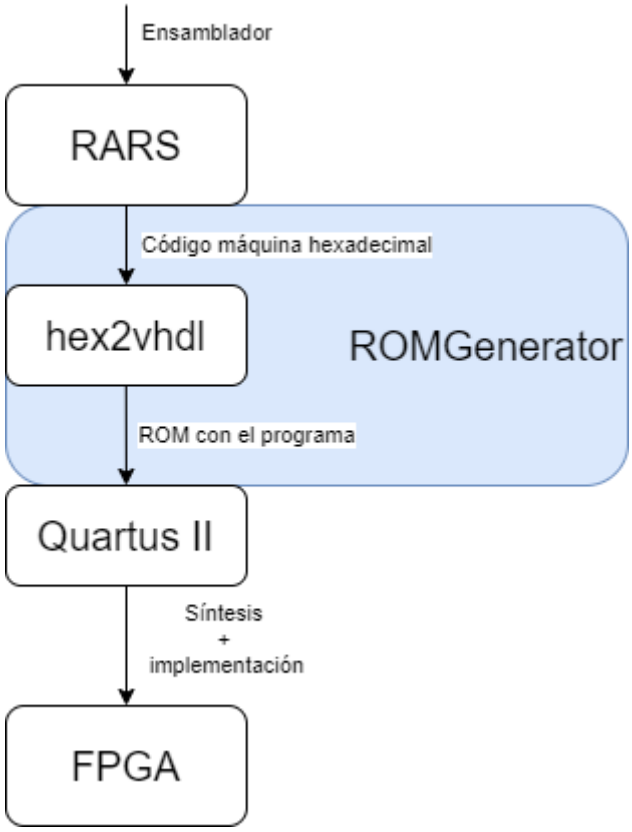


Figura 10. Proceso de implementación del código a nivel físico

Capítulo 3. ESTADO DE LA CUESTIÓN

El uso de la tecnología FPGA está cada vez más extendido a lo largo del mundo. Su bajo coste de entrada y su versatilidad la convierten en uno de los principales protagonistas a nivel de desarrollo tecnológico e industrial.

Debido también al aumento de importancia de las disciplinas STEM (Science Technology Engineering and Mathematics) en el ámbito académico es cada vez más necesario el acceso a microprocesadores por parte del grueso de la población para poder desarrollar pequeñas aplicaciones.

Además, con la existencia de internet y el aumento de las aplicaciones tecnológicas basadas en electrónica digital y concretamente en pequeños microprocesadores es cada vez más común el acceso a repositorios de código de carácter open source donde encontrar programas con diversas funcionalidades creados por otros usuarios y compartidos al público.

Esta combinación de disminución de precio de las FPGA, aumento de la necesidad de microprocesadores con fines didácticos y aumento de acceso a programas que necesitan de estos microprocesadores hace que el desarrollo de microprocesadores *softcore* implementables en FPGA se haya popularizado mucho recientemente.

Este aumento en el desarrollo de microprocesadores para FPGA y el fuerte carácter académico del mismo fue lo que motivó a la Universidad de California a crear el proyecto RISC-V y ha sido tal el éxito que la RISC-V International se ha extendido en más de 50 países y su ISA es utilizada de manera global para fines académicos. Además, debido a la filosofía de RISC-V de tener pocas funcionalidades, pero un alto grado de eficiencia, esta ISA se ha convertido también en una importante fuente de diseños a nivel industrial.

Es fácil encontrar ejemplos de diseño de microprocesadores RISC-V como el CV32E40P, CVA6 o el Ibex que son de dominio público, cada uno enfocado a diferentes funciones, con diferentes extensiones y todos ellos con colaboraciones de decenas de personas.

Además de los propios microprocesadores, son cada vez más comunes las aplicaciones de soporte para RISC-V, aplicaciones como RARS cuyo único objetivo es facilitar el acceso al diseño en RISC-V al gran público.

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

Como se ha comentado en el capítulo anterior las principales ramas de desarrollo de RISC-V enfocado a FPGA son las de elementos de hardware y software de soporte, ambos de carácter open source y de libre acceso y uso al público.

Este proyecto se ha enfocado a la primera rama de desarrollo y en concreto al desarrollo de un microprocesador RV32I completamente funcional implementable en FPGA. Es necesario que se desarrollen este tipo de proyectos para que siga aumentando el acceso al público de herramientas de computación. Los microprocesadores habituales que se encuentran en el mercado se basan en su gran mayoría en tecnologías protegidas por patentes de empresas privadas. Esto causa que el acceso a ordenadores de propósito general para comunidades con bajo nivel de recursos económicos y especialmente vulnerables se vea limitado. Además, al ser la informática uno de los pilares básicos del sistema económico global, y que tiene cada vez más peso, la falta de acceso a un ordenador no hace más que aumentar la desigualdad entre la población más desfavorecida.

Esto coloca a los procesadores para FPGA en una posición privilegiada que permite dar libre acceso al aprendizaje en áreas STEM a cada vez más gente. Se trata de una tecnología barata y altamente versátil que permite al usuario entender conceptualmente y de manera práctica el funcionamiento de los sistemas digitales complejos, cada vez más habituales en el mundo.

El desarrollo de microprocesadores para FPGA junto a todas las herramientas de software de soporte que se encuentran disponibles y que se están desarrollando configuran una herramienta muy importante para la lucha por un acceso a una educación de calidad en todo el mundo.

Además, la ISA RISC-V no solo ha demostrado su utilidad como herramienta didáctica, sino que al tratarse de una microarquitectura de bajo consumo y alto rendimiento es una opción a tener en cuenta a la hora de diseñar un sistema de control industrial. Esto dota de un alto grado de versatilidad a los diseños de CPU basadas en RISC-V para FPGA.

4.2 OBJETIVOS

Los principales objetivos perseguidos durante el desarrollo del proyecto son:

- El completo entendimiento del funcionamiento de un microprocesador con un set de instrucciones reducido. Es un objetivo fundamental que forma las bases del proyecto, de él depende completamente el siguiente objetivo.
- El desarrollo de un microprocesador implementable en FPGA con capacidad para ser utilizado en ámbitos académicos e industriales. Este es el objetivo principal del proyecto y de él dependen los dos siguientes.
- La instalación de herramientas de desarrollo que permitan compilar código ensamblador y ejecutarlo con la CPU. Este objetivo es necesario para poder verificar el funcionamiento de la CPU y dotar a los usuarios que la utilicen de las herramientas de desarrollo.
- Colaboración en el proyecto RISC-V aportando el código de descripción de la CPU con carácter open source y de libre acceso al público. Al cumplir este objetivo se pretende colaborar en la lucha por el libre acceso a la información y la tecnología.

4.3 METODOLOGÍA

Para lograr los objetivos mencionados en el punto anterior se ha seguido la siguiente metodología.

En primer lugar, se ha realizado un estudio de la ISA RISC-V, intentando comprender cada instrucción base, su funcionamiento y el motivo de su existencia. En segundo lugar, se ha diseñado un sistema digital complejo capaz de ejecutar todas las instrucciones base de RISC-V. Posteriormente se ha tratado de crear un proceso capaz de, por medio de software de terceros, traducir ensamblador a código máquina e implementar el código máquina resultante en la CPU. Por último, se han publicado los resultados del proyecto en un repositorio de acceso público en la plataforma GitHub para que puedan ser utilizados y manipulados por cualquier usuario.

4.4 ESTIMACIÓN ECONÓMICA

Para la estimación económica se tiene únicamente en cuenta el coste directo de la implementación en FPGA en base a la cual se ha desarrollado el proyecto. Esto quiere decir que se ha calculado utilizando como referencia el precio de la tarjeta de desarrollo DE1 de Terasic basada en la FPGA Cyclone II EP2C20F484C7, no se ha considerado la adquisición de otra FPGA ni de versiones de pago de herramientas de desarrollo. Además, no se consideran costes de operación como la electricidad consumida por la FPGA.

Producto	**Cantidad**	**Coste unitario (€/ud)**	**Coste total (€)**
Cyclone II EP2C20F484C7	1	294	294
		TOTAL	294

Tabla 8. Tabla de costes del producto

Capítulo 5. SISTEMA DESARROLLADO

Para el diseño de RV32I se han diseñado primero las 5 etapas del pipeline de manera independiente como bloques de VHDL y se han combinado junto a los 4 registros intermedios para formar el *top level* del microprocesador. Cada una de las etapas está compuesta de una serie de componentes o bloques más simples con funcionalidades concretas que se explicarán en detalle en el apartado correspondiente.

Las entradas del sistema son:

- **Reloj:** Es el componente que permite sincronismo en el diseño, es la base de cualquier lógica secuencial y en última instancia marca la velocidad del microprocesador. Genera una onda cuadrada en cuyos flancos de subida se actualizan todos los datos síncronos. Los elementos síncronos que tienen el reloj como entrada se marcan con un pequeño triángulo siendo está la notación habitual. Esta forma de señalar elementos sincronizados por reloj se ilustra en la Figura 11 con el ejemplo del *Program Counter* o Contador de Programa.

Figura 11. Representación del reloj como señal de entrada

- **Reset asíncrono:** el reset asíncrono o *reset_n* es una entrada de un bit que se utiliza para limpiar completamente los registros del microprocesador y reiniciarlo. Se utiliza cada vez que el microprocesador se inicia.
- **Instrucciones:** Las instrucciones son la principal entrada del sistema, marcan cuales son las operaciones a ejecutar por el microprocesador y su resolución es el objetivo mismo de una CPU. En el caso del procesador diseñado la memoria de instrucción

es de tipo ROM (Read Only Memory) y se escribe al programarla en la FPGA sin poderse modificar después. Esta forma de escritura se ha elegido por su simplicidad y para evitar la necesidad de software adicional, sin embargo, lo habitual en un procesador es que la memoria cuente con lógica de escritura y se utilice un *hardware* externo (un programador) que escriba las instrucciones en memoria o un bootloader que permita cargar el programa mediante una comunicación serie.

En cuanto a las salidas, el RV32I diseñado no tiene salidas ya que no se ha instalado ningún módulo que le permita comunicarse con periféricos. Hay que tener en cuenta que el núcleo diseñado está pensado para integrarse en un sistema final en el que se requeriría de algún componente que permita mostrar físicamente las salidas del procesador, como LEDs o un monitor y una forma de comunicarse con el periférico.

En la Figura 12 se muestra un diseño simplificado del microprocesador segmentado.

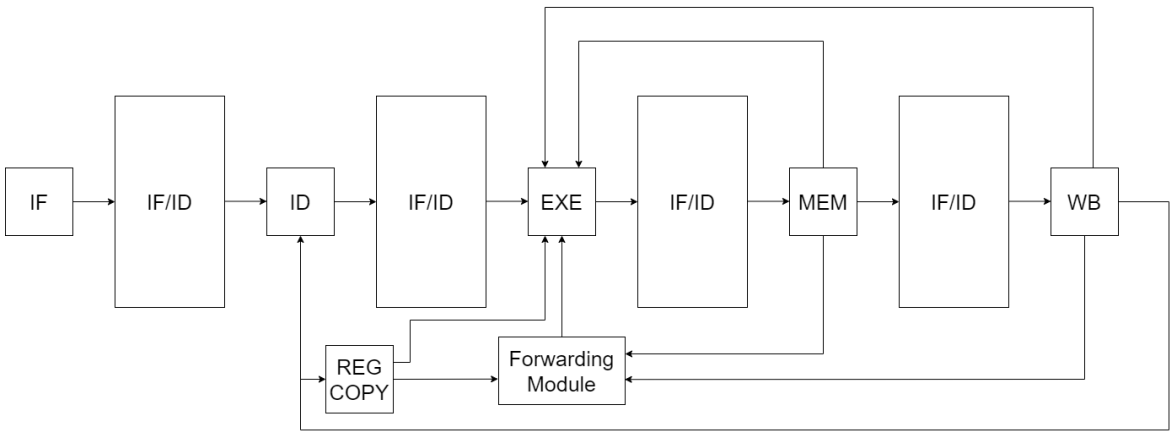


Figura 12 Diagrama RTL del microprocesador segmentado

5.1 INSTRUCTION FETCH.

La primera etapa del pipeline es la etapa IF o Instruction Fetch. Es aquí donde se busca en la memoria de programa la siguiente instrucción para ser ejecutada por el microprocesador. Este bloque consiste en:

- **Contador de programa:** Registro que almacena la dirección de memoria que debe ser leída en el siguiente ciclo de reloj
- **Sumador**: Bloque combinacional que suma el PC actual con 4 para obtener la siguiente dirección
- **Memoria de Programa:** Memoria ROM que almacena todas las instrucciones de un programa se genera mediante RARS+hex2vhdl.
- **Multiplexores**: Seleccionan datos en base a instrucciones de control.

Las entradas a la etapa de Instruction Fetch son:

- **Control de saltos:** El control de saltos o *control jump* es una señal de control de un bit de ancho generada en la etapa Execution. Tiene valor 0 cuando no se va a realizar un salto en el contador de programa y valor 1 cuando se realiza un salto (ya sea un salto condicional o incondicional). Esta señal controla ambos multiplexores, limpia el registro intermedio entre las dos primeras etapas y genera un *stall* de un ciclo en el contador de programa y la memoria de programa.
- **PC en caso de salto:** Es el valor del contador de programa en caso de realizarse un salto, se calcula en la etapa Execution y se escribe en el registro de PC cuando la señal de control de saltos lo indica.
- **Reloj.**
- **Reset asíncrono.**

Las salidas de esta etapa son:

- **Instrucción:** Es la siguiente instrucción por ejecutar, se envía a la etapa Instruction Decode sin pasar por el registro intermedio. Esto se hace así por motivos de sincronismo, al ser la memoria de programa una unidad de lógica secuencial la salida de la misma ya está sincronizada con el registro. Si la instrucción se escribiese y leyese del registro intermedio se retrasaría un ciclo de reloj respecto al contador de programa correspondiente a dicha instrucción.

- **PC:** Es el valor numérico codificado en 32 bits de la dirección de memoria de una instrucción. RISC-V tiene direccionamiento de memoria por bytes y por tanto un aumento de una palabra (longitud de las instrucciones) de 32 bits corresponde a un aumento de 4 en el PC. Esta señal se utiliza en etapas posteriores para hacer cálculos relativos a PC y así poder direccionar saltos.
- **PC+4:** Se trata del valor del PC de la instrucción siguiente, esta señal se utiliza en los saltos incondicionales (*jump and link* y *jump and link register*) para almacenar la dirección de retorno. Para generar la señal simplemente se utiliza la señal PC, pero se hace un *bypass* al primer registro intermedio, por lo que se convierte en el PC de la instrucción siguiente.

La estructura de esta primera etapa se muestra en el diagrama de la Figura 13.

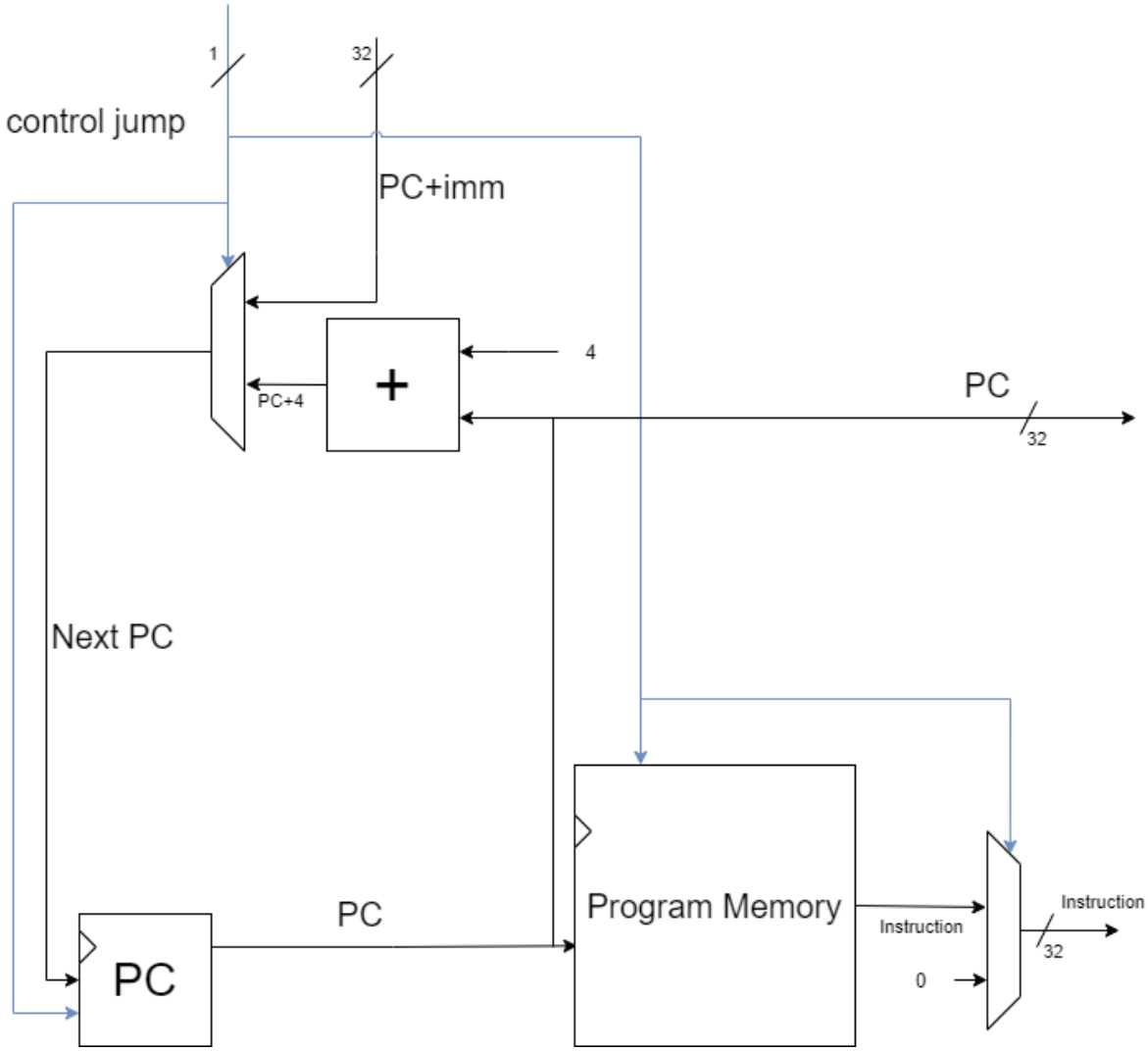


Figura 13. Diagrama RTL de Instruction Fetch

A continuación de describen en detalle los bloques que forman la primera etapa del pipeline.

1.3.1 CONTADOR DE PROGRAMA

El contador de programa o Program Counter (PC) es un registro de 32 bits de ancho con *clear* y *stall.*

A un registro de n bits se le pueden añadir además ciertas funcionalidades de control que permiten escoger si se escribe en el registro, si se lee su valor, si su valor se cambia a 0, etc.

Estas funcionalidades adicionales se pueden combinar dando lugar a multitud de registros diferentes, las funcionalidades más comunes son:

- **Enable**: Señal de control que permite elegir si se escribe el dato de entrada del registro en el siguiente ciclo.
- **Clear**: Permite fijar el valor del registro a un valor concreto (típicamente 0 para reiniciarlo), puede ser síncrono o asíncrono.

En el caso del PC, tiene un clear para poder reiniciarlo al hacer un Reset del sistema y un stall (el stall es un enable negado) para poder mantener el valor del registro de un ciclo a otro.

Las entradas del *Program Counter* son:

- **Siguiente PC**: Siguiente valor a escribir en el registro, ya sea el valor anterior del registro más 4 o un valor calculado en la etapa Execution.
- **Reset asíncrono.**
- **Reloj.**

La única salida del bloque es:

- **PC:** Es el valor de la dirección de la siguiente instrucción, se utiliza para calcular el valor del siguiente PC y para direccionamiento de la ROM de instrucciones. Además, como se ha explicado anteriormente se realiza un *bypass* del primer registro para sincronizarla con la instrucción anterior en caso de salto incondicional.

El diagrama del bloque PC se muestra en la Figura 14.

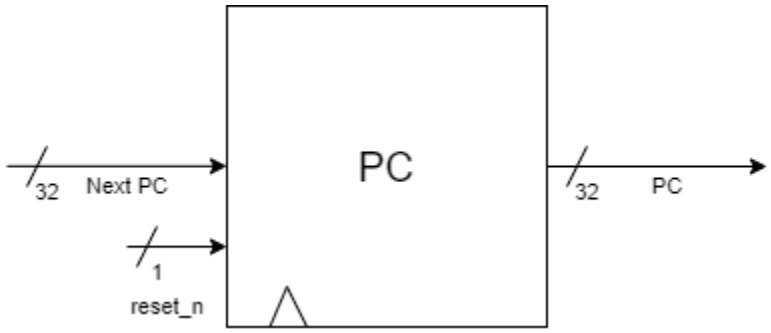


Figura 14. Diagrama RTL del program counter.

1.3.2 SUMADOR

Se trata de un bloque combinacional muy sencillo cuyo único objetivo es sumar 4 al valor del PC actual. Se utiliza para calcular el PC correspondiente a la siguiente instrucción en caso de que no haya ningún salto.

Las dos entradas de este bloque son:

- **PC:** El valor de salida del registro del contador de programa.
- **4:** Valor estático codificado en un ancho de 32 bits. En caso de utilizarse un ancho de palabra diferente a 32 este valor estático se calcularía dividiendo el ancho de palabra entre el ancho del direccionamiento a memoria de la arquitectura (8 en caso de RISC-V que direcciona a nivel de byte).

La salida del bloque es:

- **PC+4:** Valor que tendrá el registro PC si no se realiza ningún salto. Este valor se multiplexa junto a PC+imm en base a la instrucción de control de saltos.

El diagrama del bloque sumador se muestra en la Figura 15.

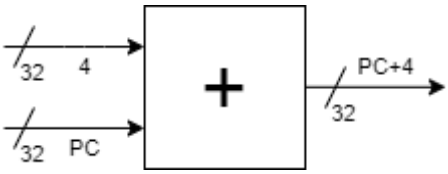


Figura 15. Diagrama RTL del sumador.

1.3.3 MEMORIA DE PROGRAMA

Como ya se ha explicado anteriormente lo habitual en una memoria de programa de un ordenador de propósito general como este es utilizar una memoria de tipo flash que permita escritura mediante comunicación por *hardware* con un programador externo (métodos habituales son la comunicación por puerto serie o la comunicación por USB).

Para realizar la escritura de la memoria de programa en este caso se ha recurrido a la capacidad de la FPGA de inicializar los componentes secuenciales a valores determinados durante la implementación física en placa. Para generar el código máquina escrito en la memoria se ha recurrido a programas externos *open source* como ya se ha explicado.

La memoria diseñada tiene una capacidad de almacenamiento de 512 bytes o 128 instrucciones y por tanto un ancho de 7 bits de dirección. 128 instrucciones es una cantidad bastante baja para un programa normal escrito en lenguajes de alto nivel como C pero suficiente para los objetivos conceptuales perseguidos en este diseño. Además, la cantidad de memoria es fácilmente ampliable en el código de descripción de la memoria por lo que resultaría sencillo convertir el diseño en plenamente funcional.

Las entradas de la memoria son:

- **Dirección:** Es el valor de 7 bits sin signo que permite acceder a la lectura de las instrucciones. La dirección se obtiene del *Program Counter* eliminando los bits más significativos hasta llegar a la longitud de la dirección de memoria.
- **Reloj.**

La salida de la memoria es:

- **Instrucción:** La instrucción codificada en 32 bits que va a ejecutar el microprocesador a continuación.

El diagrama del bloque de memoria se muestra en la Figura 16.

Figura 16. Diagrama RTL de la memoria de programa

1.3.4 MULTIPLEXOR DE PC

Es un bloque combinacional sencillo que selecciona el valor de PC que se escribe en el registro durante el siguiente flanco de reloj. Se controla mediante la variable de *control jump*.

Las entradas de este bloque son:

- **PC+4**: Valor del PC de la instrucción siguiente en caso de que no haya saltos.
- **PC+imm**: Valor del PC siguiente en caso de que se realice un salto, se calcula en la etapa *Execution.*
- **Control de salto:** Es la variable de control del multiplexor (de un solo bit ya que solo hay 2 entradas). Toma valor 1 cuando se va a producir un salto en la siguiente instrucción y 0 cuando el flujo del programa va a ser el normal.

La salida del bloque es:

- **Siguiente PC**: Es el valor que se escribirá en el registro del contador de programa durante el siguiente flanco de subida.

El diagrama del multiplexor se muestra en la Figura 17.

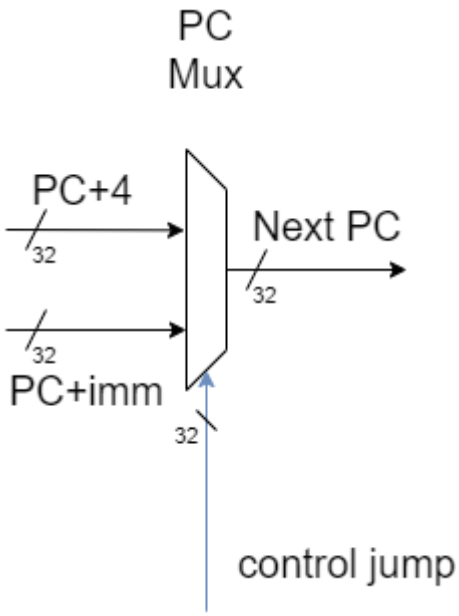


Figura 17. Diagrama RTL del multiplexor de PC.

1.3.5 MULTIPLEXOR DE INSTRUCCIÓN

Un multiplexor 2x1 (2 entradas una salida) igual que el de PC. Decide que instrucción se envía a la etapa siguiente, puede tener como salida la instrucción que corresponde al PC actual o una instrucción "vacía" que trata de escribir en x0 el valor 0 (NOP).

En caso de efectuarse un salto, el microprocesador decide que va a realizar el salto durante la etapa Execution, por tanto, las dos instrucciones que en ese momento se encuentran en Instruction Fetch e Instruction Decode no deben ejecutarse. Este multiplexor elimina la

instrucción que se encuentra en la etapa de Instruction Fetch. Para eliminar la instrucción de Instruction Decode se borra el contenido del primer registro intermedio.

Este proceso de borrar ciertos valores que se encuentran en el pipeline se conoce como *flush.*

En caso de tener en la instrucción 1 un salto condicional el flujo del pipeline con y sin flush se muestra en las figuras 18 y 19.

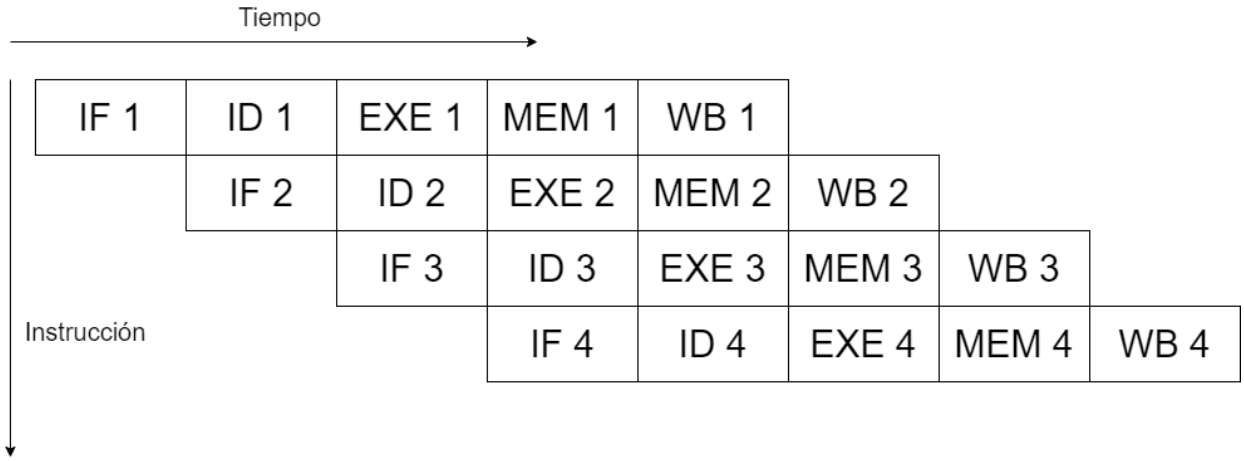


Figura 18. Salto sin flush.

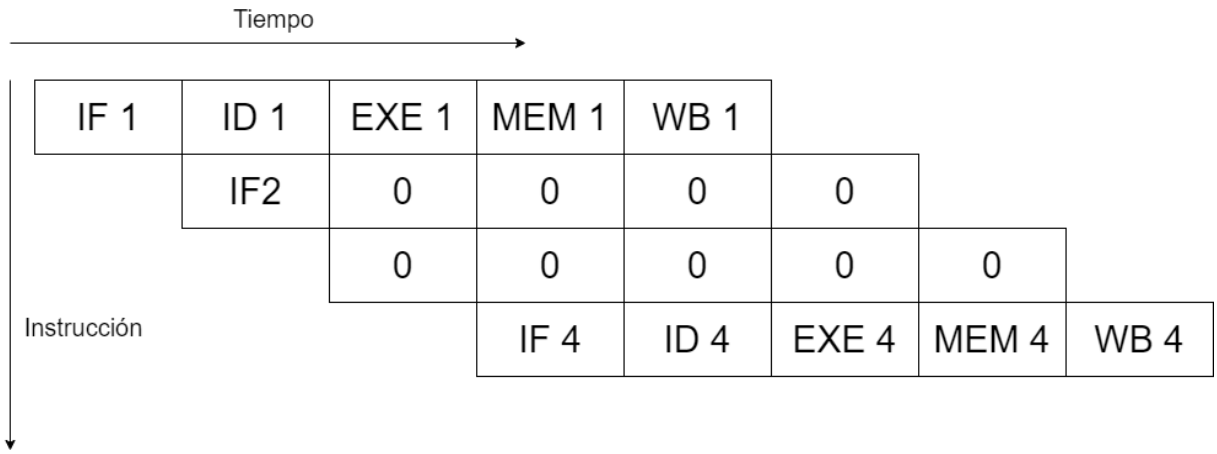


Figura 19. Salto con flush

Se puede apreciar como en caso de no realizar el *flush* del pipeline las instrucciones 2 y 3 se ejecutarían incorrectamente.

- Las entradas del multiplexor son:
- **Instrucción:** En caso de no haber salto.
- **Vector de 32 bits a 0:** En caso de realizarse un salto.
- **Control de salto:** Es la señal que decide si hay un salto o no. Toma valor 1 para saltos y valor 0 para flujo normal.

La salida del multiplexor es:

- **Instrucción siguiente:** Es la instrucción que llega a la etapa Instruction Decode

El diagrama del multiplexor de instrucción se muestra en la Figura 20.

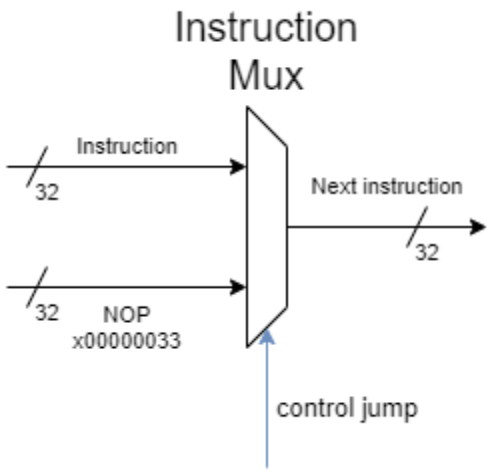


Figura 20. Diagrama RTL del multiplexor de instrucción.

5.2 INSTRUCTION DECODE

Instruction Decode es la segunda etapa del pipeline segmentado. Es la etapa que contiene la inmensa mayoría de los elementos del control path. En ella se utiliza la instrucción codificada en 32 bits para extraer toda la información que contiene y generar señales de control que utilizarán el resto de los elementos en el data path para ejecutar la instrucción. Es junto a Execution la etapa más compleja y con más consumo de hardware debido a la cantidad de lógica combinacional en la unidad de control.

La etapa consta de los siguientes elementos:

- **Unidad de control o *Control Unit*:** Es la encargada de generar las señales de control que regulan el funcionamiento del microprocesador. RISC-V está diseñado de manera específica para aligerar el consumo de *hardware* en la decodificación de instrucciones ya que es una etapa que puede consumir muchos recursos si la ISA no está bien optimizada.
- **Generador de inmediatos:** Es un módulo combinacional que recibe una señal de control y la instrucción para decodificar el inmediato requerido.
- **Banco de registros:** Contiene los 32 registros de trabajo del microprocesador donde se escriben y leen las variables.

La única entrada a esta etapa es:

- Instrucción: La instrucción codificada de 32 bits que se va a descomponer para generar todas las señales de control

Las salidas de la etapa son:

- **Control de rama:** Variable de control de 2 bits que indica si la instrucción tiene salto condicional, incondicional o de ningún tipo.

- **Control de multiplexor de datos:** Señal de control de 1 bit que indica si la instrucción a ejecutar utiliza como dato de operación el valor del registro fuente 2 o un inmediato.
- **Control LUI:** Indica si la operación es LUI (Load Upper Immediate), en caso de serlo toma valor 1 y el valor del registro fuente 1 se sustituye por 0 en la etapa Execution.
- **Control de la ALU:** Señal de 4 bits de ancho que indica que operación debe realizar la ALU en Execution.
- **Control de memoria:** Tiene 4 bits de ancho, el primero marca si la operación tiene signo o no tomando valor 1 o 0 respectivamente. Los 3 bits menos significativos indican si se trata de una instrucción de carga o almacenamiento y de que tipo.
- **Control de selector de escritura:** Elige que dato se va a escribir en el registro destino durante la etapa Write Back.
- **Control de habilitación de escritura:** Esta señal vale 1 cuando la instrucción realiza una escritura en registro y 0 en caso contrario.
- **Dirección de escritura**: Es la dirección del registro destino donde se va a escribir durante Write Back.
- **Valor de registro fuente 1.**
- **Valor de registro fuente 2.**

El diagrama RTL de la etapa se muestra en la Figura 21.

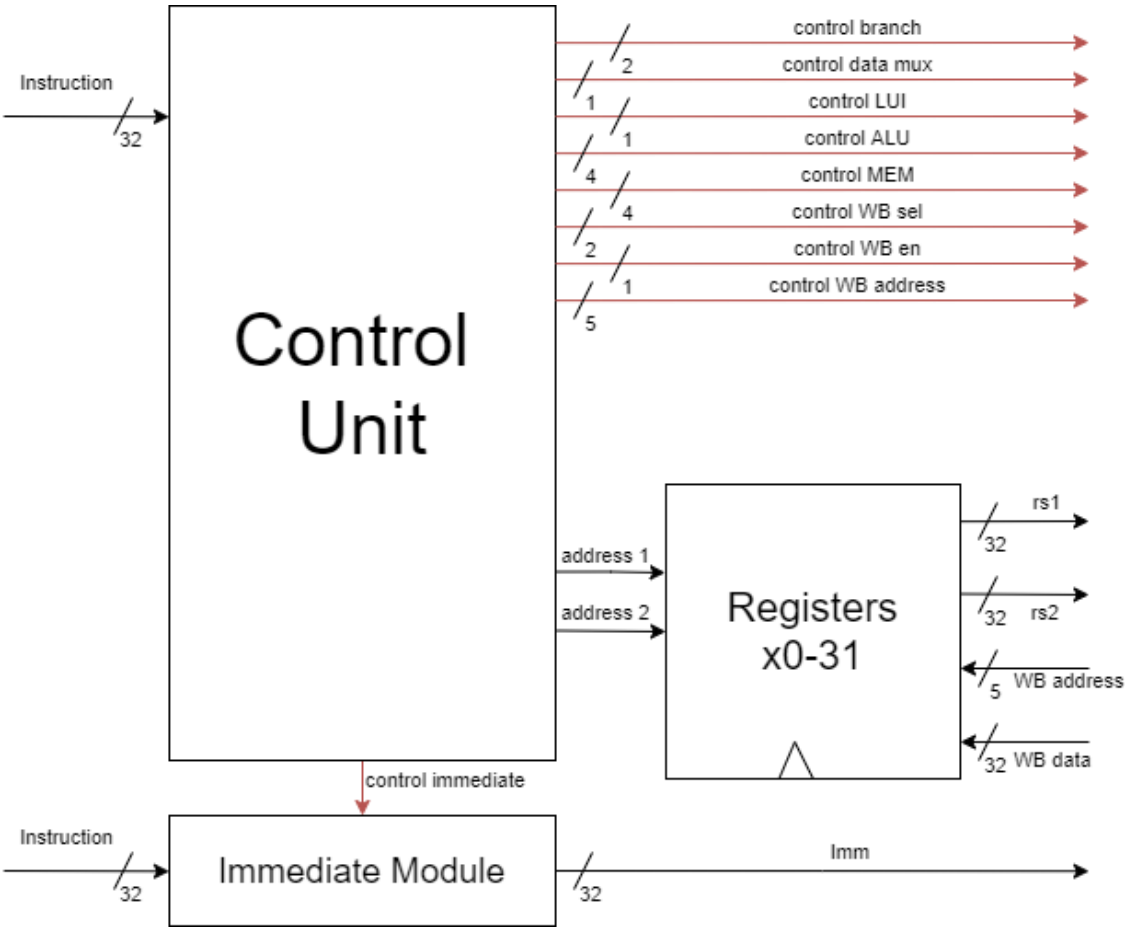


Figura 21. Diagrama RTL de Instruction Decode.

A continuación, se explica detalladamente el funcionamiento de cada componente que compone la etapa.

5.2.1 UNIDAD DE CONTROL.

La unidad de control o *Control Unit* es el elemento principal de la etapa, decodifica todos los campos de la instrucción salvo el inmediato y genera todas las señales de control del microprocesador salvo la señal de control de saltos. Es un elemento indispensable en cualquier diseño de electrónica digital ya que junto al data path forma toda la lógica a nivel de registro.

Las entradas y salidas del bloque son esencialmente las mismas que las entradas y salidas de la etapa por lo que para evitar caer en la redundancia sólo serán explicadas en este apartado aquellas señales que sean internas de la etapa. El resto de las señales simplemente se nombrarán y serán explicadas en detalle en los bloques que las utilicen.

La única entrada de la unidad de control es:

- **Instrucción.**

Las salidas de la unidad de control son:

- **Control de inmediatos:** Indica a la unidad de generación de inmediatos el tipo de instrucción que se ha decodificado.
- **Dirección de registro fuente 1.**
- **Dirección de registro fuente 2.**
- **Control de rama.**
- **Control de multiplexor de datos.**
- **Control LUI.**
- **Control de la ALU.**
- **Control de selector de escritura.**
- **Control de habilitación de escritura.**
- **Dirección de escritura**.

El diagrama RTL del bloque se muestra en la Figura 22.

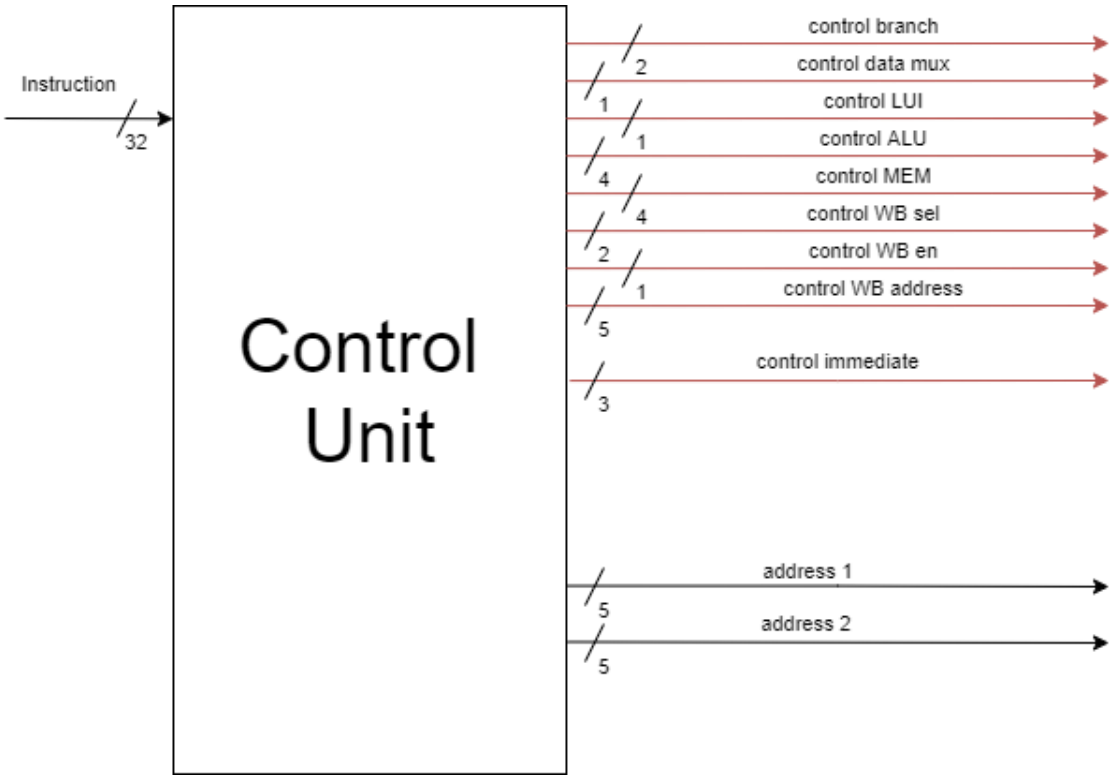


Figura 22. Diagrama RTL de la unidad de control.

5.2.2 GENERADOR DE INMEDIATOS

En este componente se decodifica el inmediato que está escrito en los diferentes campos de la instrucción. Las instrucciones de RISC-V han sido diseñadas para que la localización de los inmediatos sea consistente entre instrucciones.

Esto facilita enormemente la decodificación y disminuye el consumo de hardware requerido. A cambio de este menor uso de recursos ciertas instrucciones tienen los inmediatos fraccionados y separados en diferentes campos dentro de la instrucción e incluso desordenados dentro de un mismo campo.

Tomando como ejemplo las instrucciones de tipo U, B y J que se muestran en la Tabla 9 se puede observar que la posición de ciertos bits como el más significativo, que marca el signo,

o los bits del 10 al 5 se encuentran siempre en las mismas posiciones. Esta disposición de los inmediatos acelera la lógica de codificación y la simplifica a cambio de volver más complejo el diseño.

Tipo	31:25	24:20	19:15	14:12	11:7	6:0
U	imm (31:12)				rd	opcode
B	imm (12|10:5)	rs2	rs1	funct3	imm (4:1|11)	opcode
J	imm (20|10:1|11|19:12)				rd	opcode

Tabla 9. Configuración de las instrucciones U, B y J.

Las entradas del generador de inmediatos son:

- **Control de inmediatos:** Señal de control que indica el tipo de instrucción que se va a decodificar. Los posibles valores de la señal y su significado se muestran en la Tabla 10.

Control de inmediato	Tipo de instrucción
001	U
010	J
011	I y Load
100	B

101	S
110	I(shamt)
111	JALR
000	Nada

Tabla 10. Control de inmediatos

- **Instrucción.**

La única salida del generador de inmediatos es:

- **Inmediato:** Es un valor numérico directamente codificado en la instrucción, la mayoría de las instrucciones utilizan un inmediato, ya sea para comparaciones, uso de constantes definidas en el programa, saltos relativos, etc.
- El inmediato generado en cada instrucción se muestra en la Tabla 11.

Tipo de instrucción	**Inmediato generado**
U	Instruction (31:12) & ZE
J	SE & instruction (31|19:12|20|31:12)
I y Load	SE & instruction (31:20)
B	SE & instruction (31|7|30:25|11:8) & ZE
S	SE & instruction (31:25|11:7)
I(shamt)	ZE & instruction (24:20)

JALR	SE & instruction (31:20) & 0
Nada	0

Tabla 11. Señal de control de inmediatos.

La notación utilizada es la notación estándar de VHDL donde & representa la operación de concatenación, SE la extensión de signo y ZE la extensión de ceros. El cero resultante de una instrucción vacía tiene ancho de 32 bits, no se representa así para facilitar la lectura.

El diagrama del bloque se muestra en la Figura 23.

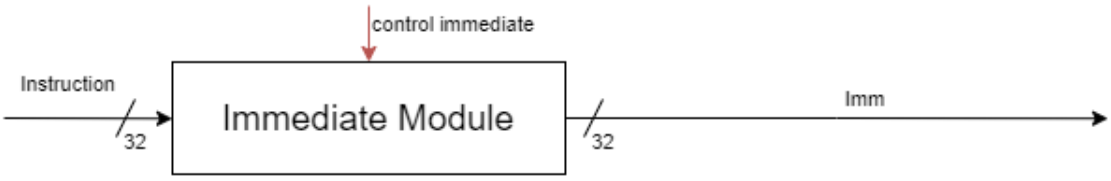


Figura 23. Diagrama RTL del generador de inmediatos.

5.2.3 BANCO DE REGISTROS

El banco de registros es la unidad de almacenamiento de datos principal del microprocesador. Su tamaño reducido implica que la lógica de direccionamiento es sencilla y rápida además de no consumir cantidades excesivas de espacio en las instrucciones para codificar las direcciones.

De los registros se obtienen los datos relacionados a las variables de un programa, tanto las locales como las globales además de almacenarse valores relevantes para el flujo del programa como la dirección de retorno al hilo principal del programa tras ejecutar una subrutina.

El banco está compuesto de 32 registros de 32 bits, los registros se direccionan a nivel de palabra, no a nivel de byte como las memorias.

Aparte de los 32 registros el banco se compone de un multiplexor para escritura y dos demultiplexores para lectura. La señal de control es la correspondiente a los registros fuente y destino para lectura y escritura respectivamente.

La lectura y la escritura no son simultáneas en una instrucción, la lectura se realiza durante la etapa de Instruction Decode mientras que la escritura se realiza al final del pipeline durante el Write Back.

Las entradas del banco de registros son:

- **Dirección de registro fuente 1.**
- **Dirección de registro fuente 2.**
- **Dato de escritura.**
- **Dirección de registro destino:** En caso de no ser una instrucción de escritura se direcciona al registro x0.

Las salidas del banco de registros son:

- **Valor del registro fuente 1.**
- **Valor del registro fuente 2.**

La estructura del banco de registros se muestra en la Figura 24.

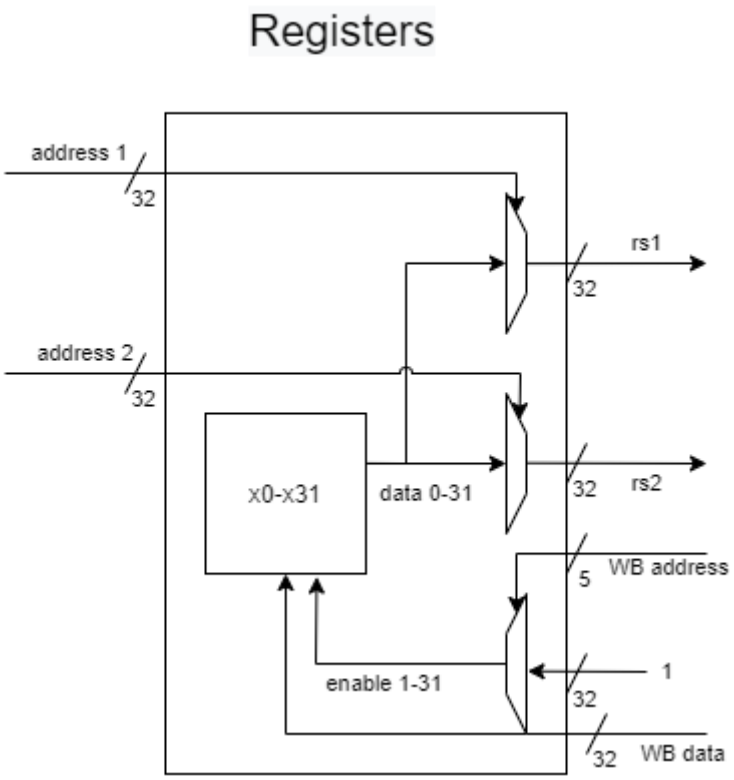


Figura 24. Diagrama RTL del banco de registros.

5.3 EXECUTION

La etapa de ejecución es donde se realizan los cálculos necesarios para determinar el resultado de una instrucción. Es la etapa con mayor consumo de hardware debido principalmente a la unidad aritmeticológica o ALU que contiene.

En esta etapa se genera también la señal de control de saltos necesaria para las instrucciones de control de flujo del programa.

Los componentes que conforman esta etapa son:

- **Unidad aritmético lógica o ALU:** Es la calculadora del microprocesador, resuelve todas las operaciones necesarias para ejecutar instrucciones, tiene un ancho de 32 bits dos entradas y una salida. Puede realizar 14 operaciones diferentes y por tanto se controla con una señal de 4 bits.
- **Sumador:** Es un bloque combinacional con el único objetivo de sumar los valores de PC y el inmediato, se utiliza en las instrucciones de salto donde la ALU tiene que calcular si se realiza un salto o no.
- **Unidad de control de saltos:** Genera la señal de control de saltos en base al control de ramas generado por la unidad de control y al resultado de la ALU.
- **Multiplexor de datos:** Selecciona uno de los datos de entrada de la ALU. Puede seleccionar el inmediato o el valor del registro fuente 2.
- **Multiplexor LUI:** Se utiliza en la instrucción LUI para sustituir el valor del registro fuente 1 por 0 como entrada de la ALU.
- **Multiplexores de forwarding:** Seleccionan entre el valor leído de los registros fuente y los valores de etapas posteriores del pipeline para adelantar un dato si fuese necesario

Las entradas de la etapa son:

- **Control de ramas.**
- **Control de multiplexor de datos.**
- **Control de la ALU**
- **Control LUI**
- **Valor del registro fuente 1**
- **Valor del registro fuente 2**
- **Contador de Programa**
- **Inmediato.**
- **Valor en etapa MEM.**
- **Valor en etapa WB.**

- **Valor en etapa ID.**

Las salidas de la etapa Execution son:

- **Resultado de la ALU:** Valor calculado por la ALU ejecutando la operación pertinente.
- **Valor de registro fuente 2:** Necesario también para operaciones de carga en memoria.
- **PC+Inmediato:** Valor necesario en las instrucciones de salto para escritura en el contador de programa y para la instrucción AUIPC para escritura en el banco de registros.
- **Control de saltos:** Señal de control que indica si en el siguiente flanco de reloj se realiza un salto en el valor del contador de programa.

El diagrama del diseño RTL de Execution se muestra en la Figura 25.

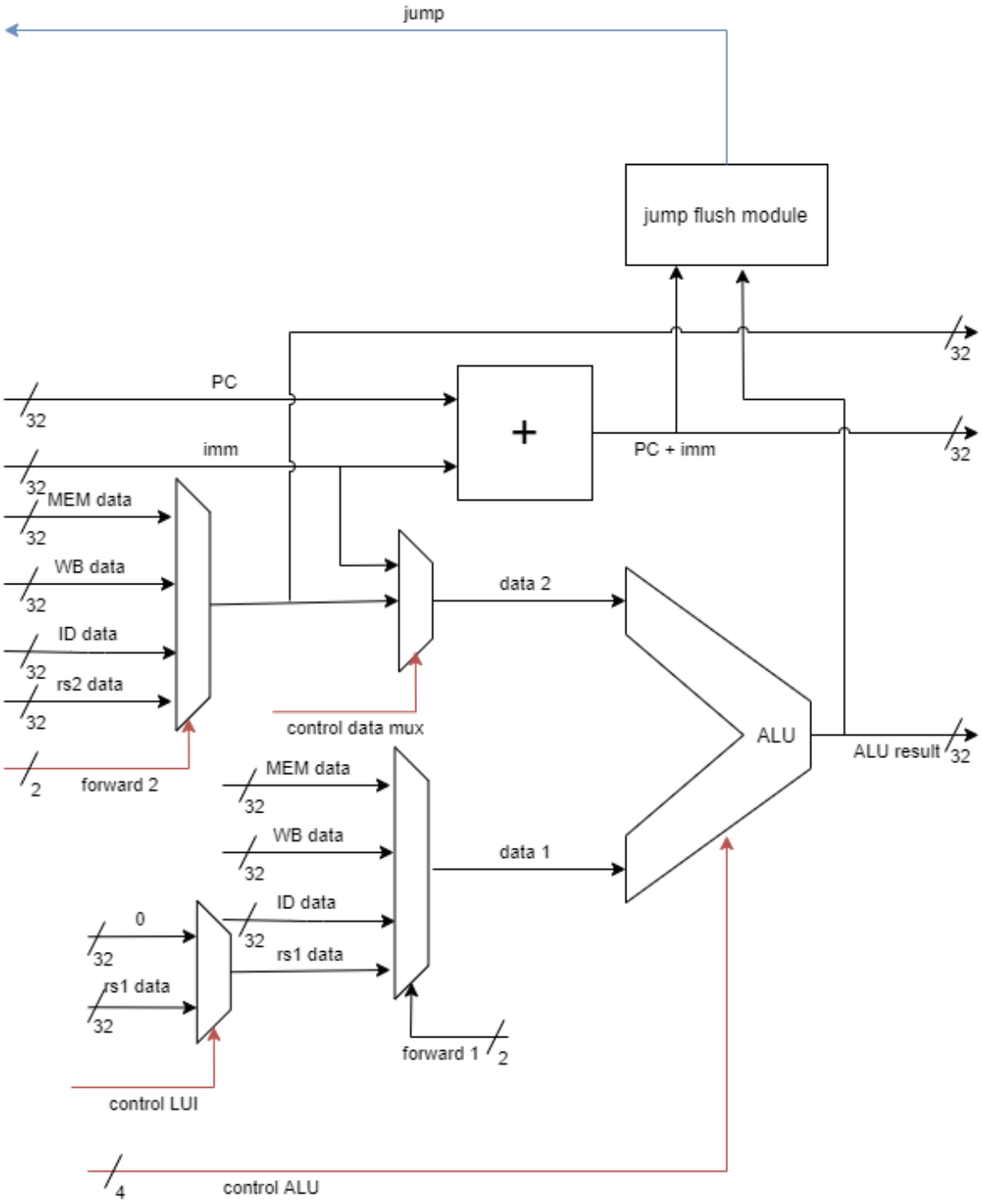


Figura 25. Diagrama RTL de Execution

A continuación, se detallan los componentes que forman la etapa.

5.3.1 UNIDAD ARITMÉTICO LÓGICA.

La unidad aritmético lógica o ALU es el componente combinacional con mayor consumo de hardware del microprocesador, se trata de una ALU de 32 bits por lo que los mayores valores que puede utilizar sin desborde son 4.294.967.296 para enteros sin signo y entre -2.147.483.648 y 2.147.483.647 para enteros con signo codificados en complemento a 2.

La ALU realiza operaciones aritméticas sencillas (suma y resta), operaciones de lógica bit a bit (or, xor y and), comparaciones con y sin signo (igualdad, desigualdad, menor que, mayor que, mayor o igual que) y desplazamientos.

El total de operaciones que puede realizar son 14 además de una operación que pone su salida a 0 que únicamente se utiliza cuando una instrucción está vacía o mal escrita. Al utilizar 15 operaciones en total requiere de una señal de control de 4 bits que decide que operación se ejecuta en cada instrucción.

Los datos que utiliza la ALU se seleccionan en los dos multiplexores de la etapa de manera acorde a la instrucción que se ejecuta.

Las entradas del componente son:

- **Control de la ALU:** Se trata del control de 4 bits de ancho mencionado anteriormente, los posibles valores de esta señal junto a la operación correspondiente que realiza la ALU se muestran en la Tabla 12.

Control ALU	Operación
• **0001**	*Suma*
• **0010**	*Resta*

• **0011**	*And lógico bit a bit*
• **0100**	*Or lógico bit a bit*
• **0101**	*Exclusive or lógico bit a bit*
• **0110**	*Desplazamiento a la izquierda*
• **0111**	*Desplazamiento a la derecha con extensión de ceros*
• **1000**	*Desplazamiento a la derecha con extensión de signo*
• **1001**	*Comparación de tipo menor que sin signo*
• **1010**	*Comparación de tipo mayor o igual que con signo*
• **1011**	*Comparación de igualdad*
• **1100**	*Comparación de desigualdad*
• **1101**	*Comparación de tipo menor que con signo*
• **1110**	*Comparación de tipo mayor o igual que con signo*

Tabla 12. Señal de control de la ALU

- **Dato 1:** Seleccionado entre valor del registro fuente 1 y 0 codificado en 32 bits.
- **Dato 2:** Seleccionado entre valor de registro fuente 2 e inmediato de la instrucción.

La salida de la ALU es:

- **Resultado de la ALU:** Puede ser el valor codificado en 32 bits fruto de la resolución de una operación o, en el caso de las comparaciones, un valor binario 1 o 0.
  En las comparaciones, utilizadas para las instrucciones tipo B de salto condicional y las instrucciones de tipo *set less than*, el resultado de la ALU es 1 en caso de ser cierta la comparación y 0 en caso contrario. Este dato binario sigue teniendo un ancho de 32 bits.
  Este resultado se utiliza para ser escrito registro o para comprobar si se toma una rama en un salto condicional.

El diagrama RTL de la ALU se muestra en la Figura 26.

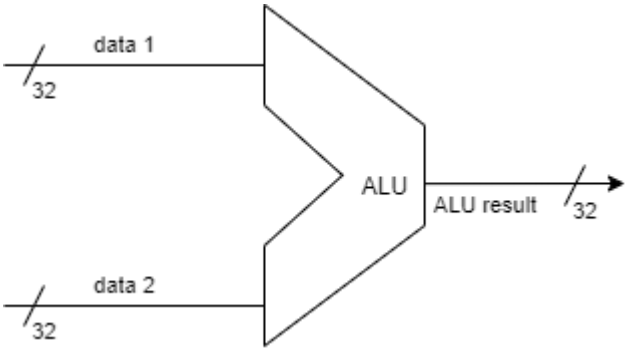


Figura 26. Diagrama RTL de la ALU.

5.3.2 SUMADOR

Sirve para evitar el riesgo de datos estructural que se daría en las instrucciones de salto condicional. Estas instrucciones requieren que la ALU realice una comparación para determinar si se toma una rama y además necesitan que la ALU calcule el salto relativo al contador de programa que debe producirse.

Cuando el mismo hardware se requiere simultáneamente para llevar a cabo dos procesos las soluciones son duplicarlo o retrasar un ciclo la ejecución de uno de los procesos. En este

caso se duplica el hardware necesario para ejecutar las instrucciones, dicho hardware es el módulo sumador de la ALU. Nótese que no se duplica la ALU entera ya que el resto de las operaciones no generan riesgos estructurales ya que ninguna instrucción las requiere por duplicado.

Las entradas del sumador son:

- **PC.**
- **Inmediato.**

La salida del sumador es

- **PC+inmediato.**

El diagrama RTL del sumador se muestra en la Figura 27.

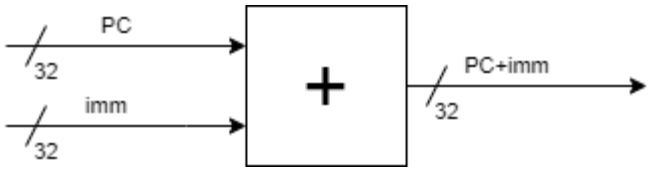


Figura 27. Diagrama RTL del sumador

5.3.3 UNIDAD DE CONTROL DE SALTOS

Junto a la unidad de control conforma la ruta de control del microprocesador y es el componente que diferencia a un microprocesador de una calculadora. Permite ejecutar las instrucciones de control de flujo de programa, ejecutando bloques de instrucciones en función de condiciones lógicas y permitiendo la ejecución de subrutinas externas al programa principal. Es la unidad que genera soporte para todos los *statements* con lógica de decisión de lenguajes de alto nivel y para la llamada a funciones dentro de un programa.

Además, la unidad determina si el salto tiene direccionamiento relativo a un valor de registro o relativo al contador de programa.

Los datos de entrada del componente son:

- **Resultado de la ALU:** Utilizado para decidir si una rama se toma o no en un salto condicional. Únicamente es relevante el bit menos significativo de la ALU ya que como se ha indicado antes el resultado de una comparación es binario.
  En el caso del salto relativo a valor de registro (instrucción jalr) se utiliza el resultado de la ALU para direccionar la memoria de programa.
- **Control de rama:** Señal de control de dos bits de ancho que determina si la instrucción que se ejecuta tiene un salto y el tipo de salto. Los valores de la señal y su significado se muestran en la Tabla 13.

Control de rama	Tipo de salto
00	Instrucción sin salto
01	Salto condicional
10	Salto incondicional jal
11	Salto incondicional jalr

Tabla 13. Señal de control de rama.

- **PC+imm.**

Las salidas de esta unidad son:

- **Nuevo PC:** Es el valor que tendrá el PC durante el siguiente ciclo de reloj si se realiza un salto. Puede ser el valor del PC de la instrucción más un inmediato en saltos relativos a PC y el valor del resultado de la ALU en saltos relativos a un valor escrito en registro.

- **Control de salto:** Es la señal binaria que determina si se genera un salto en el valor de PC. Sirve también para generar el *flush* del pipeline en caso de que se realice el salto.

El diagrama RTL de la unidad de control de saltos se muestra en la Figura 28.

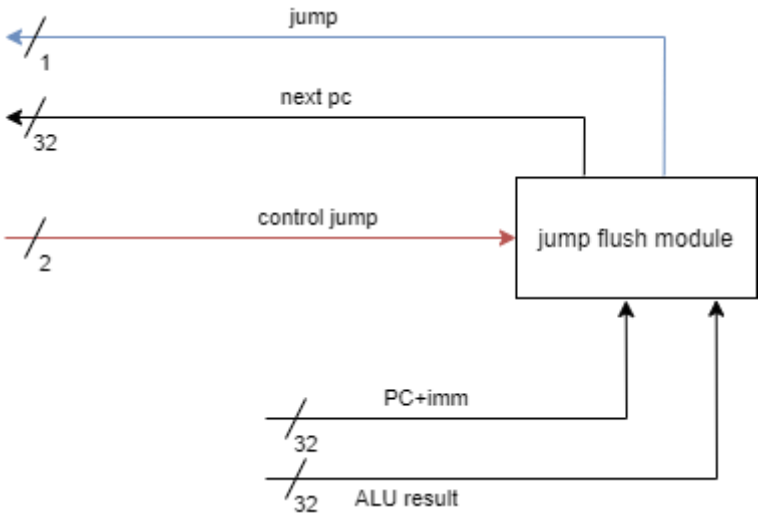


Figura 28. Diagrama RTL de la unidad de control de saltos

5.3.4 MULTIPLEXOR DE DATOS

Es el multiplexor encargado de seleccionar uno de los datos de entrada de la ALU. Puede seleccionar el valor del inmediato de la instrucción o el valor del registro fuente 2. Al ser un multiplexor de dos entradas y una salida se controla con un único bit.

Las entradas del multiplexor son:

- **Valor de registro fuente 2**
- **Inmediato**
- **Control de multiplexor de datos.**

La salida del multiplexor es:

- **Dato 2.**

El diagrama RTL del multiplexor se muestra en la Figura 29.

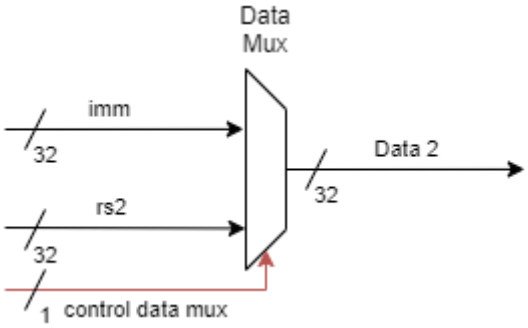


Figura 29. Diagrama RTL del multiplexor de datos.

5.3.5 MULTIPLEXOR LUI

Es el multiplexor que selecciona entre el valor del registro fuente 1 y un 0 de 32 bits en caso de utilizarse la instrucción LUI. Sirve únicamente para esta instrucción ya que requiere el valor del inmediato exclusivamente. La señal de control toma 0 en todas las instrucciones y 1 en la instrucción LUI

Las entradas del multiplexor son:

- **Valor de registro fuente 1**
- **0.**
- **Control de multiplexor LUI.**

La salida del multiplexor es:

- **Dato 1.**

El diagrama RTL del multiplexor se muestra en la Figura 30 .

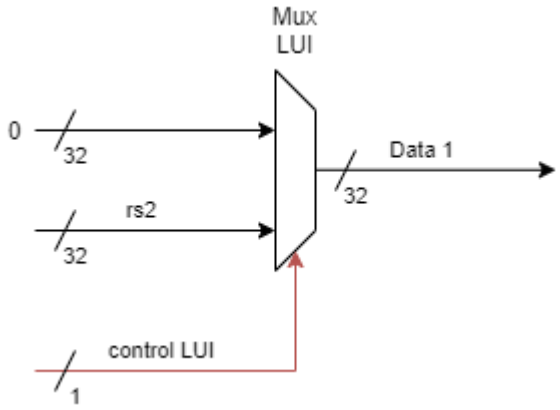


Figura 30. Diagrama RTL del multiplexor LUI.

5.3.6 MULTIPLEXORES DE FORWARDING

Ambos son iguales, eligen entre el dato que se ha leído del registro fuente y los datos en etapas más avanzadas del pipeline.

Las entradas son:

- **Dato de registro fuente.**
- **Dato en MEM.**
- **Dato en WB.**
- **Dato en ID.**

La salida es:

- **Dato de la ALU.**

El diagrama RTL se muestra en la Figura 31.

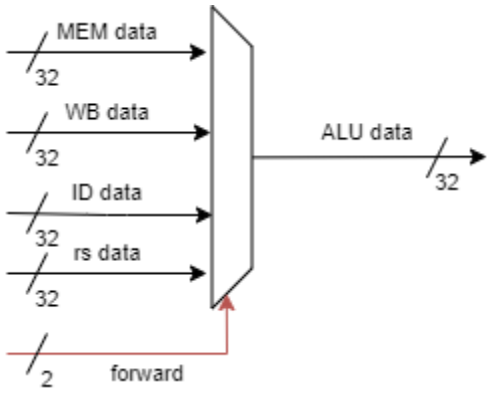


Figura 31. Diagrama RTL de los multiplexores de forwarding.

5.4MEMORY

Es la cuarta etapa en la que se dan los accesos a memoria del microprocesador. Como se ha explicado antes, RISC-V es una ISA de tipo Load/Store por lo que las operaciones de memoria son exclusivamente de carga y almacenamiento. Esto quiere decir que la etapa de memoria sirve única y exclusivamente para almacenar datos en los registros y para cargar datos de la memoria a los registros. Los datos de memoria no se utilizan de manera directa para operaciones ni para resolver condiciones lógicas.

Es una etapa del pipeline cuyo principal consumo de recursos es la memoria RAM de datos que contiene, la memoria puede llegar a ser el elemento más costoso a nivel de hardware de un microprocesador, pero debido a su naturaleza es fácil de estandarizar y existen módulos de memoria para FPGA con los cuales diseñar memorias de gran capacidad sin consumir bloques lógicos de la placa programable para almacenar datos.

Además, en esta etapa se selecciona el dato que se escribirá en la etapa de Write Back mediante un multiplexor.

Los componentes de la etapa son:

- **Unidad de habilitación de bytes:** Es el bloque encargado de escribir en la memoria, se utiliza exclusivamente en operaciones de almacenamiento o stores. Las instrucciones de almacenamiento guardan el valor de un registro en memoria.
- **Unidad de carga:** Es el bloque encargado de las lecturas de memoria, se utiliza exclusivamente en instrucciones de carga o loads. Las instrucciones de carga guardan un valor de la memoria en un registro.
- **Memoria de datos:** Se trata del componente principal de la etapa. Es una memoria RAM cuyo tamaño es variable y fácilmente manipulable mediante el código de descripción del bloque. Al ser una RAM de RISC-V se direcciona a nivel de byte, a diferencia de la memoria de programa que se direcciona a nivel de palabra.
- **Selector de datos de Write Back:** Determina que dato pasa a la etapa de Write Back para ser escrito en registro.

Las entradas de la etapa son:

- **Valor de registro fuente 2:** El valor que se escribe en la memoria si la instrucción es de almacenamiento.
- **Resultado de la ALU:** Indica la dirección de memoria tanto de lectura como de escritura. Puede ser el dato que se utilice en la etapa de Write Back.
- **Control de memoria:** Indica el tipo de operación que se va a realizar. Tiene un ancho de 4 bits de los cuales el más significativo indica si la instrucción tiene signo o no y los 3 menos significativos indican el tipo de instrucción.
- **PC+imm:** Se utiliza en la instrucción AUIPC ya que requiere escribir en registro el resultado de una operación donde el PC es un dato. Las operaciones que utilizan PC como dato no se realizan en la ALU para evitar riesgos estructurales como ya se ha explicado en la sección de Execution.
- **PC+4:** Utilizada en instrucciones de salto incondicional para almacenar la posición de la instrucción anterior a la llamada a una función.

Las salidas de la etapa son:

- **Resultado de la ALU:** Es el caso más común, se utiliza en operaciones entre registros y operaciones con inmediatos.
- **Dato de escritura en registro**: Es el dato que pasa a la siguiente etapa.

El diagrama RTL de la etapa se muestra en la Figura 32.

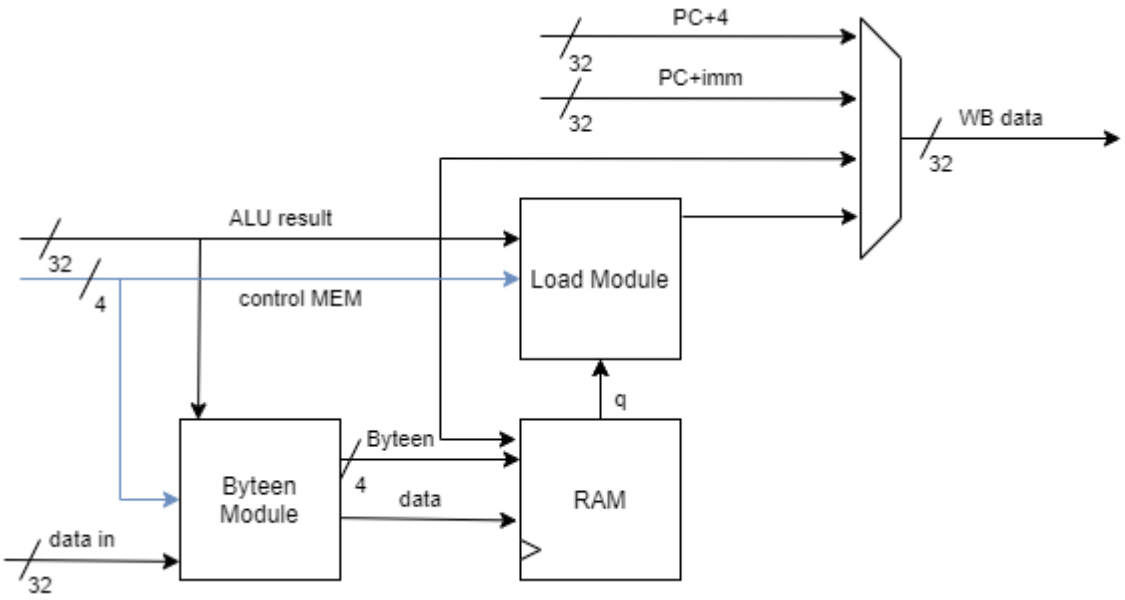


Figura 32. Diagrama RTL de la etapa Memory

5.4.1 UNIDAD DE HABILITACIÓN DE BYTES

Este bloque se utiliza exclusivamente en las instrucciones de almacenamiento, su propósito es generar una máscara al valor de entrada dependiendo del tipo de almacenamiento que requiera la instrucción. Una máscara es una selección concreta de ciertos valores dentro de un vector de bits, se utilizan para aislar ciertas secciones de un valor codificado en binario.

La unidad de habilitación de bytes se controla con la señal de control de memoria, pero utilizando únicamente los 3 bits menos significativos ya que las instrucciones de

almacenamiento no tienen signo. Esto se debe a que el dato de escritura nunca tiene menor tamaño que el conjunto de registros donde se va a escribir y por tanto nunca se realiza una extensión de signo o de ceros.

Las entradas de este bloque son:

- **Control de almacenamiento:** Compuesta por los 3 bits menos significativos del control de memoria, indica que tipo de almacenamiento se va a realizar o si no se realiza ninguna escritura en memoria durante la instrucción.
  Los posibles valores de esta señal y su significado se ilustran en la Tabla 14.

CONTROL DE ALMACENAMIENTO	TIPO DE ALMACENAMIENTO
000	Sin almacenamiento
001	Sin almacenamiento
010	Sin almacenamiento
011	Sin almacenamiento
100	Almacenamiento de Byte
101	Almacenamiento de media palabra
110	Almacenamiento de palabra

Tabla 14. Señal de control de almacenamiento

- **Resultado de la ALU:** Es el valor del que se obtiene la dirección de memoria para el almacenamiento. La cantidad de bits que se utilizan para el direccionamiento depende del tamaño de la memoria y de si permite almacenamientos sin alinear.

En este caso al tratarse de una memoria de únicamente 8 palabras la dirección sólo ocupa 5 bits y al no tener soporte para almacenar datos sin alinear las instrucciones de almacenamiento de medias palabras y de palabras ignoran el bit menos significativo y los dos bits menos significativos respectivamente.

- **Dato de escritura:** Es el dato que se utilizará en las instrucciones de almacenamiento, proviene del registro fuente 2 y requiere de una máscara para poder escribirlo correctamente en memoria.

Las salidas del bloque son:

- **Habilitación de bytes:** Es la habilitación de escritura independiente de cada uno de los 4 módulos de RAM de 1 byte que conforman la memoria. Es una señal de 4 bits en la que cada uno representa un byte de la palabra a la que se está direccionando la memoria. El valor 1 representa que se escribirá en ese byte durante la instrucción mientras el 0 implica que no.
- **Dato de escritura con máscara:** Como se ha explicado en el apartado de memoria del capítulo de ISA RISC-V, una memoria sin soporte para escritura desalineada escribe los valores de entrada en los registros de manera alineada. Es decir, el bit 0 del dato de entrada se escribe en el bit 0 del registro, el bit 1 en el bit 1 y así sucesivamente.

  La cantidad de bits de precisión con la que se puede escribir marca a que nivel se direcciona la memoria. En este caso al tratarse de direccionamiento por bytes, la menor cantidad de bits que se pueden aislar son 8.

  Cuando se desea escribir un byte en una posición que no es la que tiene en el dato de entrada se debe manipular el dato de entrada para alinear ese byte con la posición en el registro.

  Si por ejemplo se quisiese almacenar el byte 0 del dato 0x01234567 en el tercer byte de una posición de memoria sería necesario manipular el dato de entrada para que el byte 0 se encontrase en la posición 3.

Este proceso se ilustra en la Figura 33.

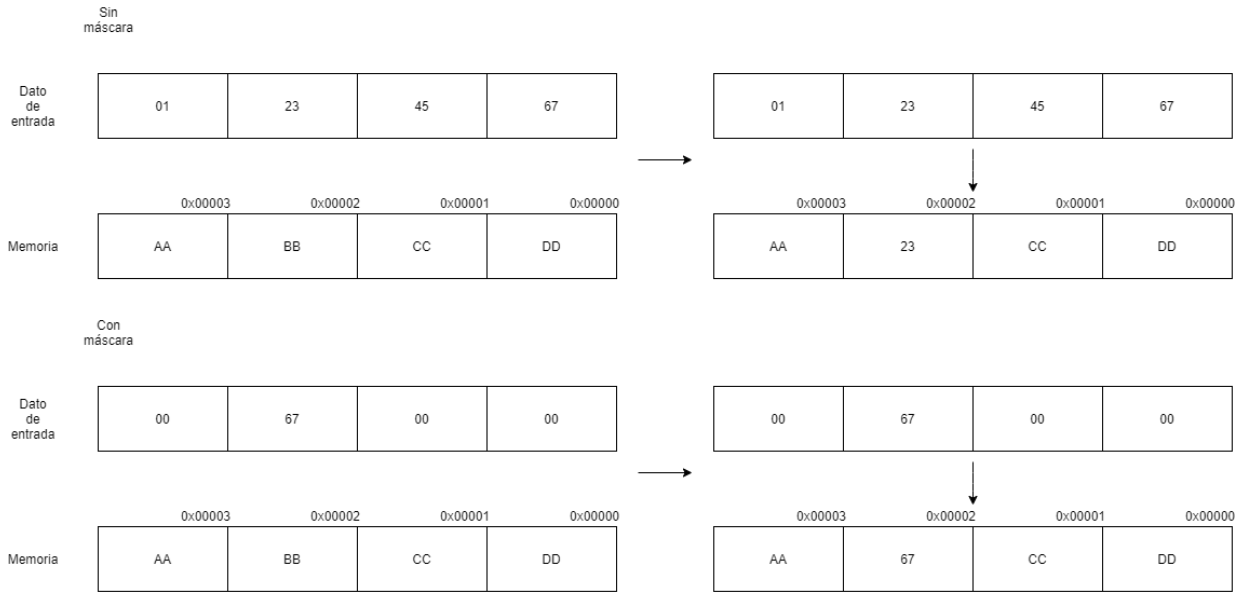


Figura 33. Uso de máscaras para almacenamiento de datos

El diagrama RTL de la unidad de habilitación de bytes se muestra en la Figura 34:

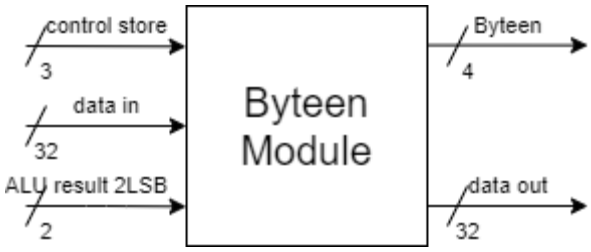


Figura 34. Diagrama RTL de la unidad de habilitación de bytes.

5.4.2 UNIDAD DE CARGA

Tiene un cometido similar a la unidad de almacenamiento salvo que en este caso se utiliza para las instrucciones de carga o loads. Su función es realizar la máscara pertinente al valor obtenido de la memoria para que cumpla las condiciones requeridas por la instrucción de

carga. Para cumplir estas condiciones se seleccionan las partes del dato de lectura pertinentes, se desplazan a los bits menos significativos y se realiza una extensión de signo o de ceros dependiendo de si la instrucción es con signo o no. Las cargas de palabra no tienen extensión ya que la longitud del dato leído coincide de manera natural con la longitud de una palabra.

Se controla con la señal de control de memoria y, a diferencia del caso anterior, se utilizan todos los bits de la señal ya que el bit más significativo marca si la carga es con signo o no.

Las entradas de la unidad de carga son:

- **Dato de lectura:** Es el dato obtenido de la memoria situado en la posición que marca el resultado de la ALU. Requiere de una máscara para ajustar la longitud del dato a la longitud de las palabras de RISC-V. La máscara aplicada depende del tipo de carga y de si es una carga con signo o sin signo
- **Control de carga:** Coincide con la señal de control de memoria, en ella se indica la operación de carga que se va a realiza y si se trata de una operación sin signo o no. El valor de esta señal determina el tipo de máscara a utilizar. Tiene un ancho de 4 bits.

  Los diferentes valores de la señal y sus respectivos significados se muestran en la Tabla 15.

Control de almacenamiento	Tipo de almacenamiento
0000	Sin almacenamiento
0001	Carga de byte sin signo
0010	Carga de media palabra sin signo
0011	Carga de palabra

0100	Instrucción sin carga
0101	Instrucción sin carga
0110	Instrucción sin carga
1001	Carga de byte son signo
1010	Carga de media palabra con signo

Tabla 15. Señal de control de carga

- **Resultado de la ALU:** En los casos de carga de byte y media palabra indica que partes del dato deben escribirse en el registro. Condiciona que máscara debe utilizarse y cuál es el bit más significativo para realizar la extensión de signo. Sólo son relevantes los dos bits menos significativos ya que la precisión de escritura es a nivel de byte y la cantidad de bytes en una palabra es 4. En el caso de las cargas de media palabra solo es relevante el segundo bit menos significativo.

  Los posibles valores de esta señal y sus implicaciones en las instrucciones de carga de byte y media palabra se ilustran para el dato de entrada 0x01234567 y carga de tipo sin signo en las Tablas 16 y 17.

Carga de byte	
ALU result 2LSB	Dato de salida
00	0x00000001
01	0x00000023
10	0x00000045

11	0x00000056

Tabla 16. Máscara de carga de byte

Carga de media palabra	
ALU result LSB	Dato de salida
0	0x00000123
1	0x00004567

Tabla 17. Máscara de carga de media palabra

La salida de la unidad de carga es:

- **Dato de memoria:** Es el dato con la máscara aplicada que se utilizará para escribirse en registro en la etapa de Write Back de las instrucciones de carga.

El diagrama RTL de la unidad de carga se muestra en la Figura 35.

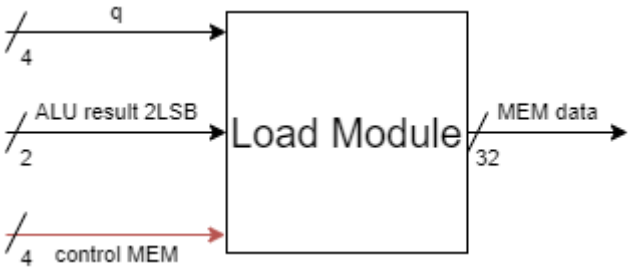


Figura 35. Diagrama RTL de la unidad de carga.

5.4.3 MEMORIA DE DATOS

Es la comúnmente conocida como RAM del ordenador, consiste en 4 memorias de 256 bytes para generar una memoria de 256 palabras de 32 bits (1KB).

La memoria diseñada se escribe a través de una dirección que apunta a nivel de palabra junto a la señal de habilitación de bytes para tener direccionamiento a nivel de byte.

El tamaño de la memoria en cuanto al número de palabras que puede almacenar es variable dependiendo de los recursos de los que se disponga. Para generar memorias de tamaño más grande o pequeño basta con modificar el número de palabras almacenadas en el código de descripción de las 4 memorias de ancho de byte. El límite de tamaño queda marcado por la arquitectura RISC-V ya que la dirección más grande que se puede escribir es tan grande como permita el número de bits de la arquitectura. En este caso son 32 bits por lo que el máximo tamaño es 4GiB.

La memoria como se ha explicado anteriormente no soporta direccionamientos no alineados, por lo que estos deberán ser resueltos vía software.

Las entradas a la memoria de datos son:

- **Reloj**
- **Valor de escritura con máscara**
- **Dirección.**
- **Habilitación de byte.**
- **Habilitación de escritura**

La salida de la memoria de datos es:

- **Valor de lectura sin máscara.**

El diagrama RTL de la memoria de datos se muestra en la Figura 35.

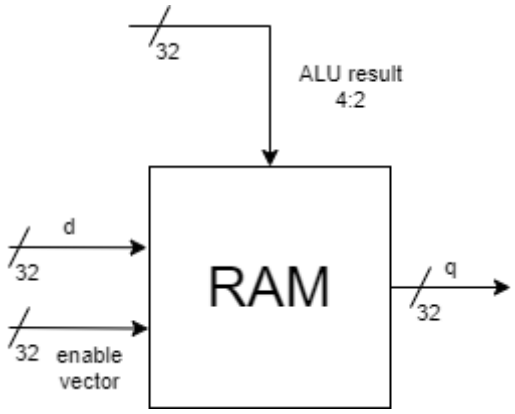


Figura 36. Diagrama RTL de la memoria de datos.

5.4.4 MULTIPLEXOR DE DATOS.

Es el multiplexor de 4 entradas y una salida que selecciona que dato debe escribirse en el registro destino. Se controla mediante la señal de control de selector de escritura con un ancho de 2 bits.

Las entradas del multiplexor de datos son:

- **Control de selección de escritura:** Selecciona el dato de escritura en registro. Los posibles valores de la señal y el dato seleccionado por el multiplexor en cada caso se muestran en la Tabla 18.

Control de selección de escritura	**Dato de escritura en registro destino**
00	PC+4
01	Resultado de la ALU
10	Dato de lectura de memoria
11	PC+imm

Tabla 18. Señal de control de selección de escritura

- **PC+imm.**
- **PC+4.**
- **Dato de lectura de memoria.**
- **Resultado de la ALU:**

La salida del multiplexor de datos es:

- **Dato de escritura en el registro destino.**

El diagrama RTL del multiplexor de datos se muestra en la Figura 37.

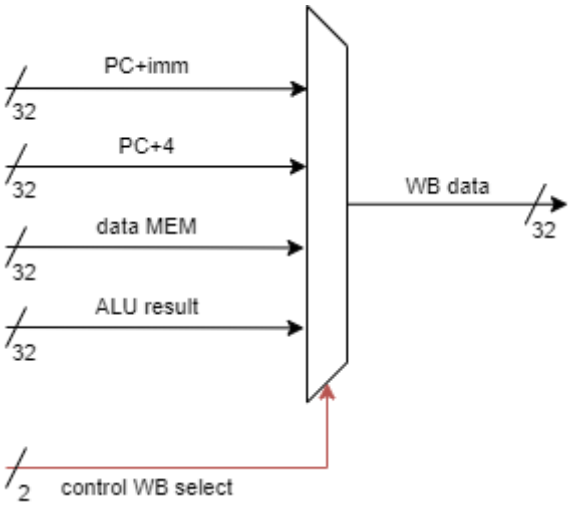


Figura 37. Diagrama RTL del multiplexor de datos.

5.5 WRITE BACK

Se trata de la quinta y última etapa del pipeline. Es la parte del microprocesador encargada de escribir los resultados de las instrucciones en el banco de registros. Es la etapa más sencilla del microprocesador constando únicamente de un multiplexor de dos entradas que decide si debe escribirse algún resultado en registro.

El componente que conforma la etapa es:

- **Multiplexor de dirección:** Selecciona en base a la señal de control de habilitación de escritura si se debe escribir algún dato en registro durante la instrucción.

Las entradas de la etapa son:

- **Control de habilitación de escritura:** Señal de control de 1 bit que determina si la instrucción debe escribir en registro o no.
- **Dirección de registro destino.**

La salida de la etapa es:

- **Dirección de registro destino.**

El diagrama RTL de la etapa Write Back se muestra en la Figura 38.

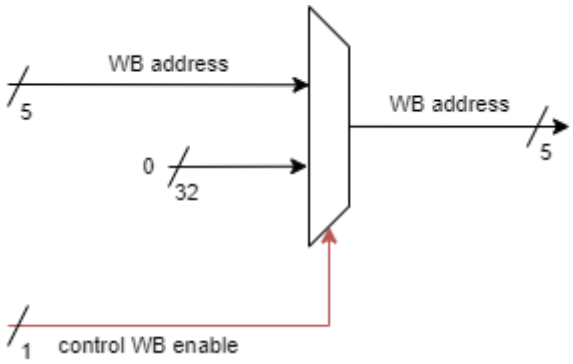


Figura 38. Diagrama RTL de la etapa Write Back

5.5.1 MULTIPLEXOR DE DIRECCIÓN

Es el multiplexor de dos entradas y una salida que se comporta esencialmente como una puerta AND. Se controla con la señal de control de habilitación de escritura que tiene un ancho de 1 bit. Cuando la instrucción tiene escritura en registro la salida es la dirección del

registro destino. Cuando la instrucción no debe escribir en registro la salida es la dirección del registro x0, en el cual no es posible escribir, por lo que así se consigue evitar la escritura.

Las entradas del multiplexor son:

- **Dirección del registro destino.**
- **0.**

La salida del multiplexor es:

- **Dirección del registro destino.**

5.6 MÓDULO DE FORWARDING

Como se ha explicado anteriormente, la segmentación del pipeline genera una serie de riesgos que pueden causar fallos en el funcionamiento de la CPU. Uno de ellos es el data hazard tipo RAW. Se intenta leer el resultado de una instrucción que aún no se ha completado, dando lugar a un dato incorrecto.

Para evitar esto se utiliza un módulo de forwarding el cual detecta el hazard y adelanta el dato desde una etapa posterior a la etapa Execution.

Para detectar esto, compara la dirección de escritura de las instrucciones en Memory, Write Back e Instruction Decode con las direcciones de registro fuente de la etapa Execution.

Si detecta que hay dos iguales y que las instrucciones sin finalizar pretenden escribir en registro, adelanta el dato directamente a Execution antes de que este pase por registro.

El caso de las instrucciones que están en ID es más complicado y requiere hardware adicional. El resultado ya ha sido escrito en el registro, pero debido al sincronismo de los registros, no será accesible hasta el siguiente ciclo. Para poder realizar el forwarding, se

coloca un registro adicional que guarda una copia de la última escritura en el banco de registros. Este registro se explicará en la sección siguiente.

El componente cuenta con las siguientes entradas:

- **Dirección de registro fuente 1.**
- **Dirección de registro fuente 2.**
- **Dirección de registro destino de Memory.**
- **Dirección de registro destino de Write Back.**
- **Dirección de registro destino de Instruction Decode.**
- **Habilitación de escritura en Memory.**
- **Habilitación de escritura en Write Back.**
- **Habilitación de escritura en Instruction Decode.**

Las salidas son las dos señales de control de los multiplexores de EXE**.**

- **Forward 1.**
- **Forward 2.**

Las salidas son idénticas, cada una hace forwarding al dato de un registro fuente. El significado de los posibles valores se muestra en la Tabla 19.

Señal de control de Forward	**Tipo de Forward**
00	Ninguno
01	Dato de Memory
10	Dato de Write Back
11	Dato de Instruction Decode

Tabla 19. Señal de control de forwarding.

El diagrama RTL del módulo de Forwarding se muestra en la Figura 39.

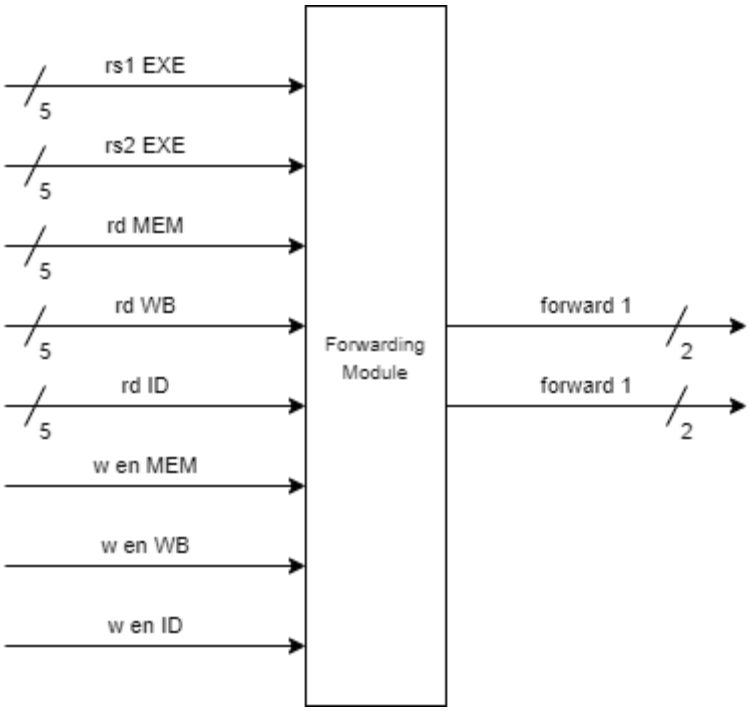


Figura 39. Diagrama RTL del módulo de Forwarding.

5.7 REGISTRO COPIA DE ID

Guarda el valor del ultimo almacenamiento en registro para que sea accesible en caso de necesitar un forwarding. Es necesario ya que no se puede obtener la dirección o el resultado de una instrucción que ya se ha terminado de ejecutar, por lo que hay que dedicarle un registro específico.

El componente en sí no tiene ninguna complejidad o interés, simplemente guarda los siguientes datos y se actualiza cada ciclo.

- **Dirección de registro destino:** La dirección 0 se corresponde con no escritura.
- **Dato de escritura.**

Capítulo 6. ANÁLISIS DE RESULTADOS

Este análisis consiste en comprobar que la CPU diseñada es capaz de ejecutar correctamente todas las instrucciones base de RISC-V. Para realizarlo se ha descompuesto el conjunto de instrucciones en 4 grupos que se han analizado de manera independiente. Todos los análisis se han llevado a cabo creando un código de ensamblador, traduciéndolo a código máquina con las herramientas de software indicadas durante el capítulo 2 y comprobando con ModelSim-Altera que los resultados son los esperados.

6.1 INSTRUCCIONES R E I

Para analizar este conjunto de instrucciones se ha utilizado el código de ensamblador *R_I.asm* que se muestra en el anexo de código fuente. El objetivo de este código es realizar todas las operaciones entre registros y con inmediatos y constatar que se ejecutan correctamente. Para comprobar el correcto funcionamiento se revisan los valores que toman los registros en el tiempo y se comparan con los valores que nos proporciona la simulación de RARS.

Se muestran los resultados obtenidos en la Figura 40 y la Figura 41.

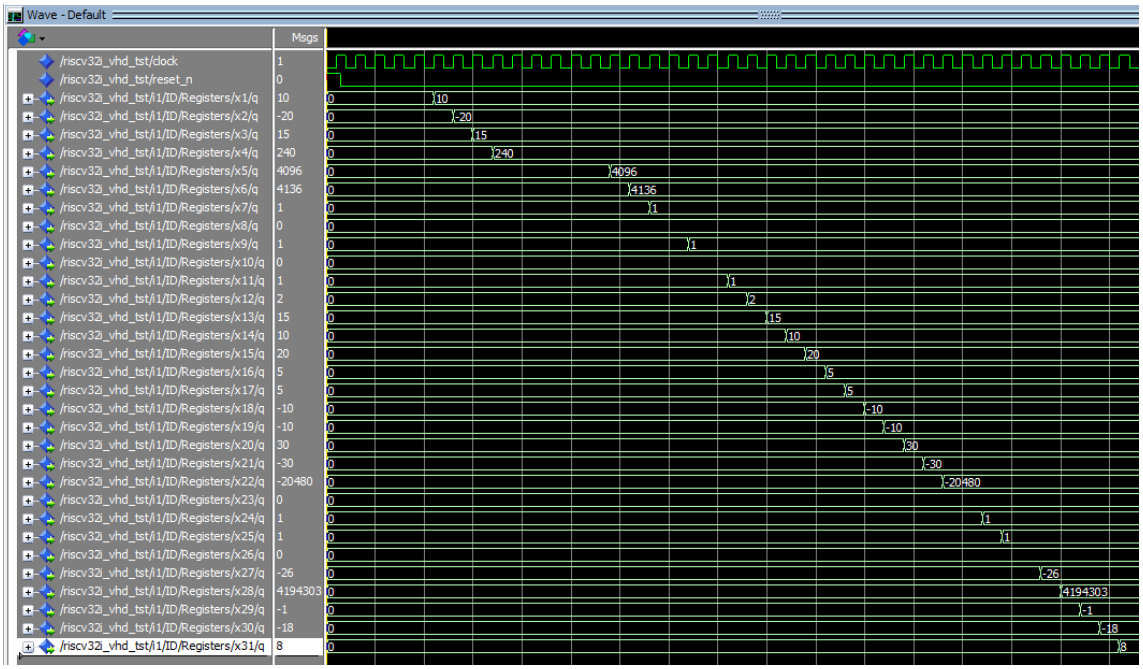


Figura 40. Simulación de instrucciones R e I. ModelsSim-Altera

Name	Number	Value
zero	0	0
ra	1	10
sp	2	-20
gp	3	15
tp	4	240
t0	5	4096
t1	6	4198440
t2	7	1
s0	8	0
s1	9	1
a0	10	0
a1	11	1
a2	12	2
a3	13	15
a4	14	10
a5	15	20
a6	16	5
a7	17	5
s2	18	-10
s3	19	-10
s4	20	30
s5	21	-30
s6	22	-20480
s7	23	0
s8	24	1
s9	25	1
s10	26	0
s11	27	-26
t3	28	4194303
t4	29	-1
t5	30	-18
t6	31	8
pc		4194452

Figura 41. Simulación de instrucciones R e I. RARS.

El dato del registro 6 que se señala en verde es el resultado de la instrucción AUIPC. Esta instrucción suma el valor de un inmediato al valor actual del PC. El software RARS tiene la inicialización del valor del PC en 4.194.344 en vez de en 0 por lo que aquellos valores que dependen del PC para calcularse no se muestran correctamente.

Los demás registros toman los valores correctos tras la ejecución de las instrucciones por lo que podemos concluir que las instrucciones entre registros y con inmediatos funcionan correctamente.

6.2 INSTRUCCIONES B

Este grupo de instrucciones encargadas de los saltos condicionales se ha analizado utilizando el código ensamblador *Branch.asm*. Se han estudiado los valores de los registros a lo largo del tiempo comprobando que el flujo del programa ha sido el correcto. Igual que en el caso anterior el resultado de las instrucciones se ha comprobado comparando la simulación de la CPU con la simulación de ensamblador de RARS.

El resultado de la simulación se muestra en la Figura 42 y la Figura 43.

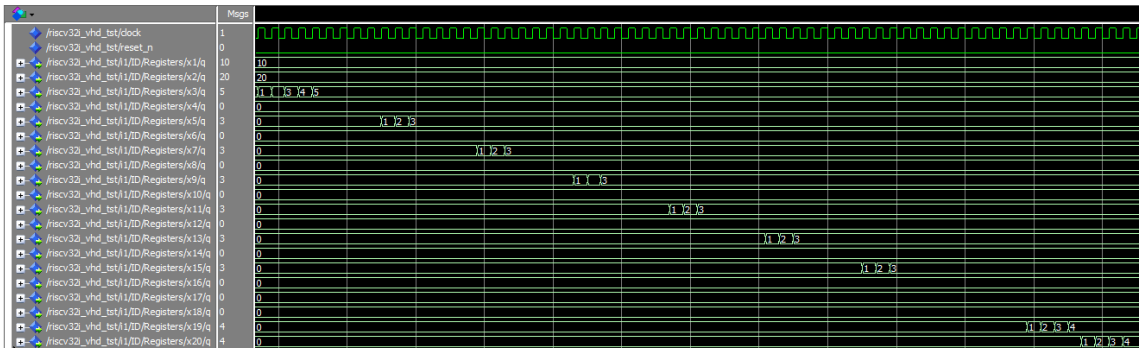


Figura 42. Simulación de instrucciones B. ModelSim-Altera

Name	Number	Value
zero	0	0
ra	1	10
sp	2	20
gp	3	5
tp	4	0
t0	5	3
t1	6	0
t2	7	3
s0	8	0
s1	9	3
a0	10	0
a1	11	3
a2	12	0
a3	13	3
a4	14	0
a5	15	3
a6	16	0
a7	17	0
s2	18	0
s3	19	4
s4	20	4

Figura 43. Simulación de instrucciones B. RARS

6.3 INSTRUCCIONES DE SALTO INCONDICIONAL

Utilizando el código *jal.asm* se ha comprobado si los saltos incondicionales se realizan de manera correcta, tanto los relativos a PC como los relativos a registro. Para ello se han estudiado los valores de los registros a lo largo del tiempo, comprobando que son los correctos y que por tanto los resultados de las operaciones y el flujo de programa se adecuan a lo esperado.

En este caso, como se ha explicado anteriormente en este capítulo, las instrucciones utilizan de manera directa el valor del PC y por tanto la simulación de RARS no proporciona los valores correctos.

En vez de los resultados simulados de RARS se proporciona una explicación sobre el resultado de las instrucciones.

En primer lugar, se inicializa el registro 1 al valor 16. Después se utiliza la instrucción JALR que debe almacenar en el registro x2 el valor del contador de programa de la siguiente instrucción y modificar el valor de PC.

El valor que debe tomar x2 es 28 ya que la instrucción JALR es la número 7 y el valor que debe tomar el PC es la suma del inmediato codificado en la operación y el valor de PC actual. El PC es 24 y el inmediato 16, por lo que el nuevo PC toma valor 40 saltando a la instrucción 11. Las instrucciones 11 y 12 fijan el valor de x3 a 2 y 3 respectivamente.

Por último, la instrucción JAL genera un salto relativo al valor de PC aumentándolo en 8, saltándose la instrucción número 12 y almacena el valor del PC siguiente, que es 44, en el registro 4.

El resultado de la simulación se muestra en la Figura 44.

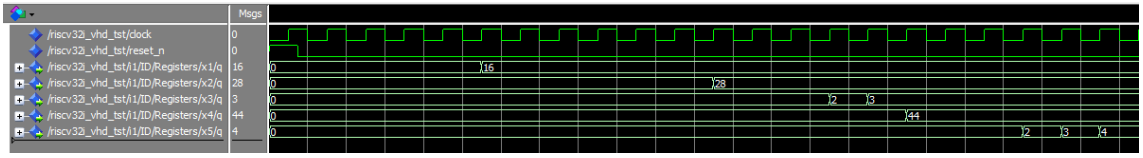


Figura 44. Simulación de instrucciones de salto incondicional. ModelSim--Altera.

6.4 INSTRUCCIONES DE MEMORIA

Para realizar el análisis de este último grupo de instrucciones se ha utilizado el código *loadstore.asm* con el cual se ha comprobado si los accesos a memoria tanto de lectura como de escritura son correctos, en sus variantes con y sin signo.

El diseño de la memoria de datos se ha realizado expresamente para facilitar la comprobación de estos resultados. Para realizar dicha comprobación basta con observar los valores que toman los datos en las direcciones de memoria y el banco de registros a lo largo del tiempo. De manera similar a los casos anteriores se ha utilizado RARS como apoyo para comprobar los datos, en este caso se incluye también el valor de la memoria que proporciona RARS.

El resultado de la simulación se muestra en la Figura 45, la Figura 46 y la Figura 47.

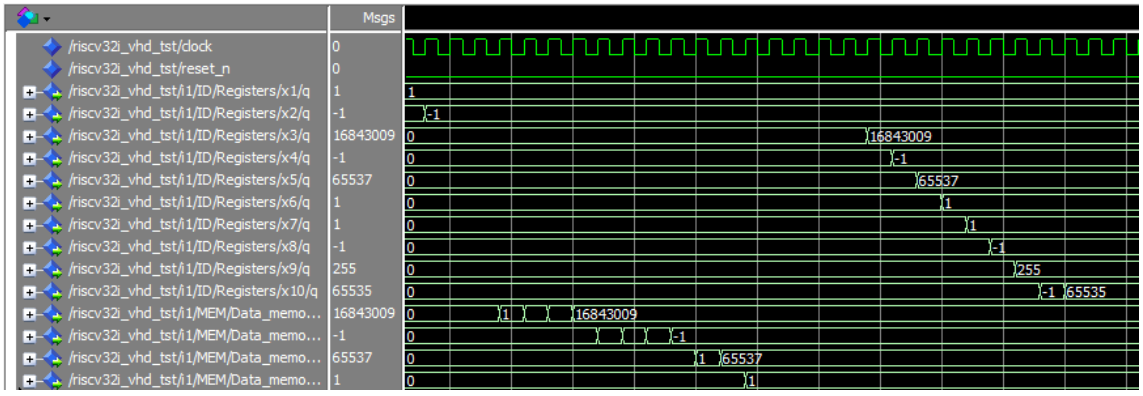


Figura 45. Simulación de instrucciones de memoria. ModelSim-Altera.

Name	Number	Value
zero	0	0
ra	1	1
sp	2	-1
gp	3	16843009
tp	4	-1
t0	5	65537
t1	6	1
t2	7	1
s0	8	-1
s1	9	255
a0	10	65535

Figura 46. Simulación de instrucciones de memoria. RARS. Registros.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x00000000	16843009	-1	65537	1
0x00000020	0	0	0	0

Figura 47. Simulación de instrucciones de memoria. RARS. Memoria.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se va a reflexionar sobre el cumplimiento de los objetivos planteados sobre el proyecto, las conclusiones obtenidas al respecto y posibles desarrollos futuros en línea con los objetivos.

7.1 OBJETIVOS CUMPLIDOS

El primer objetivo planteado en la sección 4.2 era el estudio y comprensión de la ISA RISC-V para lograr este objetivo se ha recurrido a diversos manuales oficiales publicados por la RISC-V International Association que documentan todos los detalles de la microarquitectura. Gracias a esta información se ha podido diseñar la CPU softcore RISC-V de 32 bits.

El segundo objetivo era el diseño de un microprocesador softcore completamente funcional válido para aplicaciones industriales y académicas. El diseño desarrollado a lo largo del proyecto permite ejecutar de manera correcta todas las instrucciones base de RISC-V como se ha mostrado en el capítulo 6 de análisis de resultados. Por tanto, este objetivo se considera alcanzado satisfactoriamente.

El tercer objetivo fijado para el proyecto era la instalación de herramientas de desarrollo para poder compilar y ejecutar código en ensamblador. Para ello se han utilizado las herramientas externas RARS y hex2vhdl además del microprocesador diseñado. El microprocesador ejecuta correctamente las instrucciones en código máquina que proporcionan las herramientas externas, por lo que este objetivo se considera cumplido.

El cuarto y último objetivo era la colaboración con la comunidad de desarrolladores de hardware y software de RISC-V por medio de la publicación del código fuente. Los archivos del proyecto de Quartus con el código VHDL que describe el microprocesador son de acceso público en el repositorio de código GitHub, donde cualquiera puede descargarlos para

instalar el RV32I en su FPGA. Este último objetivo se considera cumplido satisfactoriamente.

7.2 POSIBLES FUTUROS DESARROLLOS

Algunas posibles ramas de desarrollo del proyecto realizado serían:

- La instalación de una herramienta para la compilación de C utilizando la RISC-V GNU Compiler Toolchain. Se trata también de una herramienta de código abierto y permitiría la ejecución de programas en C, lenguaje de uso mucho más común que ensamblador
- El diseño e implantación de módulos que ofrezcan soporte para periféricos, ya sean teclados, ratones, displays LED, monitores, etc. Esto permitiría un uso mucho más cómodo y accesible de la CPU.
- Uso de un bootloader para la programación de la memoria de instrucciones mediante un programador externo diseñado en FPGA, lo que facilitaría la escritura de los programas y permitiría no requerir de un ordenador con Quartus instalado para programar la memoria.
- Ampliar la cantidad de instrucciones y tipos de variable de la ISA añadiendo extensiones de RISC-V que permitan ejecutar códigos complejos con mayor eficiencia.
- Integrar una unidad de predicción de saltos para evitar tener que hacer un flush del pipeline cada vez que se realiza un salto en una instrucción de control.

Como se puede apreciar, el desarrollo de CPU para FPGA es una tecnología con mucho potencial y que puede desarrollarse de diversas maneras. Además de las mencionadas existen otras muchas vertientes de desarrollo posibles.

BIBLIOGRAFÍA

Waterman, A.; Krste, A. (2019). The RISC-V *Instruction Set Manual. Volume I: Unprivileged ISA.* University of California, Berkeley

Krste, A.; Patterson A. (2014) *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report No. UCB/EECS-2014-146.

Armstrong, A.; Bauereiss, T.; Campbell, B., Reid, A., Gray, K. E., Norton-Wright, R., Mundkur, P., et al. (2019). *ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proceedings of the ACM on Programming Languages*. Proceedings of the ACM on programming Languages, ACM Journals.

Mouser Electronics (s.f). *Cyclone II EP2C20F484C7*. Obtenido de https://www.mouser.es/ProductDetail/Intel-Altera/EP2C20F484C7?qs=jblrfmjbeiHvgbQJ2%2FqIdQ%3D%3D. Último acceso el 06/07/2021.

Intel. *M4K memory block*. Obtenido de https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_m4k.htm. Último acceso el 06/07/2021.

Intel. *About the MegaWizard Plug-In Manager*. Obtenido de https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/mega/mega_view_megawiz.htm. Último acceso el 06/07/2021.

Altera. (2008). *Cyclone II Device Handbook, Volume 1*.

Altera (20014). *Internal Memory (RAM and ROM) User Guide*.

Md Ashraful, I.; Hiromu, M,; Kenji, K. (2020) *RVCoreP-32IM: An effective architecture to implement mul/div instructions for five stage RISC-V soft processors*. Cornell Uniuversity. arXiv:2010.16171

RISC-V International Association (s.f). *History of RISC-V*. Obtenido de https://riscv.org/about/history/. Último acceso el 07/07/2021.

Waterman A. (2016) *Design of the RISC-V Instruction Set Architecture*. University of California, Berkeley.

Winans, J. *RISC-V Assembly Language Programming*.

ANEXO I. CÓDIGO FUENTE

1. *RISCV32I.VHD*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity RISCV32I is
 port (
   clock    : in std_logic;
   reset_n : in std_logic
 ) ;
end RISCV32I;

architecture RISCV32I_arc of RISCV32I is

 component Instruction_Fetch
   port (
     clock         : in  std_logic;
     reset_n       : in  std_logic;
     stall         : in  std_logic;--si hay que stallear el fetcheo
     instruction   : out std_logic_vector (31 downto 0);--la
instruction fetcheada
     pc_4_out      : out std_logic_vector (31 downto 0);--pc de
siguiente instruccion
     EXE_pc_imm    : in  std_logic_vector (31 downto 0);--pc+imm desde
exe
     pc_out        : out std_logic_vector (31 downto 0)--pc de la
instruction fetcheada
   ) ;
 end component;

 component Instruction_Decode
   port (
     clock                   : in  std_logic;
     reset_n                 : in  std_logic;
     wb_data_in              : in  std_logic_vector (31 downto 0);
     wb_address_in           : in  std_logic_vector (4  downto 0);
     -- es x0 si no hay enable
     instruction             : in  std_logic_vector (31 downto 0);
     data_rs1                : out std_logic_vector (31 downto 0);
     data_rs2                : out std_logic_vector (31 downto 0);
     address_rs1             : out std_logic_vector (4  downto 0);
     address_rs2             : out std_logic_vector (4  downto 0);
     imm                     : out std_logic_vector (31 downto 0);
```

```
    control_EXE_data_mux    : out std_logic;
    -- multiplexa rs1 rs2 imm etc
    control_EXE_LUI         : out std_logic;
    -- dice si es una LUI o no
    control_EXE_ALU         : out std_logic_vector (3 downto 0);
     -- elige operacion de la ALU
    control_MEM             : out std_logic_vector (3 downto 0);
    -- marca la operacion de MEM y si tiene signo.
    control_WB_sel          : out std_logic_vector (1 downto 0);
    -- multiplexa la salida de wb entre alu mem pc
    control_WB_wren         : out std_logic;
    -- dice si hay que escribir en rd en la instruction, escribir en
pc lo hara el jump control
    control_WB_address_out : out std_logic_vector (4 downto 0); -- rd
address
    control_EXE_branch      : out std_logic_vector (1 downto 0)
    -- dice si existe posibilidad de branch y de escribir en pc 00 no
salto 01 cond 10 uncond
  ) ;
end component;

component Execution
  port (
    pc                : in  std_logic_vector (31 downto 0);
    imm_in            : in  std_logic_vector (31 downto 0);
    control_data_mux  : in  std_logic;
    rs1_data_in       : in  std_logic_vector (31 downto 0);
    rs2_data_in       : in  std_logic_vector (31 downto 0);
    control_LUI       : in  std_logic;
    control_jump      : in  std_logic_vector (1 downto 0);
    control_ALU       : in  std_logic_vector (3  downto 0);
    ALU_result        : out std_logic_vector (31 downto 0);
    pc_imm            : out std_logic_vector (31 downto 0);
    rs2_out_store     : out std_logic_vector (31 downto 0);
    jump_flush        : out std_logic;
    forward_sel_1     : in  std_logic_vector (1 downto 0);
    forward_sel_2     : in  std_logic_vector (1 downto 0);
    MEM_data          : in  std_logic_vector (31 downto 0);
    WB_data           : in  std_logic_vector (31 downto 0);
    ID_data           : in  std_logic_vector (31 downto 0)
  ) ;
end component;

component Memory
  port (
    reset_n          : in  std_logic;
    clock            : in  std_logic;
    ALU_result_in    : in  std_logic_vector (31 downto 0);
    --resultado de la ALU (es la address en stores y loads)
    data_in          : in  std_logic_vector (31 downto 0);
    --dato a cargar en instrucciones store (rs2)
    control_MEM_in   : in  std_logic_vector (3  downto 0);
```

```
    --control word de la etapa MEM. MSB-> signed 1, unsigned 0. 3LSB
(LB,LH...)
    memory_out         : out std_logic_vector (31 downto 0);
    --dato que sale de la memoria
    ALU_result_out   : out std_logic_vector (31 downto 0);
    --res de alu por si se necesita en wb, no se modifica
    load_address     : in  std_logic_vector (31 downto 0);
    load_control_in : in  std_logic_vector (3  downto 0);
    load_control_out: out std_logic_vector (3  downto 0);
    control_WB_sel   : in  std_logic_vector (1  downto 0);
    PC_imm           : in  std_logic_vector (31 downto 0);
    PC_4_in          : in  std_logic_vector (31 downto 0)
  ) ;
end component;

component Write_Back
  port (
    control_WB_sel   : in  std_logic_vector (1 downto 0);
    --multiplexa el dato de salida entre (nada alu mem pc+imm)
    control_WB_wren : in  std_logic; -- marca si hay que escribir en
rd
    wb_address_in    : in  std_logic_vector (4 downto 0);
    --direccion del registro a escribir. 5 bits para 32 registros.
    wb_address_out   : out std_logic_vector (4 downto 0);
     --salida multiplexada de wb_address_in y x0.
    wb_data_in       : in  std_logic_vector (31 downto 0);
    wb_data_out      : out std_logic_vector (31 downto 0)
    --salida del dato seleccionado entre alu mem pc y X
    ) ;
end component;

component REG1_IF_ID
  port (
    clock            : in  std_logic;
    clear            : in  std_logic;
    pc_in            : in  std_logic_vector (31 downto 0);
    --instruction_in   : in  std_logic_vector (31 downto 0);
    pc_out           : out std_logic_vector (31 downto 0)
    --instruction_out : out std_logic_vector (31 downto 0)
    ) ;
end component;

component REG2_ID_EXE
  port (
    clock            : in  std_logic;
    clear            : in  std_logic;
    pc_in            : in  std_logic_vector (31 downto 0);
    pc_out           : out std_logic_vector (31 downto 0);
    control_branch_in : in std_logic_vector (1 downto 0);
    control_branch_out : out std_logic_vector (1 downto 0);
    imm_in : in std_logic_vector (31 downto 0);
    imm_out : out std_logic_vector (31 downto 0);
```

```
    pc_4_in : in std_logic_vector (31 downto 0);
    pc_4_out : out std_logic_vector (31 downto 0);
    control_mux_in : in std_logic;
    control_mux_out : out std_logic;
    control_LUI_in : in std_logic;
    control_LUI_out : out std_logic;
    control_ALU_in : in std_logic_vector (3 downto 0);
    control_ALU_out : out std_logic_vector (3 downto 0);
    control_MEM_in : in std_logic_vector (3 downto 0);
    control_MEM_out : out std_logic_vector (3 downto 0);
    control_wb_sel_in : in std_logic_vector (1 downto 0);
    control_wb_sel_out : out std_logic_vector (1 downto 0);
    control_wb_en_in : in std_logic;
    control_wb_en_out : out std_logic;
    control_wb_address_in : in std_logic_vector (4 downto 0);
    control_wb_address_out : out std_logic_vector (4 downto 0);
    data1_in : in std_logic_vector (31 downto 0);
    data1_out : out std_logic_vector (31 downto 0);
    data2_in : in std_logic_vector (31 downto 0);
    data2_out : out std_logic_vector (31 downto 0);
    rs1_address_in  : in  std_logic_vector (4 downto 0);
    rs2_address_in  : in  std_logic_vector (4 downto 0);
    rs1_address_out : out std_logic_vector (4 downto 0);
    rs2_address_out : out std_logic_vector (4 downto 0)
    ) ;
end component;

component REG3_EXE_MEM
  port (
    clock           : in  std_logic;
    clear           : in  std_logic;
    pc_imm_in           : in  std_logic_vector (31 downto 0);
    pc_imm_out          : out std_logic_vector (31 downto 0);
    ALU_res_in : in std_logic_vector (31 downto 0);
    ALU_res_out : out std_logic_vector (31 downto 0);
    pc_4_in : in std_logic_vector (31 downto 0);
    pc_4_out : out std_logic_vector (31 downto 0);
    rs2_in : in std_logic_vector (31 downto 0);
    rs2_out : out std_logic_vector (31 downto 0);
    control_MEM_in : in std_logic_vector (3 downto 0);
    control_MEM_out : out std_logic_vector (3 downto 0);
    control_wb_sel_in : in std_logic_vector (1 downto 0);
    control_wb_sel_out : out std_logic_vector (1 downto 0);
    control_wb_en_in : in std_logic;
    control_wb_en_out : out std_logic;
    control_wb_address_in : in std_logic_vector (4 downto 0);
    control_wb_address_out : out std_logic_vector (4 downto 0)
    ) ;
end component;

component REG4_MEM_WB
  port (
```

```
    clock           : in  std_logic;
    clear           : in  std_logic;
    pc_imm_in          : in  std_logic_vector (31 downto 0);
    pc_imm_out         : out std_logic_vector (31 downto 0);
    ALU_res_in : in std_logic_vector (31 downto 0);
    ALU_res_out : out std_logic_vector (31 downto 0);
    pc_4_in : in std_logic_vector (31 downto 0);
    pc_4_out : out std_logic_vector (31 downto 0);
    control_wb_sel_in : in std_logic_vector (1 downto 0);
    control_wb_sel_out : out std_logic_vector (1 downto 0);
    control_wb_en_in : in std_logic;
    control_wb_en_out : out std_logic;
    control_wb_address_in : in std_logic_vector (4 downto 0);
    control_wb_address_out : out std_logic_vector (4 downto 0);
    MEM_data_in : in std_logic_vector (31 downto 0);
    MEM_data_out : out std_logic_vector (31 downto 0);
    load_control_in: in std_logic_vector (3 downto 0);
    load_control_out: out std_logic_vector(3 downto 0)
    ) ;
end component;

component Forward_EXE
  port (
     w_e_MEM      : in  std_logic;
     w_e_WB       : in  std_logic;
     address1_EXE : in  std_logic_vector (4 downto 0);
     address2_EXE : in  std_logic_vector (4 downto 0);
     address_MEM  : in  std_logic_vector (4 downto 0);
     address_WB   : in  std_logic_vector (4 downto 0);
     address_ID   : in  std_logic_vector (4 downto 0);
     forward1     : out std_logic_vector (1 downto 0);
     forward2     : out std_logic_vector (1 downto 0)
  ) ;
end component;

component ID_Register_Copy
  port (
    clock          : in  std_logic;
    reset_n        : in  std_logic;
    id_data_in     : in  std_logic_vector (31 downto 0);
    id_data_out    : out std_logic_vector (31 downto 0);
    id_address_in  : in  std_logic_vector ( 4 downto 0);
    id_address_out : out std_logic_vector ( 4 downto 0)
    );
end component;

--IF in
--signal pc_imm_EXE_out : std_logic_vector (31 downto 0);
--signal jump_flush_stall : std_logic;

--IF out
signal instruction_IF_out : std_logic_vector (31 downto 0);
```

```
signal pc_4_IF_out : std_logic_vector (31 downto 0);
signal pc_IF_out : std_logic_vector (31 downto 0);

--ID in
signal wb_data_ID : std_logic_vector (31 downto 0);
signal wb_address_ID : std_logic_vector (4 downto 0);

signal instruction_ID_in : std_logic_vector (31 downto 0);

--ID out
signal rs1_ID_out : std_logic_vector (31 downto 0);
signal rs2_ID_out : std_logic_vector (31 downto 0);
signal imm_ID_out : std_logic_vector (31 downto 0);

  --control EXE
  signal control_data_mux_ID_out : std_logic;
  signal control_LUI_ID_out : std_logic;
  signal control_ALU_ID_out : std_logic_vector (3 downto 0);
  signal control_branch_ID_out : std_logic_vector (1 downto 0);

  --control MEM
  signal control_MEM_ID_out : std_logic_vector (3 downto 0);

  --control WB
  signal control_WB_wren_ID_out : std_logic;
  signal control_WB_address_ID_out : std_logic_vector (4 downto 0);
  signal control_WB_sel_ID_out : std_logic_vector(1 downto 0);

--ID Bypass
signal pc_ID_bp : std_logic_vector (31 downto 0);

--EXE in
signal rs1_EXE_in : std_logic_vector (31 downto 0);
signal rs2_EXE_in : std_logic_vector (31 downto 0);
signal pc_EXE_in : std_logic_vector(31 downto 0);
signal imm_EXE_in : std_logic_vector (31 downto 0);
signal control_branch_EXE_in : std_logic_vector (1 downto 0);
signal control_data_mux_EXE_in : std_logic;
signal control_LUI_EXE_in : std_logic;
signal control_ALU_EXE_in : std_logic_vector (3 downto 0);

--EXE out
signal jump_flush_stall : std_logic;

signal pc_imm_EXE_out : std_logic_vector (31 downto 0);
signal ALU_result_EXE_out : std_logic_vector(31 downto 0);
signal rs2_EXE_out : std_logic_vector (31 downto 0);

--EXE Bypass
signal pc_4_EXE_bp : std_logic_vector (31 downto 0);
signal control_MEM_EXE_bp : std_logic_vector (3 downto 0);
signal control_WB_sel_EXE_bp : std_logic_vector (1 downto 0);
```

```
signal control_WB_wren_EXE_bp   : std_logic;
signal control_WB_address_EXE_bp : std_logic_vector(4 downto 0);

--MEM in
signal ALU_result_MEM_in : std_logic_vector (31 downto 0);
signal rs2_MEM_in : std_logic_vector (31 downto 0);
signal control_MEM_MEM_in : std_logic_vector (3 downto 0);

--MEM out
signal MEM_data_MEM_out : std_logic_vector (31 downto 0);
signal ALU_result_MEM_out : std_logic_vector (31 downto 0);

--MEM Bypass
signal pc_4_MEM_bp : std_logic_vector (31 downto 0);
signal pc_imm_MEM_bp : std_logic_vector (31 downto 0);
signal control_WB_sel_MEM_bp : std_logic_vector (1 downto 0);
signal control_WB_wren_MEM_bp : std_logic;
signal control_WB_address_MEM_bp : std_logic_vector(4 downto 0);

--WB in
signal ALU_result_WB_in : std_logic_vector (31 downto 0);
signal MEM_data_WB_in : std_logic_vector (31 downto 0);
signal control_WB_sel_WB_in : std_logic_vector (1 downto 0);
signal control_WB_Wren_WB_in : std_logic;
signal control_WB_address_WB_in : std_logic_vector (4 downto 0);
signal pc_4_WB_in : std_logic_vector (31 downto 0);
signal pc_imm_WB_in : std_logic_vector (31 downto 0);

--WB out
--signal wb_data_ID : std_logic_vector (31 downto 0);
--signal wb_address_ID : std_logic_vector (4 downto 0);
signal flush_aux : std_logic;

signal load_control_MEM_out : std_logic_vector (3 downto 0);
signal load_control_WB_out : std_logic_vector (3 downto 0);
signal load_address_MEM_out : std_logic_vector (31 downto 0);
signal load_address_WB_out : std_logic_vector (31 downto 0);

signal address_rs1_ID_out : std_logic_vector (4 downto 0);
signal address_rs2_ID_out : std_logic_vector (4 downto 0);
signal address_rs1_EXE_in : std_logic_vector (4 downto 0);
signal address_rs2_EXE_in : std_logic_vector (4 downto 0);
signal id_data_EXE_in :std_logic_vector (31 downto 0);
signal id_address_fwd_in : std_logic_vector (4 downto 0);

signal forward1_out : std_logic_vector (1 downto 0);
signal forward2_out : std_logic_vector (1 downto 0);

signal control_address_ID : std_logic_vector (4 downto 0);
signal control_WB_address_ID : std_logic_vector (4 downto 0);
```

```
begin

 flush_aux <= jump_flush_stall or reset_n;


 IF_i : Instruction_Fetch --IF es reservado por el if logico
   port map(
     clock         => clock,
     reset_n       => reset_n,
     stall         => jump_flush_stall,
     instruction   => instruction_IF_out,
     pc_4_out      => pc_4_IF_out,
     EXE_pc_imm    => pc_imm_EXE_out,
     pc_out        => pc_IF_out
     );

 REG1 :REG1_IF_ID
   port map(
     clock           => clock,
     clear           => reset_n, --flush_aux
     pc_in           => pc_IF_out,
     --instruction_in  => instruction_IF_out,
     pc_out          => pc_ID_bp
     --instruction_out => instruction_ID_in
     ) ;

 ID : Instruction_Decode
   port map (
       clock                 => clock,
       reset_n               => reset_n,
       wb_data_in            => wb_data_ID,
       wb_address_in         => wb_address_ID,
       instruction           => instruction_IF_out,
       data_rs1              => rs1_ID_out,
       data_rs2              => rs2_ID_out,
       address_rs1           => address_rs1_ID_out,
       address_rs2           => address_rs2_ID_out,
       imm                   => imm_ID_out,
       control_EXE_data_mux  => control_data_mux_ID_out,
       control_EXE_LUI       => control_LUI_ID_out,
       control_EXE_ALU       => control_ALU_ID_out,
       control_MEM           => control_MEM_ID_out,
       control_WB_sel        => control_WB_sel_ID_out,
       control_WB_wren       => control_WB_wren_ID_out,
       control_WB_address_out => control_WB_address_ID_out,
       control_EXE_branch    => control_branch_ID_out
     ) ;

 REG2 : REG2_ID_EXE
   port map(
     clock           => clock,
     clear           => reset_n,
```

```
    pc_in           => pc_ID_bp,
    pc_out          => pc_EXE_in,
    control_branch_in => control_branch_ID_out,
    control_branch_out => control_branch_EXE_in,
    imm_in => imm_ID_out,
    imm_out => imm_EXE_in,
    pc_4_in => pc_4_IF_out,
    pc_4_out => pc_4_EXE_bp,
    control_mux_in => control_data_mux_ID_out,
    control_mux_out =>control_data_mux_EXE_in,
    control_LUI_in => control_LUI_ID_out,
    control_LUI_out => control_LUI_EXE_in,
    control_ALU_in => control_ALU_ID_out,
    control_ALU_out => control_ALU_EXE_in,
    control_MEM_in => control_MEM_ID_out,
    control_MEM_out => control_MEM_EXE_bp,
    control_wb_sel_in => control_WB_sel_ID_out,
    control_wb_sel_out => control_WB_sel_EXE_bp,
    control_wb_en_in => control_WB_wren_ID_out,
    control_wb_en_out => control_WB_wren_EXE_bp,
    control_wb_address_in => control_WB_address_ID_out,
    control_wb_address_out => control_WB_address_EXE_bp,
    data1_in => rs1_ID_out,
    data1_out => rs1_EXE_in,
    data2_in => rs2_ID_out,
    data2_out => rs2_EXE_in,
    rs1_address_in   => address_rs1_ID_out,
    rs2_address_in   => address_rs2_ID_out,
    rs1_address_out  => address_rs1_EXE_in,
    rs2_address_out  => address_rs2_EXE_in
    );

EXE : Execution
  port map(
    pc               => pc_EXE_in,
    imm_in           => imm_EXE_in,
    control_data_mux => control_data_mux_EXE_in,
    rs1_data_in      => rs1_EXE_in,
    rs2_data_in      => rs2_EXE_in,
    control_LUI      => control_LUI_EXE_in,
    control_jump     => control_branch_EXE_in,
    control_ALU      => control_ALU_EXE_in,
    ALU_result       => ALU_result_EXE_out,
    pc_imm           => pc_imm_EXE_out,
    rs2_out_store    => rs2_EXE_out,
    jump_flush       => jump_flush_stall,
    forward_sel_1    => forward1_out,
    forward_sel_2    => forward2_out,
    MEM_data         => MEM_data_MEM_out,
    WB_data          => wb_data_ID,
    ID_data          => id_data_EXE_in
  ) ;
```

```
REG3 : REG3_EXE_MEM
  port map(
    clock           => clock,
    clear           => reset_n,
    pc_imm_in       => pc_imm_EXE_out,
    pc_imm_out      => pc_imm_MEM_bp,
    ALU_res_in    => ALU_result_EXE_out,
    ALU_res_out     => ALU_result_MEM_in,
    pc_4_in => pc_4_EXE_bp,
    pc_4_out => pc_4_MEM_bp,
    rs2_in => rs2_EXE_out,
    rs2_out => rs2_MEM_in,
    control_MEM_in => control_MEM_EXE_bp,
    control_MEM_out => control_MEM_MEM_in,
    control_wb_sel_in => control_WB_sel_EXE_bp,
    control_wb_sel_out => control_WB_sel_MEM_bp,
    control_wb_en_in => control_WB_wren_EXE_bp,
    control_wb_en_out => control_WB_wren_MEM_bp,
    control_wb_address_in => control_WB_address_EXE_bp,
    control_wb_address_out => control_WB_address_MEM_bp
    ) ;

MEM : Memory
  port map (
    reset_n         => reset_n,
    clock           => clock,
    ALU_result_in   => ALU_result_MEM_in,
    data_in         => rs2_MEM_in,
    control_MEM_in  => control_MEM_MEM_in,
    memory_out      => MEM_data_MEM_out,
    ALU_result_out  => ALU_result_MEM_out,
    load_address    => ALU_result_MEM_in,
    load_control_in => control_MEM_MEM_in,
    load_control_out=> load_control_MEM_out,
    control_WB_sel  => control_WB_sel_MEM_bp,
    pc_imm          => pc_imm_MEM_bp,
    pc_4_in         => pc_4_MEM_bp
    ) ;

REG4 :REG4_MEM_WB
  port map(
    clock           => clock,
    clear           => reset_n,
    pc_imm_in           => pc_imm_MEM_bp,
    pc_imm_out          => pc_imm_WB_in,
    ALU_res_in => ALU_result_MEM_out,
    ALU_res_out => ALU_result_WB_in,
    pc_4_in => pc_4_MEM_bp,
    pc_4_out => pc_4_WB_in,
    control_wb_sel_in => control_WB_sel_MEM_bp,
    control_wb_sel_out => control_WB_sel_WB_in,
```

```
    control_wb_en_in => control_WB_Wren_MEM_bp,
    control_wb_en_out => control_WB_Wren_WB_in,
    control_wb_address_in => control_WB_address_MEM_bp,
    control_wb_address_out => control_WB_address_WB_in,
    MEM_data_in => MEM_data_MEM_out,
    MEM_data_out => MEM_data_WB_in,
    load_control_in => load_control_MEM_out,
    load_control_out => load_control_WB_out
    ) ;

WB : Write_Back
  port map(
    control_WB_sel  => control_WB_sel_WB_in,
    control_WB_wren => control_WB_Wren_WB_in,
    wb_address_in   => control_WB_address_WB_in,
    wb_data_in      => MEM_data_WB_in,
    wb_data_out     => wb_data_ID,
    wb_address_out  => wb_address_ID
  ) ;

Forward_EXE_i : Forward_EXE
  port map(
    w_e_MEM      => control_WB_Wren_MEM_bp,
    w_e_WB       => control_WB_Wren_WB_in,
    address1_EXE => address_rs1_EXE_in,
    address2_EXE => address_rs2_EXE_in,
    address_MEM  => control_WB_address_MEM_bp,
    address_WB   => WB_address_ID,
    address_ID   => id_address_fwd_in,
    forward1   => forward1_out,
    forward2     => forward2_out
    );

Register_copy : ID_Register_Copy
  port map(
    clock          => clock,
    reset_n        => reset_n,
    id_data_in     => wb_data_ID,
    id_data_out    => id_data_EXE_in,
    id_address_in  => wb_address_ID,
    id_address_out => id_address_fwd_in
    );


end  RISCV32I_arc;
```

1.1 INSTRUCTION_FETCH.VHD

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity Instruction_Fetch is
 port (
   clock         : in  std_logic;
   reset_n       : in  std_logic;
   stall         : in  std_logic;--si hay que stallear el fetcheo,
tambien es el jump y el flush
   instruction   : out std_logic_vector (31 downto 0);--la instruction
fetcheada
   pc_4_out      : out std_logic_vector (31 downto 0);--pc de la
instrucion siguiente que se salta el reg IF ID
   EXE_pc_imm    : in  std_logic_vector (31 downto 0);--pc+imm desde
exe
   pc_out        : out std_logic_vector (31 downto 0)--pc de la
instruction fetcheada
 ) ;
end Instruction_Fetch;

architecture Instruction_Fetch_arc of Instruction_Fetch is

   signal enable : std_logic; --es el opuesto al stall. NOT stall
   signal pc_aux : std_logic_vector (31 downto 0); --el pc dentro del
IF
   signal pc_4_aux : std_logic_vector (31 downto 0);
   signal next_pc_aux : std_logic_vector (31 downto 0);
   signal instruction_aux : std_logic_vector(31 downto 0);

   component ROM
     port(
          clk: in std_logic;                          -- Synchronous ROM
          en_pc: in std_logic;                        -- Whith enable
          addr: in  std_logic_vector(11 downto 0);    -- Address bus
          data: out std_logic_vector(31 downto 0) ); -- Data out
   end component;

   component IF_PC
     port (
       clock  : in  std_logic;
       enable : in  std_logic;
       clear  : in  std_logic;
       stall  : in  std_logic;
       d      : in  std_logic_vector (31 downto 0);
       q      : out std_logic_vector (31 downto 0)
       ) ;
   end component;

   component IF_PC_adder
     port (
       pc    : in std_logic_vector (31 downto 0);
```

```
        pc_4 : out std_logic_vector (31 downto 0)
        ) ;
    end component;

    component IF_Jump_Mux2x1
      port (
        control_jump : in  std_logic; --1 jump, 0 not jump
        pc_4         : in  std_logic_vector (31 downto 0);
        pc_imm       : in  std_logic_vector (31 downto 0);
        next_pc      : out std_logic_vector (31 downto 0)
        ) ;
    end component;

    component IF_Instruction_Mux2x1
      port (
        control_jump    : in  std_logic; --1 jump, 0 not jump
        instruction_in  : in  std_logic_vector (31 downto 0);
        instruction_out : out std_logic_vector (31 downto 0)
        ) ;
    end component;

begin

 enable <= NOT (stall OR reset_n);

 Instruction_fetch_mem : ROM
   port map(
          addr  => pc_aux (11 downto 0), -- nuestra memoria solo tiene
capacidad para 1024 palabras y como no accedemos a bytes en IF los dos
primeros bits son 0 para alinear
          en_pc   => enable, --el clock enable permite ignorar o no el
reloj. en un stall queremos ignorarlo porque queremos que la memoria
stallee
          clk     => clock,
          data  => instruction_aux --es la instruccion que se ha
fetcheado
        );

 Adder : IF_PC_adder
   port map(
        pc => pc_aux,
        pc_4 => pc_4_aux

     );

 Jump_Mux : IF_Jump_Mux2x1 --decide que escribir en pc
   port map(
        control_jump => stall,
        pc_4 => pc_4_aux,
        pc_imm => EXE_pc_imm,
        next_pc => next_pc_aux
     );
```

```
    PC : IF_PC
      port map(
        clock => clock,
        enable => '1',
        clear => reset_n,
        stall => '0',
        d => next_pc_aux,
        q => pc_aux
        );

    Instruction_Mux : IF_Instruction_Mux2x1
      port map (
        control_jump    => stall,
        instruction_in  => instruction_aux,
        instruction_out => instruction
          );

      pc_out <= pc_aux; --usamos pc para sacar el address de memoria y lo
   pasamos a la siguiente stage de la pipeline sin modificar.
      pc_4_out <= pc_aux; --pc_4 es pc pero bypassea reg1

   end  Instruction_Fetch_arc;
```

1.1.1. IF_Instruction_Mux2x1.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity IF_Instruction_Mux2x1 is
    port (
      control_jump    : in  std_logic; --1 jump, 0 not jump
      instruction_in  : in  std_logic_vector (31 downto 0);
      instruction_out : out std_logic_vector (31 downto 0)
      ) ;
end IF_Instruction_Mux2x1;

architecture IF_Instruction_Mux2x1_arc of IF_Instruction_Mux2x1 is

begin

  instruction_out <=
    --not jump
    instruction_in when (control_jump = '0') else
    --jump
    x"00000033";
```

```
end  IF_Instruction_Mux2x1_arc;
```

1.1.2 IF_Jump_Mux2x1.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity IF_Jump_Mux2x1 is
    port (
      control_jump : in  std_logic; --1 jump, 0 not jump
      pc_4         : in  std_logic_vector (31 downto 0);
      pc_imm       : in  std_logic_vector (31 downto 0);
      next_pc      : out std_logic_vector (31 downto 0)
      ) ;
end IF_Jump_Mux2x1;

architecture IF_Jump_Mux2x1_arc of IF_Jump_Mux2x1 is

begin

  next_pc <=
    --not jump
    pc_4 when (control_jump = '0') else
    --jump
    pc_imm;

end  IF_Jump_Mux2x1_arc;
```

1.1.3 IF_PC

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity IF_PC is
  port (
    clock  : in  std_logic;
    enable : in  std_logic;
    clear  : in  std_logic;
    stall  : in  std_logic;
    d      : in  std_logic_vector (31 downto 0);
    q      : out std_logic_vector (31 downto 0)
    ) ;
```

```
end IF_PC;

architecture IF_PC_arc of IF_PC is

  begin
    process(clock, clear, stall)
        begin
            if clear = '1' then
                q <= "00000000000000000000000000000000";

            elsif stall = '0' then
              if rising_edge(clock) then
                  if enable = '1' then
                      q <= d;
                  end if;
              end if;
          end if;
      end process;

end  IF_PC_arc;
```

1.1.4 IF_PC_adder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity IF_PC_adder is
  port (
    pc   : in std_logic_vector (31 downto 0);
    pc_4 : out std_logic_vector (31 downto 0)
  ) ;
end IF_PC_adder;

architecture IF_PC_adder_arc of IF_PC_adder is

  signal pc_aux : std_logic_vector (32 downto 0);

begin

  pc_aux <= std_logic_vector('0' & unsigned(pc) + to_unsigned(4, 33));
  pc_4 <= pc_aux (31 downto 0);

end  IF_PC_adder_arc;
```

1.1.5 ROM.vhd

```
-- ROM file for the ICAI-RISC-V processor.
-- Generated from the hex file:  ROM.vhd

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

entity ROM is
  port(
    clk: in std_logic;                              -- Synchronous ROM
    en_pc: in std_logic;                            -- Whith enable
    addr: in  std_logic_vector(11 downto 0);    -- Address bus
    data: out std_logic_vector(31 downto 0) ); -- Data out
end ROM;

architecture Behavioural of ROM is
  -- The internal address is word address, no byte address
  signal internal_addr : std_logic_vector(6 downto 0);

  -- ROM declaration
 type mem_t is array (0 to 127 ) of std_logic_vector(31 downto 0);
  signal memory : mem_t:= (
     0  => X"08000093",
     1  => X"fff00113",
     2  => X"00100023",
     3  => X"001000a3",
     4  => X"00100123",
     5  => X"001001a3",
     6  => X"00200223",
     7  => X"002002a3",
     8  => X"00200323",
     9  => X"002003a3",
     10  => X"00101423",
     11  => X"00101523",
     12  => X"00102623",
     13  => X"00002083",
     14  => X"00402203",
     15  => X"00802283",
     16  => X"00c02303",
     17  => X"00000383",
     18  => X"00400403",
     19  => X"00404483",
     20  => X"00401503",
     21  => X"00405503",
     22  => X"002085b3",
     23  => X"40208633",
     24  => X"0020f6b3",
     others => X"00000000");
begin
```

```
  internal_addr <= addr(8 downto 2);

  mem_rom: process(clk)
  begin
    if clk'event and clk = '1' then
      if en_pc = '1' then
        data <= memory(to_integer(unsigned(internal_addr))); else
        data <= x"00000000";
      end if;
    end if;
  end process mem_rom;
end architecture Behavioural;
2
```

2.1 INSTRUCTION DECODE

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity Instruction_Decode is
  port (
    clock                  : in  std_logic;
    reset_n                : in  std_logic;
    wb_data_in             : in  std_logic_vector (31 downto 0);
    wb_address_in          : in  std_logic_vector (4  downto 0); -- es x0
si no hay enable
    instruction            : in  std_logic_vector (31 downto 0);
    data_rs1               : out std_logic_vector (31 downto 0);
    data_rs2               : out std_logic_vector (31 downto 0);
    address_rs1            : out std_logic_vector (4  downto 0);
    address_rs2            : out std_logic_vector (4  downto 0);
    imm                    : out std_logic_vector (31 downto 0);
    control_EXE_data_mux   : out std_logic;-- multiplexa rs1 rs2 imm etc
    control_EXE_LUI        : out std_logic;-- dice si es una LUI o no
    control_EXE_ALU        : out std_logic_vector (3 downto 0);-- elige
operacion de la ALU
    control_MEM            : out std_logic_vector (3 downto 0);-- marca
la operacion de MEM y si tiene signo.
    control_WB_sel         : out std_logic_vector (1 downto 0);--
multiplexa la salida de wb entre alu mem pc
    control_WB_wren        : out std_logic; -- dice si hay que escribir
en rd en la instruction, escribir en pc lo hara el jump control
    control_WB_address_out : out std_logic_vector (4 downto 0);--rd
address
```

```
    control_EXE_branch     : out std_logic_vector (1 downto 0) -- dice si
existe posibilidad de branch y de escribir en pc 00 no salto 01 cond 10
uncond
  ) ;
end Instruction_Decode;

architecture Instruction_Decode_arc of Instruction_Decode is

  signal control_imm_aux : std_logic_vector (2 downto 0);

  component ID_Control_Unit
    port (
      instruction            : in  std_logic_vector (31 downto 0);
      control_imm            : out std_logic_vector (2 downto 0);
      control_EXE_data_mux   : out std_logic;-- multiplexa rs1 rs2 imm
etc
      control_EXE_LUI        : out std_logic;-- dice si es una LUI o no
      control_EXE_ALU        : out std_logic_vector (3 downto 0);-- elige
operacion de la ALU
      control_MEM            : out std_logic_vector (3 downto 0);-- marca
la operacion de MEM y si tiene signo.
      control_WB_sel         : out std_logic_vector (1 downto 0);--
multiplexa la salida de wb entre alu mem pc
      control_WB_wren        : out std_logic; -- dice si hay que escribir
en rd en la instruction, escribir en pc lo hara el jump control
      control_WB_address_out : out std_logic_vector (4 downto 0);
      control_EXE_branch     : out std_logic_vector (1 downto 0) -- dice
si existe posibilidad de branch y de escribir en pc 00 no salto 01 cond
10 uncond
  ) ;
  end component;

  component ID_Registers
    port (
      clock      : in  std_logic;
      reset_n    : in  std_logic;
      address_rd : in  std_logic_vector (4  downto 0); --0 si no se
necesita escribir
      data_in    : in  std_logic_vector (31 downto 0);
      address1   : in  std_logic_vector (4  downto 0);
      address2   : in  std_logic_vector (4  downto 0);
      dataRS1    : out std_logic_vector (31 downto 0);
      dataRS2    : out std_logic_vector (31 downto 0)
    ) ;
  end component;

  component ID_Immediate_module
    port (
      control_imm : in  std_logic_vector (2 downto 0);--U J I B S
I(shamt) nada (nada lleva cÃ³digo 110)
      instruction : in  std_logic_vector (31 downto 0); --solo los 24MSB
importan pero asi es mucho mas comodo de programar
```

```
      imm_out      : out std_logic_vector (31 downto 0)
    ) ;
  end component;

begin

  address_rs1 <= instruction(19 downto 15);
  address_rs2 <= instruction(24 downto 20);

  Registers : ID_Registers
    port map(
      clock => clock,
      reset_n => reset_n,
      address_rd => wb_address_in,
      data_in => wb_data_in,
      address1 => instruction(19 downto 15),
      address2 => instruction(24 downto 20),
      dataRS1 => data_rs1,
      dataRS2 => data_rs2
      );

  Immediate_decode : ID_Immediate_module
    port map(
      control_imm => control_imm_aux,
      instruction => instruction,
      imm_out     => imm
      );

  Control_Unit : ID_Control_Unit
    port map(
      instruction => instruction,
      control_imm => control_imm_aux,
      control_EXE_data_mux =>control_EXE_data_mux,
      control_EXE_LUI => control_EXE_LUI,
      control_EXE_ALU => control_EXE_ALU,
      control_MEM => control_MEM,
      control_WB_sel => control_WB_sel,
      control_WB_wren => control_WB_wren,
      control_WB_address_out => control_WB_address_out,
      control_EXE_branch => control_EXE_branch
      );

end  Instruction_Decode_arc;
```

1.2.1 ID_Control_Unit.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions
```

```
entity ID_Control_Unit is
  port (
    instruction             : in  std_logic_vector (31 downto 0);
    control_imm             : out std_logic_vector (2 downto 0);
    control_EXE_data_mux    : out std_logic;-- multiplexa rs1 rs2 imm etc
    control_EXE_LUI         : out std_logic;-- dice si es una LUI o no
    control_EXE_ALU         : out std_logic_vector (3 downto 0);-- elige
operacion de la ALU
    control_MEM             : out std_logic_vector (3 downto 0);-- marca
la operacion de MEM y si tiene signo.
    control_WB_sel          : out std_logic_vector (1 downto 0);--
multiplexa la salida de wb entre alu mem pc
    control_WB_wren         : out std_logic; -- dice si hay que escribir
en rd en la instruction, escribir en pc lo hara el jump control
    control_WB_address_out  : out std_logic_vector (4 downto 0);
    control_EXE_branch      : out std_logic_vector (1 downto 0) -- dice si
existe posibilidad de branch y de escribir en pc 00 no salto 01 cond 10
uncond
  ) ;

end ID_Control_Unit;

architecture ID_Control_Unit_arc of ID_Control_Unit is

begin

  control_imm <=
      --U
      "000" when (instruction(6 downto 0) = "0110111" OR instruction(6
downto 0) = "0010111") else
      --J
      "001" when (instruction(6 downto 0) = "1101111") else
      --I (Loads)
      "010" when ( (instruction(6 downto 0) = "0000011") OR
(instruction(6 downto 0) = "0010011" AND instruction (14 downto 12)
        /= "001" AND instruction (14 downto 12) /= "101" )) else
      --B
      "011" when (instruction(6 downto 0) = "1100011") else
      --S
      "100" when (instruction(6 downto 0) = "0100011") else
      --I(shamt)
      "101" when ( (instruction(6 downto 0) = "0010011")   AND
(instruction (14 downto 12)
        = "001" OR instruction (14 downto 12) = "101")  ) else
      --JALR
      "111" when (instruction(6 downto 0) = "1100111") else
      --others
      "110" ;

    control_EXE_data_mux <= -- multiplexa rs2 y imm .rs2 imm-0 1
        --rs2 R B
```

```
        '0' when ((instruction (6 downto 0) = "0110011") OR (instruction
(6 downto 0)) = "1100011") else
        '1';

    control_EXE_LUI <=
      --LUI
      '1' when instruction(6 downto 0) = "0110111" else
      --otros
      '0' ;

    control_EXE_ALU <=
      --ADD
      "0001" when ( (instruction(6 downto 0) = "1100111" ) OR --JALR
              (instruction(6 downto 0) = "0000011" ) OR --load
              (instruction(6 downto 0) = "0100011" ) OR --store
              (instruction(6 downto 0) = "0010011" AND  instruction (14
downto 12) = "000") OR --ADDI
              ((instruction(6 downto 0) = "0110011") AND (instruction (14
downto 12) = "000") AND (instruction (30) ='0')) OR --ADD
              (instruction(6 downto 0) = "0110111")) else --LUI
      --SUB
      "0010" when (instruction(6 downto 0) = "0110011" AND  instruction
(30) ='1' AND instruction (14 downto 12) = "000") else
      --AND ANDI
      "0011" when ((instruction(6 downto 0) = "0110011" OR instruction(6
downto 0) = "0010011") AND  instruction (14 downto 12) = "111") else
      --OR ORI
      "0100" when ((instruction(6 downto 0) = "0110011" OR instruction(6
downto 0) = "0010011") AND  instruction (14 downto 12) = "110") else
      --XOR XORI
      "0101" when ((instruction(6 downto 0) = "0110011" OR instruction(6
downto 0) = "0010011") AND  instruction (14 downto 12) = "100") else
      --SLL SLLI
      "0110" when ( (instruction(6 downto 0) = "0010011" AND  instruction
(14 downto 12) = "001") OR
              (instruction(6 downto 0) = "0110011" AND  instruction (14
downto 12) = "001") ) else
      --SRLI SRL
      "0111" when (  (instruction(6 downto 0) = "0010011" AND
instruction (14 downto 12) = "101" AND instruction(30) = '0') OR
               (instruction(6 downto 0) = "0110011" AND  instruction (14
downto 12) = "101" AND instruction(30) = '0') ) else
      --SRAI SRA
      "1000" when (  (instruction(6 downto 0) = "0010011" AND
instruction (14 downto 12) = "101" AND instruction(30) = '1') OR
               (instruction(6 downto 0) = "0110011" AND  instruction (14
downto 12) = "101" AND instruction(30) = '1') ) else
      --s1<s2 U   sltiu sltu bltu
      "1001" when ( (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "110") OR --bltu
             (instruction(6 downto 0) = "0110011" AND instruction(14
downto 12) = "011") OR --sltu
```

```
            (instruction(6 downto 0) = "0010011" AND instruction(14
downto 12) = "011") )  else --sltiu
      --s1>=s2 U bgeu
      "1010" when (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "111") else
      --s1=s2 beq
      "1011" when (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "000") else
      --s1/=s2
      "1100" when (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "001") else
      --s1<s2 S slt slti
      "1101" when ( (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "100") OR --blt
            (instruction(6 downto 0) = "0110011" AND instruction(14
downto 12) = "010") OR --slt
            (instruction(6 downto 0) = "0010011" AND instruction(14
downto 12) = "010") )  else --slti
      --s1>s2 S bge
      "1110" when (instruction(6 downto 0) = "1100011" AND  instruction
(14 downto 12) = "101") else
      --others
      "0000" ;

    control_MEM(3) <= -- marca la operacion de MEM y si tiene signo.MSB->
signed 1, unsigned 0. LB LW
      '1' when ((instruction(6 downto 0) = "0000011") AND (instruction(14
downto 12) = "000" OR instruction(14 downto 12) = "001")) else
      '0' ;
    control_MEM(2 downto 0) <= --LB LH LW SB SH SW Nada. 001 010 010 011
100 101 110
      --LB
      "001" when (instruction(6 downto 0) = "0000011" AND (instruction(14
downto 12) = "000" OR instruction(14 downto 12) = "100")) else
      --LH
      "010" when (instruction(6 downto 0) = "0000011" AND (instruction(14
downto 12) = "001" OR instruction(14 downto 12) = "101")) else
      --LW
      "011" when (instruction(6 downto 0) = "0000011" AND instruction(14
downto 12) = "010") else
      --SB
      "100" when (instruction(6 downto 0) = "0100011" AND instruction(14
downto 12) = "000") else
      --SH
      "101" when (instruction(6 downto 0) = "0100011" AND instruction(14
downto 12) = "001") else
      --SW
      "110" when (instruction(6 downto 0) = "0100011" AND instruction(14
downto 12) = "010") else
      --others
      "000" ;
```

```
    control_WB_sel <= -- multiplexa la salida de wb entre alu mem pc+imm
(el dato a escribir en registro cual es para esta instruct?)
              -- pc+4 alu mem pc+imm - 00 01 10 11
      --JAL JALR
      "00" when (instruction (6 downto 0) = "1101111" OR instruction (6
downto 0) = "1100111") else
      --ALU R I LUI
      "01" when (instruction (6 downto 0) = "0110011" OR instruction(6
downto 0) = "0010011" OR instruction(6 downto 0) = "0110111") else
      --MEM Loads
      "10" when (instruction (6 downto 0) = "0000011") else
      -- PC+imm B AUIPC
      "11" when ( (instruction (6 downto 0) = "1100011") OR (instruction
(6 downto 0) = ("0010111") ) )else
      "00" ;

    control_WB_wren <=
      --escribir en rd R JALR LOADS I LUI AUIPC JAL
      '1' when (instruction (6 downto 0) = "0110011" OR instruction (6
downto 0) = "1100111" OR instruction (6 downto 0) = "0000011"
            OR instruction (6 downto 0) = "0010011" OR instruction (6
downto 0) = "0110111" OR instruction (6 downto 0) = "0010111"
            OR instruction (6 downto 0) = "1101111") else
      --no escribir en rd
      '0' ;

    control_EXE_branch <=
      -- conditional branch B
      "01" when instruction (6 downto 0) = "1100011" else
      --JAL
      "10" when (instruction (6 downto 0) = "1101111") else
      --JALR
      "11" when instruction (6 downto 0) = "1100111" else
      -- los demas
      "00" ;

    control_WB_address_out <= instruction (11 downto 7);

end  ID_Control_Unit_arc;
```

1.2.2 ID_Demux32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity ID_Demux32 is
port (
  address    : in  std_logic_vector(4  downto 0);
```

```
  demux_out : out std_logic_vector(31 downto 0)
);
end ID_Demux32;

architecture behaviour of ID_Demux32 is

begin

  demux32to1: process (address)
  begin
    demux_out <= (others => '0');
      demux_out(to_integer(unsigned(address))) <= '1';

  end process;

end behaviour;
```

1.2.3 ID_Immediate_module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity  ID_Immediate_module is
  port (
      control_imm : in  std_logic_vector (2 downto 0);--U J I B S
I(shamt) nada (nada lleva código 110)
      instruction : in  std_logic_vector (31 downto 0); --solo los 24MSB
importan pero asi es mucho mas comodo de programar
      imm_out     : out std_logic_vector (31 downto 0)
  ) ;
end   ID_Immediate_module;

architecture ID_Immediate_module_arc of  ID_Immediate_module is

    signal sign_extend : std_logic_vector (31 downto 0);

begin

    sign_extend <= instruction(31) & instruction(31) & instruction(31) &
instruction(31) & instruction(31) & instruction(31) & instruction(31) &
instruction(31) &
            instruction(31) & instruction(31) & instruction(31) &
instruction(31) & instruction(31) & instruction(31) & instruction(31) &
instruction(31) &
            instruction(31) & instruction(31) & instruction(31) &
instruction(31) & instruction(31) & instruction(31) & instruction(31) &
```

```
            instruction(31) & instruction(31) & instruction(31) &
instruction(31) & instruction(31) & instruction(31) & instruction(31) &
instruction(31);

  imm_out <=

    --U
    instruction (31 downto 12) & "000000000000" when (control_imm =
"000") else
    --JAL
    sign_extend (31 downto 22) & instruction(31) & instruction (19 downto
12) & instruction (20) & instruction (31 downto 21) & '0' when
(control_imm = "001") else
    --I
    sign_extend(31 downto 12) & instruction (31 downto 20) when
(control_imm = "010") else
    --B
    sign_extend(31 downto 13) & instruction (31) & instruction (7) &
instruction (30 downto 25) & instruction (11 downto 8) & '0' when
(control_imm = "011") else
    --S
    sign_extend(31 downto 12) & instruction (31 downto 25) & instruction
(11 downto 7) when (control_imm = "100") else
    --I (shamt)
    "00000000000000000000000000000" & instruction (24 downto 20) when
(control_imm = "101") else
    --JALR
    sign_extend(31 downto 12) & instruction (31 downto 20) when
(control_imm = "111") else --estan desordenados porque me salte jalr

    --others ponemos 0 porque no son operaciones de imm, nada llevará
código 110
    "00000000000000000000000000000000";


end   ID_Immediate_module_arc;
```

1.2.4 ID_Mux32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity ID_Mux32 is
  port (
    data0   : in  std_logic_vector (31 downto 0);
    data1   : in  std_logic_vector (31 downto 0);
    data2   : in  std_logic_vector (31 downto 0);
    data3   : in  std_logic_vector (31 downto 0);
```

```
    data4  : in  std_logic_vector (31 downto 0);
    data5  : in  std_logic_vector (31 downto 0);
    data6  : in  std_logic_vector (31 downto 0);
    data7  : in  std_logic_vector (31 downto 0);
    data8  : in  std_logic_vector (31 downto 0);
    data9  : in  std_logic_vector (31 downto 0);
    data10 : in  std_logic_vector (31 downto 0);
    data11 : in  std_logic_vector (31 downto 0);
    data12 : in  std_logic_vector (31 downto 0);
    data13 : in  std_logic_vector (31 downto 0);
    data14 : in  std_logic_vector (31 downto 0);
    data15 : in  std_logic_vector (31 downto 0);
    data16 : in  std_logic_vector (31 downto 0);
    data17 : in  std_logic_vector (31 downto 0);
    data18 : in  std_logic_vector (31 downto 0);
    data19 : in  std_logic_vector (31 downto 0);
    data20 : in  std_logic_vector (31 downto 0);
    data21 : in  std_logic_vector (31 downto 0);
    data22 : in  std_logic_vector (31 downto 0);
    data23 : in  std_logic_vector (31 downto 0);
    data24 : in  std_logic_vector (31 downto 0);
    data25 : in  std_logic_vector (31 downto 0);
    data26 : in  std_logic_vector (31 downto 0);
    data27 : in  std_logic_vector (31 downto 0);
    data28 : in  std_logic_vector (31 downto 0);
    data29 : in  std_logic_vector (31 downto 0);
    data30 : in  std_logic_vector (31 downto 0);
    data31 : in  std_logic_vector (31 downto 0);
    d_out  : out std_logic_vector (31 downto 0);
    sel    : in  std_logic_vector (4  downto 0)
  ) ;
end ID_Mux32;

architecture ID_Mux32_arc of ID_Mux32 is

  signal data_aux : std_logic_vector (31 downto 0);

begin

  d_out <= data_aux;

  with sel select data_aux <=
    data0  when "00000",
    data1  when "00001",
    data2  when "00010",
    data3  when "00011",
    data4  when "00100",
    data5  when "00101",
    data6  when "00110",
    data7  when "00111",
    data8  when "01000",
    data9  when "01001",
```

```
    data10 when "01010",
    data11 when "01011",
    data12 when "01100",
    data13 when "01101",
    data14 when "01110",
    data15 when "01111",
    data16 when "10000",
    data17 when "10001",
    data18 when "10010",
    data19 when "10011",
    data20 when "10100",
    data21 when "10101",
    data22 when "10110",
    data23 when "10111",
    data24 when "11000",
    data25 when "11001",
    data26 when "11010",
    data27 when "11011",
    data28 when "11100",
    data29 when "11101",
    data30 when "11110",
    data31 when "11111",
    data0  when others;



end  ID_Mux32_arc;
```

1.3.1 ID_Registers

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity ID_Registers is
port (
clock      : in  std_logic;
reset_n    : in  std_logic;
address_rd : in  std_logic_vector (4  downto 0); --0 si no se necesita
escribir
data_in    : in  std_logic_vector (31 downto 0);
address1   : in  std_logic_vector (4  downto 0);
address2   : in  std_logic_vector (4  downto 0);
dataRS1    : out std_logic_vector (31 downto 0);
dataRS2    : out std_logic_vector (31 downto 0)
) ;

end ID_Registers;
```

```
architecture ID_Registers_arc of ID_Registers is

component Register32bits
port (
clock  : in  std_logic;
enable : in  std_logic;
clear  : in  std_logic;
d      : in  std_logic_vector (31 downto 0);
q      : out std_logic_vector (31 downto 0)
) ;
end component;

component ID_Demux32
port (
address   : in  std_logic_vector(4  downto 0);
demux_out : out std_logic_vector(31 downto 0)
);
end component;

component ID_Mux32
port (
data0  : in  std_logic_vector (31 downto 0);
data1  : in  std_logic_vector (31 downto 0);
data2  : in  std_logic_vector (31 downto 0);
data3  : in  std_logic_vector (31 downto 0);
data4  : in  std_logic_vector (31 downto 0);
data5  : in  std_logic_vector (31 downto 0);
data6  : in  std_logic_vector (31 downto 0);
data7  : in  std_logic_vector (31 downto 0);
data8  : in  std_logic_vector (31 downto 0);
data9  : in  std_logic_vector (31 downto 0);
data10 : in  std_logic_vector (31 downto 0);
data11 : in  std_logic_vector (31 downto 0);
data12 : in  std_logic_vector (31 downto 0);
data13 : in  std_logic_vector (31 downto 0);
data14 : in  std_logic_vector (31 downto 0);
data15 : in  std_logic_vector (31 downto 0);
data16 : in  std_logic_vector (31 downto 0);
data17 : in  std_logic_vector (31 downto 0);
data18 : in  std_logic_vector (31 downto 0);
data19 : in  std_logic_vector (31 downto 0);
data20 : in  std_logic_vector (31 downto 0);
data21 : in  std_logic_vector (31 downto 0);
data22 : in  std_logic_vector (31 downto 0);
data23 : in  std_logic_vector (31 downto 0);
data24 : in  std_logic_vector (31 downto 0);
data25 : in  std_logic_vector (31 downto 0);
data26 : in  std_logic_vector (31 downto 0);
data27 : in  std_logic_vector (31 downto 0);
data28 : in  std_logic_vector (31 downto 0);
data29 : in  std_logic_vector (31 downto 0);
```

```
data30 : in  std_logic_vector (31 downto 0);
data31 : in  std_logic_vector (31 downto 0);
d_out  : out std_logic_vector (31 downto 0);
sel    : in  std_logic_vector (4  downto 0)
) ;
end component;

signal demux_out_aux : std_logic_vector (31 downto 0);
signal data0_aux     : std_logic_vector (31 downto 0);
signal data1_aux     : std_logic_vector (31 downto 0);
signal data2_aux     : std_logic_vector (31 downto 0);
signal data3_aux     : std_logic_vector (31 downto 0);
signal data4_aux     : std_logic_vector (31 downto 0);
signal data5_aux     : std_logic_vector (31 downto 0);
signal data6_aux     : std_logic_vector (31 downto 0);
signal data7_aux     : std_logic_vector (31 downto 0);
signal data8_aux     : std_logic_vector (31 downto 0);
signal data9_aux     : std_logic_vector (31 downto 0);
signal data10_aux    : std_logic_vector (31 downto 0);
signal data11_aux    : std_logic_vector (31 downto 0);
signal data12_aux    : std_logic_vector (31 downto 0);
signal data13_aux    : std_logic_vector (31 downto 0);
signal data14_aux    : std_logic_vector (31 downto 0);
signal data15_aux    : std_logic_vector (31 downto 0);
signal data16_aux    : std_logic_vector (31 downto 0);
signal data17_aux    : std_logic_vector (31 downto 0);
signal data18_aux    : std_logic_vector (31 downto 0);
signal data19_aux    : std_logic_vector (31 downto 0);
signal data20_aux    : std_logic_vector (31 downto 0);
signal data21_aux    : std_logic_vector (31 downto 0);
signal data22_aux    : std_logic_vector (31 downto 0);
signal data23_aux    : std_logic_vector (31 downto 0);
signal data24_aux    : std_logic_vector (31 downto 0);
signal data25_aux    : std_logic_vector (31 downto 0);
signal data26_aux    : std_logic_vector (31 downto 0);
signal data27_aux    : std_logic_vector (31 downto 0);
signal data28_aux    : std_logic_vector (31 downto 0);
signal data29_aux    : std_logic_vector (31 downto 0);
signal data30_aux    : std_logic_vector (31 downto 0);
signal data31_aux    : std_logic_vector (31 downto 0);


begin

Mux_1 : ID_Mux32
port map(
data0  => data0_aux,
data1  => data1_aux,
data2  => data2_aux,
data3  => data3_aux,
data4  => data4_aux,
data5  => data5_aux,
```

```
data6  => data6_aux,
data7  => data7_aux,
data8  => data8_aux,
data9  => data9_aux,
data10 => data10_aux,
data11 => data11_aux,
data12 => data12_aux,
data13 => data13_aux,
data14 => data14_aux,
data15 => data15_aux,
data16 => data16_aux,
data17 => data17_aux,
data18 => data18_aux,
data19 => data19_aux,
data20 => data20_aux,
data21 => data21_aux,
data22 => data22_aux,
data23 => data23_aux,
data24 => data24_aux,
data25 => data25_aux,
data26 => data26_aux,
data27 => data27_aux,
data28 => data28_aux,
data29 => data29_aux,
data30 => data30_aux,
data31 => data31_aux,
d_out  => dataRS1,
sel    => address1
);

Mux_2 : ID_Mux32
port map(
data0  => data0_aux,
data1  => data1_aux,
data2  => data2_aux,
data3  => data3_aux,
data4  => data4_aux,
data5  => data5_aux,
data6  => data6_aux,
data7  => data7_aux,
data8  => data8_aux,
data9  => data9_aux,
data10 => data10_aux,
data11 => data11_aux,
data12 => data12_aux,
data13 => data13_aux,
data14 => data14_aux,
data15 => data15_aux,
data16 => data16_aux,
data17 => data17_aux,
data18 => data18_aux,
data19 => data19_aux,
```

```
data20 => data20_aux,
data21 => data21_aux,
data22 => data22_aux,
data23 => data23_aux,
data24 => data24_aux,
data25 => data25_aux,
data26 => data26_aux,
data27 => data27_aux,
data28 => data28_aux,
data29 => data29_aux,
data30 => data30_aux,
data31 => data31_aux,
d_out  => dataRS2,
sel    => address2
);

demux : ID_Demux32
port map (
address => address_rd,
demux_out => demux_out_aux
);


x0 : Register32bits
port map(
clock  => clock,
enable => '0',
clear  => '1',
d      => "00000000000000000000000000000000",
q      => data0_aux
);


x1 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(1),
clear  => reset_n,
d      => data_in,
q      => data1_aux
);

x2 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(2),
clear  => reset_n,
d      => data_in,
q      => data2_aux
);

x3 : Register32bits
```

```
port map(
clock  => clock,
enable => demux_out_aux(3),
clear  => reset_n,
d      => data_in,
q      => data3_aux
);

x4 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(4),
clear  => reset_n,
d      => data_in,
q      => data4_aux
);

x5 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(5),
clear  => reset_n,
d      => data_in,
q      => data5_aux
);

x6 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(6),
clear  => reset_n,
d      => data_in,
q      => data6_aux
);

x7 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(7),
clear  => reset_n,
d      => data_in,
q      => data7_aux
);

x8 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(8),
clear  => reset_n,
d      => data_in,
q      => data8_aux
);
```

```
x9 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(9),
clear  => reset_n,
d      => data_in,
q      => data9_aux
);

x10 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(10),
clear  => reset_n,
d      => data_in,
q      => data10_aux
);

x11 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(11),
clear  => reset_n,
d      => data_in,
q      => data11_aux
);

x12 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(12),
clear  => reset_n,
d      => data_in,
q      => data12_aux
);

x13 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(13),
clear  => reset_n,
d      => data_in,
q      => data13_aux
);

x14 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(14),
clear  => reset_n,
d      => data_in,
```

```
q      => data14_aux
);

x15 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(15),
clear  => reset_n,
d      => data_in,
q      => data15_aux
);

x16 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(16),
clear  => reset_n,
d      => data_in,
q      => data16_aux
);

x17 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(17),
clear  => reset_n,
d      => data_in,
q      => data17_aux
);

x18 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(18),
clear  => reset_n,
d      => data_in,
q      => data18_aux
);

x19 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(19),
clear  => reset_n,
d      => data_in,
q      => data19_aux
);

x20 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(20),
```

```
clear  => reset_n,
d      => data_in,
q      => data20_aux
);

x21 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(21),
clear  => reset_n,
d      => data_in,
q      => data21_aux
);

x22 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(22),
clear  => reset_n,
d      => data_in,
q      => data22_aux
);

x23 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(23),
clear  => reset_n,
d      => data_in,
q      => data23_aux
);

x24 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(24),
clear  => reset_n,
d      => data_in,
q      => data24_aux
);

x25 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(25),
clear  => reset_n,
d      => data_in,
q      => data25_aux
);

x26 : Register32bits
port map(
```

```
clock  => clock,
enable => demux_out_aux(26),
clear  => reset_n,
d      => data_in,
q      => data26_aux
);

x27 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(27),
clear  => reset_n,
d      => data_in,
q      => data27_aux
);

x28 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(28),
clear  => reset_n,
d      => data_in,
q      => data28_aux
);

x29 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(29),
clear  => reset_n,
d      => data_in,
q      => data29_aux
);

x30 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(30),
clear  => reset_n,
d      => data_in,
q      => data30_aux
);

x31 : Register32bits
port map(
clock  => clock,
enable => demux_out_aux(31),
clear  => reset_n,
d      => data_in,
q      => data31_aux
);
```

```
end  ID_Registers_arc;
```

2.2 EXECUTION.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity Execution is
  port (
    pc               : in  std_logic_vector (31 downto 0);
    imm_in           : in  std_logic_vector (31 downto 0);
    control_data_mux : in  std_logic;
    rs1_data_in      : in  std_logic_vector (31 downto 0);
    rs2_data_in      : in  std_logic_vector (31 downto 0);
    control_LUI      : in  std_logic;
    control_jump     : in  std_logic_vector (1 downto 0);
    control_ALU      : in  std_logic_vector (3  downto 0);
    ALU_result       : out std_logic_vector (31 downto 0);
    pc_imm           : out std_logic_vector (31 downto 0);
    rs2_out_store    : out std_logic_vector (31 downto 0);
    jump_flush       : out std_logic;
    forward_sel_1    : in  std_logic_vector (1 downto 0);
    forward_sel_2    : in  std_logic_vector (1 downto 0);
    MEM_data         : in  std_logic_vector (31 downto 0);
    WB_data          : in  std_logic_vector (31 downto 0);
    ID_data          : in  std_logic_vector (31 downto 0)
  ) ;
end Execution;

architecture Execution_arc of Execution is

  signal data1_aux      : std_logic_vector (31 downto 0);
  signal data2_aux      : std_logic_vector (31 downto 0);
  signal ALU_result_aux : std_logic_vector (31 downto 0);
  signal pc_imm_aux     : std_logic_vector (31 downto 0);
  signal forwarded1     : std_logic_vector (31 downto 0);
  signal forwarded2     : std_logic_vector (31 downto 0);

  component EXE_Jump_Flush_module
    port (
      control_jump  : in  std_logic_vector (1 downto 0); --00 instruction
sin salto. 01 instruction salto condicional. 10 salto incondicional
      ALU_result    : in  std_logic_vector (31 downto 0);
      flush         : out std_logic;
      pc_imm_in     : in  std_logic_vector(31 downto 0);
      pc_imm_out    : out std_logic_vector(31 downto 0)
      ) ;
```

```
  end component;

  component EXE_LUI_Mux2x1
    port (
      control_LUI : in  std_logic; --1 si LUI, 1 si otra
      rs1_data    : in  std_logic_vector (31 downto 0);
      data1_out   : out std_logic_vector (31 downto 0)
      ) ;
  end component;

  component EXE_Data_Mux2x1
    port (
      control_data_selector : in  std_logic; --0=rs1&rs2   1=rs1&imm
      imm_in                : in  std_logic_vector (31 downto 0);
      rs2_data              : in  std_logic_vector (31 downto 0);
      data2_out             : out std_logic_vector (31 downto 0)
      ) ;
  end component;

  component EXE_PC_Adder
    port (
      pc_in        : in  std_logic_vector (31 downto 0);
      imm          : in  std_logic_vector (31 downto 0);
      pc_adder_out : out std_logic_vector (31 downto 0)
      ) ;
  end component;

  component EXE_ALU32I
    port (
      data1  : in  std_logic_vector (31 downto 0);
      data2  : in  std_logic_vector (31 downto 0);
      sel    : in  std_logic_vector (3  downto 0);
      result : out std_logic_vector (31 downto 0)
    ) ;
  end component;

  component EXE_forward_Mux is
    port (
      sel      : in  std_logic_vector ( 1 downto 0);
      EXE_data : in  std_logic_vector (31 downto 0);
      MEM_data : in  std_logic_vector (31 downto 0);
      WB_data  : in  std_logic_vector (31 downto 0);
      ID_data  : in  std_logic_vector (31 downto 0);
      data_out : out std_logic_vector (31 downto 0)
      ) ;
  end component;

begin

  LUI_Mux : EXE_LUI_Mux2x1
    port map(
      control_LUI => control_LUI,
```

```
    rs1_data => rs1_data_in,
    data1_out => data1_aux
    );

Data_Mux : EXE_Data_Mux2x1
  port map(
    control_data_selector => control_data_mux,
    imm_in => imm_in,
    rs2_data => forwarded2,
    data2_out => data2_aux
    );

PC_adder : EXE_PC_Adder
  port map(
    pc_in =>pc,
    imm =>imm_in,
    pc_adder_out => pc_imm_aux
    );

ALU : EXE_ALU32I
port map(
  data1 => forwarded1,
  data2 => data2_aux ,
  sel   => control_ALU,
  result => ALU_result_aux
  );

Jump_control : EXE_Jump_Flush_module
port map(
  control_jump  => control_jump,
  ALU_result    => ALU_result_aux,
  flush => jump_flush,
  pc_imm_in =>  pc_imm_aux,
  pc_imm_out => pc_imm
  );

forward_mux_1 : EXE_forward_Mux
  port map(
    sel       => forward_sel_1,
    EXE_data => data1_aux,
    MEM_data => MEM_data,
    WB_data  => WB_data,
    ID_data  => ID_data,
    data_out => forwarded1
    );

forward_mux_2 : EXE_forward_Mux
  port map(
    sel       => forward_sel_2,
    EXE_data => rs2_data_in,
    MEM_data => MEM_data,
    WB_data  => WB_data,
```

```
      ID_data  => ID_data,
      data_out => forwarded2
      );


  ALU_result <= ALU_result_aux;
  rs2_out_store <= forwarded2;

end  Execution_arc;
```

1.3.2 EXE_ALU32I.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_ALU32I is
  port (
    data1  : in  std_logic_vector (31 downto 0);
    data2  : in  std_logic_vector (31 downto 0);
    sel    : in  std_logic_vector (3  downto 0);
    result : out std_logic_vector (31 downto 0)
  ) ;
end EXE_ALU32I;

architecture EXE_ALU32I_arc of EXE_ALU32I is

begin

  sel_case : process (sel, data1, data2)

  begin

    case sel is
      when "0001" => --ADD JALR LOAD STORE ADDI ADD LUI
        result <= (data1 + data2);

      when "0010" => --SUB
        result <= (data1 - data2);

      when "0011" => --AND
        result <= data1 AND data2;

      when "0100" => --OR
        result <= data1 OR data2;

      when "0101" => --XOR
        result <= data1 XOR data2;
```

```
when "0110" => --SLU
  result <= std_logic_vector(shift_left(unsigned(data1),
    to_integer(unsigned(data2(4 downto 0)))));

when "0111" => --SRU
  result <= std_logic_vector(shift_right(unsigned(data1),
    to_integer(unsigned(data2(4 downto 0)))));

when "1000" => --SRS
  result <= std_logic_vector(shift_right(signed(data1),
    to_integer(unsigned(data2(4 downto 0)))));

when "1001" => --s1<s2 U
  if data1 < data2 then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
  end if;
when "1010" => --s1>=s2 U
  if data1 >= data2 then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
  end if;

when "1011" => --s1=s2
  if (data1 = data2) then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
  end if;

when "1100" => --s1!=s2
  if (data1 /= data2) then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
  end if;

when "1101" => --s1<s2 S
  if (signed (data1) < signed(data2)) then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
  end if;

when "1110" =>--s1>=s2 S BGE
  if (signed(data1) >= signed(data2)) then
    result <= std_logic_vector(to_unsigned(1, 32));
  else
    result <= std_logic_vector(to_unsigned(0, 32));
```

```
        end if;

      when others =>
        result <= std_logic_vector(to_unsigned(0,32));
    end case;
  end process;

end  EXE_ALU32I_arc;
```

1.3.3 EXE_Data_Mux2x1.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_Data_Mux2x1 is --selecciona los datos que entran a la ALU, los
imm siempre son data2
  port (
    control_data_selector : in  std_logic; --0=rs1&rs2   1=rs1&imm
    imm_in                : in  std_logic_vector (31 downto 0);
    rs2_data              : in  std_logic_vector (31 downto 0);
    data2_out             : out std_logic_vector (31 downto 0)
  ) ;
end EXE_Data_Mux2x1;

architecture EXE_Data_Mux2x1_arc of EXE_Data_Mux2x1 is

begin

  data2_out <=
    --rs2
    rs2_data when (control_data_selector = '0') else
    imm_in   when (control_data_selector = '1') ;

end  EXE_Data_Mux2x1_arc;
```

1.3.4 EXE _Flush_module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_Flush_module is
  port (
```

```
    control_jump  : in  std_logic_vector (1 downto 0); --00 instruction
sin salto. 01 instruction salto condicional. 10 salto incondicional si la
instruccion puede tener un salto
    ALU_resultLSB : in  std_logic;
    flush         : out std_logic
    ) ;
end EXE_Flush_module;

architecture EXE_Flush_module_arc of EXE_Flush_module is

begin

  flush <=
    --condicional
    '1' when (control_jump = "01" AND ALU_resultLSB = '1') else
    --incondicional
    '1' when (control_jump = "10") else
    --sin salto
    '0';

end  EXE_Flush_module_arc;
```

1.3.5 EXE_LUI_Mux2x1.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_LUI_Mux2x1 is
  port (
    control_LUI : in  std_logic; --1 si LUI, 1 si otra
    rs1_data    : in  std_logic_vector (31 downto 0);
    data1_out   : out std_logic_vector (31 downto 0)
  ) ;
end EXE_LUI_Mux2x1;

architecture EXE_LUI_Mux2x1_arc of EXE_LUI_Mux2x1 is

begin

  data1_out <=
    --Cualquier intruction menos LUI o store
    rs1_data when (control_LUI = '0') else
    --LUI o cosas raras
    "00000000000000000000000000000000";


end  EXE_LUI_Mux2x1_arc;
```

1.3.6 EXE_PC_Adder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_PC_Adder is
  port (
    pc_in        : in  std_logic_vector (31 downto 0);
    imm          : in  std_logic_vector (31 downto 0);
    pc_adder_out : out std_logic_vector (31 downto 0)
  ) ;
end EXE_PC_Adder;

architecture EXE_PC_Adder_arc of EXE_PC_Adder is

begin

  pc_adder_out <= imm + pc_in; --El bloque es muy simple pero en las
instrucciones de control
                --la ALU está ocupada determinando si branchea así que
algo tiene que calcular el salto

end  EXE_PC_Adder_arc;
```

1.3.7 EXE_Forward_Mux

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity EXE_forward_Mux is
  port (
    sel      : in  std_logic_vector ( 1 downto 0);
    EXE_data : in  std_logic_vector (31 downto 0);
    MEM_data : in  std_logic_vector (31 downto 0);
    WB_data  : in  std_logic_vector (31 downto 0);
    ID_data  : in  std_logic_vector (31 downto 0);
    data_out : out std_logic_vector (31 downto 0)
  ) ;
end EXE_forward_Mux;

architecture EXE_forward_Mux_arc of EXE_forward_Mux is

begin
```

```
  data_out <=
    EXE_data when sel = "00" else
    MEM_data when sel = "01" else
    WB_data  when sel = "10" else
    ID_data  when sel = "11";


end  EXE_forward_Mux_arc;
```

1.4 MEMORY.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity  Memory is

  generic(
    words_num  : integer := 256;
    addr_width : integer := 8
        );

  port (
    reset_n          : in  std_logic;
    clock            : in  std_logic;
    ALU_result_in    : in  std_logic_vector (31 downto 0); --resultado de
la ALU (es la address en stores y loads)
    data_in          : in  std_logic_vector (31 downto 0); --dato a cargar
en instrucciones store (rs2)
    control_MEM_in   : in  std_logic_vector (3  downto 0); --control word
de la etapa MEM. MSB-> signed 1, unsigned 0. 3LSB (LB,LH...)
    memory_out       : out std_logic_vector (31 downto 0); --dato que sale
de la memoria
    ALU_result_out   : out std_logic_vector (31 downto 0); --res de alu
por si se necesita en wb, no se modifica
    load_address     : in  std_logic_vector (31 downto 0);
    load_control_in  : in  std_logic_vector (3  downto 0);
    load_control_out: out std_logic_vector (3  downto 0);
    control_WB_sel   : in  std_logic_vector (1  downto 0);
    PC_imm           : in  std_logic_vector (31 downto 0);
    PC_4_in          : in  std_logic_vector (31 downto 0)
  ) ;
end   Memory;

architecture Memory_arc of  Memory is

    signal ALU_result_aux : std_logic_vector (31 downto 0);
```

```
    signal byteen_aux : std_logic_vector (3 downto 0);
    signal en_vec_aux : std_logic_vector (31 downto 0);
    signal data_aux : std_logic_vector (31 downto 0);
    signal data_mem_load : std_logic_vector (31 downto 0);
    signal control_MEM_aux : std_logic_vector(3 downto 0);
    signal load_control : std_logic_vector(3 downto 0);
    signal re_aux : std_logic;
    signal mem_out : std_logic_vector (31 downto 0);

    component MEM_Load_module
      port (
          ALU_result2LSB      : in  std_logic_vector (1 downto 0); --2LSB
de ALU_result (da info del byte)
          Control_signed      : in  std_logic; --1 signed, 0 unsigned
          control_instruction : in  std_logic_vector (2 downto 0); --tipo
de operacion LB LH LW SB SH SW Nada. 000 001 010 011 100 101 110
          data_in             : in  std_logic_vector (31 downto 0); --
dato de entrada
          data_out            : out std_logic_vector (31 downto 0)
         );
    end component;


    component MEM_RAM
      port (
        clock      : in  std_logic;
        re         : in  std_logic;
        byteen     : in  std_logic_vector (3 downto 0);
        address    : in  std_logic_vector (addr_width-1 downto 2);
        data_in    : in  std_logic_vector (31 downto 0);
        data_out   : out std_logic_vector (31 downto 0)
      ) ;
    end component;

    component MEM_Byteen_module
      port (
        ALU_result2LSB : in  std_logic_vector (1 downto 0);            --
solo importan los 2 lsb en byte y el 2lsb en halfword, no importa en word
        control_store  : in  std_logic_vector (2  downto 0); --
LB,LW,LH,SB,SH,SW,Nada. 000 001 010 011 100 101 110
        data_in        : in  std_logic_vector (31 downto 0);-- el dato a
escribir en memoria
        data_out       : out std_logic_vector (31 downto 0);-- le hacemos
una mascara al dato para que lo escriba bien
                                -- sin esto copia las posiciones una a
una (bit0 por bit 0, bit 1 por bit1 etc)
        byteen         : out std_logic_vector (3 downto 0);-- controla
los bytes a escribir. Se divide la palabra en 4 bytes abcd. 0101 es
escribir b y d.
        re_out         : out std_logic
        ) ;
    end component;
```

```
    component MEM_Write_Back_select
      port (
        Memory_data     : in  std_logic_vector (31 downto 0);
        PC_imm          : in  std_logic_vector (31 downto 0);
        ALU_Data        : in  std_logic_vector (31 downto 0);
        PC_4            : in  std_logic_vector (31 downto 0);
        control_WB_sel  : in  std_logic_vector (1  downto 0); -- 00 no
necesita wb, 01 ALU, 10 MEM, 11 PC.
        WB_data_out : out std_logic_vector (31 downto 0)
        ) ;
    end component;

begin

  ALU_result_aux <= ALU_result_in;
  ALU_result_out <= ALU_result_aux;
  load_control <= load_control_in;
  load_control_out <= control_MEM_in;


  Data_memory : MEM_RAM
    port map(
      clock    => clock,
      data_in  => data_aux,
      address  => ALU_result_aux(addr_width-1 downto 2),
      re       => '1',
      byteen   => byteen_aux,
      data_out => data_mem_load
        );

  Byteen : MEM_Byteen_module
    port map(
      ALU_result2LSB => ALU_result_aux (1 downto 0),
      control_store  => control_MEM_in (2 downto 0),
      data_in        => data_in,
      data_out       => data_aux,
      byteen         => byteen_aux,
      re_out         => re_aux
        );

  Load_module : MEM_Load_module
    port map(
      ALU_result2LSB => ALU_result_aux (1 downto 0),
      Control_signed => load_control(3),
      control_instruction => load_control (2 downto 0),
      data_in  => data_mem_load,
      data_out => mem_out
        );

  Data_select : MEM_Write_Back_select
    port map(
```

```
      Memory_data => mem_out,
      ALU_Data => ALU_result_in,
      PC_4 => PC_4_in,
      PC_imm    => PC_imm,
      control_WB_sel => control_WB_sel,
      WB_data_out => memory_out
        );

end   Memory_arc;
```

1.4.1 MEM_Byteen_module

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity  MEM_Byteen_module is --modulo de los stores
  port (
    ALU_result2LSB : in  std_logic_vector (1 downto 0);            --solo
importan los 2 lsb en byte y el 2lsb en halfword, no importa en word
    control_store  : in  std_logic_vector (2  downto 0); --
LB,LW,LH,SB,SH,SW,Nada. 001 010 011 100 101 110 000
    data_in        : in  std_logic_vector (31 downto 0);-- el dato a
escribir en memoria
    data_out       : out std_logic_vector (31 downto 0);-- le hacemos una
mascara al dato para que lo escriba bien
                           -- sin esto copia las posiciones una a una
(bit0 por bit 0, bit 1 por bit1 etc)
    byteen         : out std_logic_vector (3 downto 0);-- controla los
bytes a escribir. Se divide la palabra en 4 bytes abcd. 0101 es escribir
b y d.
    re_out         : out std_logic
  ) ;
end   MEM_Byteen_module;

architecture MEM_Byteen_module_arc of   MEM_Byteen_module is

  signal data_aux : std_logic_vector (31 downto 0);

begin


  byteen <=
      --SB
       "0001" when (control_store = "100" AND ALU_result2LSB (1 downto 0)
= "00") else
       "0010" when (control_store = "100" AND ALU_result2LSB (1 downto 0)
= "01") else
```

```
        "0100" when (control_store = "100" AND ALU_result2LSB (1 downto 0)
= "10") else
        "1000" when (control_store = "100" AND ALU_result2LSB (1 downto 0)
= "11") else
       --SH
        "0011" when (control_store = "101" AND ALU_result2LSB(1) = '0')
else
        "1100" when (control_store = "101" AND ALU_result2LSB(1) = '1')
else
        --SW
        "1111" when (control_store = "110") else
        "0000" ;
  data_aux <=
       --SB
       "00000000" & "00000000" & "00000000" & data_in (7 downto 0) when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "00") else
       "00000000" & "00000000" & data_in (7 downto 0) & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "01") else
       "00000000" & data_in (7 downto 0) & "00000000" & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "10") else
       data_in (7 downto 0) & "00000000" & "00000000" & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "11") else
       --SH
       "00000000" & "00000000" & data_in (15 downto 0) when (control_store
= "101" AND ALU_result2LSB(1) = '0') else
       data_in (15 downto 0) & "00000000" & "00000000" when (control_store
= "101" AND ALU_result2LSB(1) = '1') else
       --SW
       data_in when (control_store = "110") else
       "00000000000000000000000000000000" ;
  data_out <= data_aux;

  re_out <=
    '1' when control_store = "100" or control_store = "101" or
control_store = "110" else
    '0';



end   MEM_Byteen_module_arc;
```

1.4.2 MEM_RAM_8.vhd

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

entity MEM_RAM_8 is
  generic(
```

```
    words_num  : integer := 256;
    addr_width : integer := 8
    );

  port (
    clock    : in  std_logic ;
    d_in     : in  std_logic_vector (7 downto 0);
    address  : in  std_logic_vector (addr_width-1 downto 2);
    re       : in  std_logic ; -- Read enable
    we       : in  std_logic ; -- Write enable
    d_out    : out std_logic_vector (7 downto 0)
    );
end MEM_RAM_8;

architecture rtl of MEM_RAM_8 is

type mem_t is array(0 to words_num-1) of std_logic_vector (7 downto 0);

signal RAM_byte : mem_t;

begin

  process (clock)

begin

    if we = '0' then
      d_out <= RAM_byte ( to_integer (unsigned(address ))));
    elsif (we = '1') and rising_edge(clock) then
      RAM_byte ( to_integer (unsigned(address ))) <= d_in;
    end if;

end process;

end rtl;
```

1.4.3 MEM_RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity MEM_RAM is

  generic(
    addr_width : integer := 8
    );

  port (
```

```
    clock       : in  std_logic;
    re          : in  std_logic;
    byteen      : in  std_logic_vector (3 downto 0);
    address     : in  std_logic_vector (addr_width-1 downto 2);
    data_in     : in  std_logic_vector (31 downto 0);
    data_out    : out std_logic_vector (31 downto 0)
  ) ;

end MEM_RAM;

architecture MEM_RAM_arc of MEM_RAM is

  type t_data   is array(0 to 3) of std_logic_vector (7 downto 0);


  signal t_data_in  : t_data;
  signal t_data_out : t_data;


  component MEM_RAM_8
    port (
      clock    : in  std_logic ;
      d_in     : in  std_logic_vector (7 downto 0);
      address  : in  std_logic_vector (addr_width-1 downto 2);
      re       : in  std_logic ; -- Read enable
      we       : in  std_logic ; -- Write enable
      d_out    : out std_logic_vector (7 downto 0)
      );
  end component;

begin

  gen_RAM : for i in 0 to 3 generate
    RAM : MEM_RAM_8
      port map(
        clock      => clock,
        we         => byteen(3-i),
        re         => re,
        d_in       => t_data_in(i),
        address    => address(addr_width-1 downto 2),
        d_out      => t_data_out(i)
        );
  end generate gen_RAM;

  data_out <= t_data_out(0) & t_data_out(1) & t_data_out(2) &
t_data_out(3);
  t_data_in(3) <= data_in (7  downto 0);
  t_data_in(2) <= data_in (15 downto 8);
  t_data_in(1) <= data_in (23 downto 16);
  t_data_in(0) <= data_in (31 downto 24);

end  MEM_RAM_arc;
```

1.4.4 MEM_Load_Module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity  MEM_Load_module is --se usa en operaciones load
  port (
    ALU_result2LSB      : in  std_logic_vector (1 downto 0); --2LSB de
alusesult (da info del byte)
    Control_signed      : in  std_logic; --1 signed, 0 unsigned
    control_instruction : in  std_logic_vector (2 downto 0); --tipo de
operacion LB LH LW SB SH SW Nada. 000 001 010 011 100 101 110
    data_in             : in  std_logic_vector (31 downto 0); --dato de
entrada
    data_out            : out std_logic_vector (31 downto 0)  -- dato de
salida a escribir en registro
  ) ;
end   MEM_Load_module;

architecture MEM_Load_module_arc of   MEM_Load_module is

  signal data_aux : std_logic_vector (31 downto 0);
  signal LB1 : std_logic_vector (23 downto 0);
  signal LB2 : std_logic_vector (23 downto 0);
  signal LB3 : std_logic_vector (23 downto 0);
  signal LB4 : std_logic_vector (23 downto 0);
  signal LH1 : std_logic_vector (15 downto 0);
  signal LH2 : std_logic_vector (15 downto 0);

begin

  data_aux <=
      --LB (signed)
      LB1 & data_in (7  downto 0 ) when (control_instruction = "001" AND
ALU_result2LSB = "00" AND Control_signed  = '1') else
      LB2 & data_in (15 downto 8 ) when (control_instruction = "001" AND
ALU_result2LSB = "01" AND Control_signed  = '1') else
      LB3 & data_in (23 downto 16) when (control_instruction = "001" AND
ALU_result2LSB = "10" AND Control_signed  = '1') else
      LB4 & data_in (31 downto 24) when (control_instruction = "001" AND
ALU_result2LSB = "11" AND Control_signed  = '1') else
      --LH (signed)
      LH1 & data_in (15 downto 0)  when (control_instruction = "010" AND
ALU_result2LSB(1) = '0' AND Control_signed  = '1') else
      LH2 & data_in (31 downto 16) when (control_instruction = "010" AND
ALU_result2LSB(1) = '1' AND Control_signed  = '1') else
      --LBU (unsigned)
```

```
      ("00000000000000000000000" & data_in (7  downto 0 )) when
(control_instruction = "001" AND ALU_result2LSB = "00" AND Control_signed
  = '0') else
      ("00000000000000000000000" & data_in (15 downto 8 )) when
(control_instruction = "001" AND ALU_result2LSB = "01" AND Control_signed
  = '0') else
      ("00000000000000000000000" & data_in (23 downto 16)) when
(control_instruction = "001" AND ALU_result2LSB = "10" AND Control_signed
  = '0') else
      ("00000000000000000000000" & data_in (31 downto 24)) when
(control_instruction = "001" AND ALU_result2LSB = "11" AND Control_signed
  = '0') else
      --LHU
      ("0000000000000000" & data_in (15 downto 0) ) when
(control_instruction = "010" AND ALU_result2LSB(1) = '0' AND
Control_signed = '0') else
      ("0000000000000000" & data_in (31 downto 16)) when
(control_instruction = "010" AND ALU_result2LSB(1) = '1' AND
Control_signed = '0') else
      --LW
      data_in when control_instruction = "000" else --esta linea esta
incluida en el when others pero esto me parece mas claro.
      data_in ;
  LB1 <= (others => data_in(7));


  LB2 <= (others => data_in(15));

  LB3 <= (others => data_in(23));

  LB4 <= (others => data_in(31));

  LH1 <= (others => data_in(15));

  LH2 <= (others => data_in(31));

  data_out <= data_aux;

end   MEM_Load_module_arc;
```

1.4.5 MEM_Mux_8

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity MEM_Mux8 is
    port (
    data0   : in  std_logic_vector (31 downto 0);
```

```
    data1   : in  std_logic_vector (31 downto 0);
    data2   : in  std_logic_vector (31 downto 0);
    data3   : in  std_logic_vector (31 downto 0);
    data4   : in  std_logic_vector (31 downto 0);
    data5   : in  std_logic_vector (31 downto 0);
    data6   : in  std_logic_vector (31 downto 0);
    data7   : in  std_logic_vector (31 downto 0);
    d_out   : out std_logic_vector (31 downto 0);
    sel     : in  std_logic_vector (2  downto 0)
    );
end MEM_Mux8;

architecture MEM_Mux8_arc of MEM_Mux8 is

  signal data_aux : std_logic_vector (31 downto 0);

begin

  d_out <= data_aux;

  with sel select data_aux <=
    data0  when "000",
    data1  when "001",
    data2  when "010",
    data3  when "011",
    data4  when "100",
    data5  when "101",
    data6  when "110",
    data7  when "111",
    x"00000000" when others;

end  MEM_Mux8_arc;
```

1.4.6 MEM_Store_Module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity MEM_Store_module is
  port (
    ALU_result5LSB : in  std_logic_vector (4 downto 0); --define la
ALU_result5LSB
    control_store  : in  std_logic_vector (2  downto 0); --
LB,LW,LH,SB,SH,SW,Nada. 000 001 010 011 100 101 110
    data_in        : in  std_logic_vector (31 downto 0);-- el dato a
escribir en memoria
    data_out       : out std_logic_vector (31 downto 0);-- le hacemos una
mascara al dato para que lo escriba bien
```

```
    enable_vector : out std_logic_vector (31 downto 0) --Enable de cada
una de las partes de la memoria
  ) ;
end MEM_Store_module;

architecture MEM_Store_module_arc of MEM_Store_module is

  signal data_aux : std_logic_vector (31 downto 0);
  signal ALU_result2LSB : std_logic_vector (1 downto 0);
  signal Byte : std_logic_vector (31 downto 0);
  signal Half : std_logic_vector (31 downto 0);
  signal Word : std_logic_vector (31 downto 0);
begin

  Byte <= "00000000000000000000000000000001";
  Half <= "00000000000000000000000000000011";
  Word <= "00000000000000000000000000001111";

  ALU_result2LSB <= ALU_result5LSB (1 downto 0);

  enable_vector <=

  std_logic_vector(shift_left(unsigned(Byte),to_integer(unsigned(ALU_resu
lt5LSB)))) when (control_store = "100") else --SB

  std_logic_vector(shift_left(unsigned(Half),to_integer(unsigned(ALU_resu
lt5LSB)))) when (control_store = "101") else --SH

  std_logic_vector(shift_left(unsigned(Word),to_integer(unsigned(ALU_resu
lt5LSB)))) when (control_store = "110") else --SW
    x"00000000";
  data_aux <=
      --SB
      "00000000" & "00000000" & "00000000" & data_in (7 downto 0) when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "00") else
      "00000000" & "00000000" & data_in (7 downto 0) & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "01") else
      "00000000" & data_in (7 downto 0) & "00000000" & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "10") else
      data_in (7 downto 0) & "00000000" & "00000000" & "00000000" when
(control_store = "100" AND ALU_result2LSB (1 downto 0) = "11") else
      --SH
      "00000000" & "00000000" & data_in (15 downto 0) when (control_store
= "101" AND ALU_result2LSB(1) = '0') else
      data_in (15 downto 0) & "00000000" & "00000000" when (control_store
= "101" AND ALU_result2LSB(1) = '1') else
      --SW
      data_in when (control_store = "110") else
      "00000000000000000000000000000000" ;
  data_out <= data_aux;

end  MEM_Store_module_arc;
```

1.4.7 MEM_Write_Back_select.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity MEM_Write_Back_select is --WB solo escribe en registros, en pc se
escribe desde EXE con jump control
  port (
    Memory_data     : in  std_logic_vector (31 downto 0);
    PC_imm          : in  std_logic_vector (31 downto 0);
    ALU_Data        : in  std_logic_vector (31 downto 0);
    PC_4            : in  std_logic_vector (31 downto 0);
    control_WB_sel  : in  std_logic_vector (1  downto 0); -- 00 pc+4, 01
ALU, 10 MEM, 11 PC+imm.
    WB_data_out     : out std_logic_vector (31 downto 0)
  ) ;
end MEM_Write_Back_select;

architecture MEM_Write_Back_select_arc of MEM_Write_Back_select is

begin

  WB_data_out <=
    --PC+4 JAL JALR
    PC_4 when control_WB_sel = "00" else
    --ALU
    ALU_Data when control_WB_sel = "01" else
    --MEM
    Memory_data when control_WB_sel = "10" else
    --PC
    PC_imm when control_WB_sel = "11";

end  MEM_Write_Back_select_arc;
```

1.5 WRITE_BACK.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity Write_Back is
  port (
    control_WB_sel  : in  std_logic_vector (1 downto 0);--multiplexa el
dato de salida entre (nada alu mem pc+imm)
    control_WB_wren : in  std_logic; -- marca si hay que escribir en rd
```

```
    wb_address_in    : in  std_logic_vector (4 downto 0); --direccion del
registro a escribir. 5 bits para 32 registros.
    wb_address_out   : out std_logic_vector (4 downto 0); --salida
multiplexada de wb_address_in y x0.
    wb_data_in       : in  std_logic_vector (31 downto 0);
    wb_data_out      : out std_logic_vector (31 downto 0) --salida del
dato seleccionado entre alu mem pc y X
  ) ;
end Write_Back;

architecture Write_Back_arc of Write_Back is

  signal wb_data_aux : std_logic_vector (31 downto 0); --une los wb_data

  component WB_Write_Back_select
  port (
    Memory_data      : in  std_logic_vector (31 downto 0);
    PC_imm           : in  std_logic_vector (31 downto 0);
    ALU_Data         : in  std_logic_vector (31 downto 0);
    PC_4             : in  std_logic_vector (31 downto 0);
    control_WB_sel   : in  std_logic_vector (1  downto 0); -- 00 PC+4, 01
ALU, 10 MEM, 11 PC.
    WB_data_sel_out : out std_logic_vector (31 downto 0)
  ) ;
  end component;

  component wb_address_module
    port (
      control_WB_enable : in  std_logic;
      address           : in  std_logic_vector (4 downto 0);
      address_out       : out std_logic_vector (4 downto 0)
        );
  end component;

  begin


    address_select : wb_address_module
      port map(
          control_WB_enable => control_WB_wren,
          address           => wb_address_in,
          address_out       => wb_address_out
        );

    wb_data_aux  <= wb_data_in;
    wb_data_out <= wb_data_aux;

end  Write_Back_arc;
```

1.5.1 WB_address_module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity  WB_address_module is
port (
control_WB_enable : in  std_logic;
address           : in  std_logic_vector (4 downto 0);
address_out       : out std_logic_vector (4 downto 0)
);
end   WB_address_module;

architecture WB_address_module_arc of   WB_address_module is

signal address_aux : std_logic_vector (4 downto 0);

begin

address_out <= address_aux;
address_aux <=
address when control_WB_enable = '1' else
"00000";

end   WB_address_module_arc;
```

1.6 Forward_EXE

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity Forward_EXE is
  port (
     w_e_MEM      : in  std_logic;
     w_e_WB       : in  std_logic;
     address1_EXE : in  std_logic_vector (4 downto 0);
     address2_EXE : in  std_logic_vector (4 downto 0);
     address_MEM  : in  std_logic_vector (4 downto 0);
     address_WB   : in  std_logic_vector (4 downto 0);
     address_ID   : in  std_logic_vector (4 downto 0);
     forward1     : out std_logic_vector (1 downto 0);
     forward2     : out std_logic_vector (1 downto 0)
  ) ;
end Forward_EXE;

architecture Forward_EXE_arc of Forward_EXE is
```

```
  signal w_e_ID : std_logic;

begin



    w_e_ID <=
      '0' when address_ID = 0 else
      '1';

  forward1 <=
    "01" when w_e_MEM = '1' and address_MEM = address1_EXE else
    "10" when w_e_WB  = '1' and address_WB  = address1_EXE else
    "11" when w_e_ID  = '1' and address_ID  = address1_EXE else
    "00";

  forward2 <=
    "01" when w_e_MEM = '1' and address_MEM = address2_EXE else
    "10" when w_e_WB  = '1' and address_WB  = address2_EXE else
    "11" when w_e_ID  = '1' and address_ID  = address2_EXE else
    "00";

end  Forward_EXE_arc;
```

1.7 ID_Register_Copy

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity ID_Register_Copy is
  port (
    clock          : in  std_logic;
    reset_n        : in  std_logic;
    id_data_in     : in  std_logic_vector (31 downto 0);
    id_data_out    : out std_logic_vector (31 downto 0);
    id_address_in  : in  std_logic_vector ( 4 downto 0);
    id_address_out : out std_logic_vector ( 4 downto 0)
    );
end ID_Register_Copy;

architecture ID_Register_Copy_arc of ID_Register_Copy is

  signal id_data    : std_logic_vector (31 downto 0);
  signal id_address : std_logic_vector (4 downto 0);


begin
```

```
    process(clock, reset_n)
        begin
            if reset_n = '1' then
                id_data_out <= "00000000000000000000000000000000";
              id_address_out <= "00000";
            elsif rising_edge(clock) then
              id_data_out <= id_data_in;
              id_address_out <= id_address_in;

            end if;
      end process;


end  ID_Register_Copy_arc;
```

1.8 REG1_IF_ID

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity REG1_IF_ID is
  port (
    clock           : in  std_logic;
    clear           : in  std_logic;
    pc_in           : in  std_logic_vector (31 downto 0);
    --instruction_in  : in  std_logic_vector (31 downto 0);
    pc_out          : out std_logic_vector (31 downto 0)
    --instruction_out : out std_logic_vector (31 downto 0)
    ) ;
end REG1_IF_ID;

architecture REG1_IF_ID_arc of REG1_IF_ID is

  signal pc : std_logic_vector(31 downto 0);
  --signal instruction : std_logic_vector (31 downto 0);

  begin
    process(clock, clear)
        begin
            if clear = '1' then
                --instruction_out <= "00000000000000000000000000000000";
                pc_out <= "00000000000000000000000000000000";
            elsif rising_edge(clock) then
               --instruction_out <= instruction;
                    pc_out <= pc;
            end if;
      end process;
```

```
      pc <= pc_in;
      --instruction <= instruction_in;

end  REG1_IF_ID_arc;
```

1.9 REG2_ID_EXE

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity REG2_ID_EXE is
  port (
    clock           : in  std_logic;
    clear           : in  std_logic;
    pc_in           : in  std_logic_vector (31 downto 0);
    pc_out          : out std_logic_vector (31 downto 0);
    control_branch_in : in std_logic_vector (1 downto 0);
    control_branch_out : out std_logic_vector (1 downto 0);
    imm_in : in std_logic_vector (31 downto 0);
    imm_out : out std_logic_vector (31 downto 0);
    pc_4_in : in std_logic_vector (31 downto 0);
    pc_4_out : out std_logic_vector (31 downto 0);
    control_mux_in : in std_logic;
    control_mux_out : out std_logic;
    control_LUI_in : in std_logic;
    control_LUI_out : out std_logic;
    control_ALU_in : in std_logic_vector (3 downto 0);
    control_ALU_out : out std_logic_vector (3 downto 0);
    control_MEM_in : in std_logic_vector (3 downto 0);
    control_MEM_out : out std_logic_vector (3 downto 0);
    control_wb_sel_in : in std_logic_vector (1 downto 0);
    control_wb_sel_out : out std_logic_vector (1 downto 0);
    control_wb_en_in : in std_logic;
    control_wb_en_out : out std_logic;
    control_wb_address_in : in std_logic_vector (4 downto 0);
    control_wb_address_out : out std_logic_vector (4 downto 0);
    data1_in : in std_logic_vector (31 downto 0);
    data1_out : out std_logic_vector (31 downto 0);
    data2_in : in std_logic_vector (31 downto 0);
    data2_out : out std_logic_vector (31 downto 0);
    rs1_address_in  : in  std_logic_vector (4 downto 0);
    rs2_address_in  : in  std_logic_vector (4 downto 0);
    rs1_address_out : out std_logic_vector (4 downto 0);
    rs2_address_out : out std_logic_vector (4 downto 0)
    ) ;
end REG2_ID_EXE;

architecture REG2_ID_EXE_arc of REG2_ID_EXE is
```

```
signal pc : std_logic_vector(31 downto 0);
signal control_branch : std_logic_vector(1 downto 0);
signal imm : std_logic_vector(31 downto 0);
signal pc_4 : std_logic_vector(31 downto 0);
signal control_mux : std_logic;
signal control_LUI : std_logic;
signal control_ALU : std_logic_vector(3 downto 0);
signal control_MEM : std_logic_vector (3 downto 0);
signal control_wb_sel : std_logic_vector(1 downto 0);
signal control_wb_en : std_logic;
signal control_wb_address : std_logic_vector(4 downto 0);
signal data1 : std_logic_vector (31 downto 0);
signal data2 : std_logic_vector (31 downto 0);
signal rs1_address : std_logic_vector (4 downto 0);
signal rs2_address : std_logic_vector (4 downto 0);


begin
  process(clock, clear)
      begin
          if clear = '1' then
              pc_out <= "00000000000000000000000000000000";
            control_branch_out <= "00";
            imm_out <= "00000000000000000000000000000000";
            pc_4_out <= "00000000000000000000000000000000";
            control_mux_out <= '0';
            control_LUI_out <= '0';
            control_ALU_out <= "0000";
            control_MEM_out <= "0000";
            control_wb_sel_out <= "00";
            control_wb_en_out <= '0';
            control_wb_address_out <= "00000";
            data1_out <= x"00000000";
            data2_out <= x"00000000";
            rs1_address_out <= "00000";
            rs2_address_out <= "00000";
          elsif rising_edge(clock) then
            pc_out <= pc;
            control_branch_out <= control_branch;
            imm_out <= imm;
            pc_4_out <= pc_4;
            control_mux_out <= control_mux;
            control_LUI_out <= control_LUI;
            control_ALU_out <= control_ALU;
            control_MEM_out <= control_MEM;
            control_wb_sel_out <= control_wb_sel;
            control_wb_en_out <= control_wb_en;
            control_wb_address_out <= control_wb_address;
            data1_out <= data1;
            data2_out <= data2;
            rs1_address_out <= rs1_address;
```

```
            rs2_address_out <= rs2_address;

        end if;
    end process;

    pc <= pc_in;
        control_branch <= control_branch_in;
        imm <= imm_in;
        pc_4 <= pc_4_in;
        control_mux <= control_mux_in;
        control_LUI <= control_LUI_in;
        control_ALU <= control_ALU_in;
        control_MEM <= control_MEM_in;
        control_wb_sel <= control_wb_sel_in;
        control_wb_en <= control_wb_en_in;
        control_wb_address <= control_wb_address_in;
        data1 <= data1_in;
        data2 <= data2_in;
        rs1_address <= rs1_address_in;
        rs2_address <= rs2_address_in;


end  REG2_ID_EXE_arc;
```

1.10 REG3_EXE_MEM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;        -- for addition & counting
use ieee.numeric_std.all;               -- for type conversions

entity REG3_EXE_MEM is -- los _out son los guardados y leidos; los in son
la entrada a escribir en un flanco
  port (
    clock           : in  std_logic;
    clear           : in  std_logic;
    pc_imm_in           : in  std_logic_vector (31 downto 0);
    pc_imm_out          : out std_logic_vector (31 downto 0);
    ALU_res_in : in std_logic_vector (31 downto 0);
    ALU_res_out : out std_logic_vector (31 downto 0);
    pc_4_in : in std_logic_vector (31 downto 0);
    pc_4_out : out std_logic_vector (31 downto 0);
    rs2_in : in std_logic_vector (31 downto 0);
    rs2_out : out std_logic_vector (31 downto 0);
    control_MEM_in : in std_logic_vector (3 downto 0);
    control_MEM_out : out std_logic_vector (3 downto 0);
    control_wb_sel_in : in std_logic_vector (1 downto 0);
    control_wb_sel_out : out std_logic_vector (1 downto 0);
    control_wb_en_in : in std_logic;
```

```
    control_wb_en_out : out std_logic;
    control_wb_address_in : in std_logic_vector (4 downto 0);
    control_wb_address_out : out std_logic_vector (4 downto 0)
    ) ;
end REG3_EXE_MEM;

architecture REG3_EXE_MEM_arc of REG3_EXE_MEM is

  signal pc_imm : std_logic_vector(31 downto 0);
  signal pc_4 : std_logic_vector(31 downto 0);
  signal control_MEM : std_logic_vector (3 downto 0);
  signal control_wb_sel : std_logic_vector(1 downto 0);
  signal control_wb_en : std_logic;
  signal control_wb_address : std_logic_vector(4 downto 0);
  signal ALU_res : std_logic_vector (31 downto 0);
  signal rs2 : std_logic_vector (31 downto 0);


  begin
    process(clock, clear)
        begin
            if clear = '1' then
              rs2_out <= "00000000000000000000000000000000";
              ALU_res_out <= "00000000000000000000000000000000";
                 pc_imm_out <= "00000000000000000000000000000000";
              pc_4_out <= "00000000000000000000000000000000";
              control_MEM_out <= "0000";
              control_wb_sel_out <= "00";
              control_wb_en_out <= '0';
              control_wb_address_out <= "00000";
            elsif rising_edge(clock) then
              pc_imm_out <= pc_imm;
              pc_4_out <= pc_4;
              control_MEM_out <= control_MEM;
              control_wb_sel_out <= control_wb_sel;
              control_wb_en_out <= control_wb_en;
              control_wb_address_out <= control_wb_address;
              ALU_res_out <= ALU_res;
              rs2_out <= rs2;

            end if;
      end process;

      pc_imm <= pc_imm_in;
      pc_4 <= pc_4_in;
      control_MEM <= control_MEM_in;
      control_wb_sel <= control_wb_sel_in;
      control_wb_en <= control_wb_en_in;
      control_wb_address <= control_wb_address_in;
      rs2 <= rs2_in;
      ALU_res <= ALU_res_in;
```

```
end  REG3_EXE_MEM_arc;
```

1.11 REG4_MEM_WB

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;         -- for addition & counting
use ieee.numeric_std.all;                -- for type conversions

entity REG4_MEM_WB is
port (
clock           : in  std_logic;
clear           : in  std_logic;
pc_imm_in           : in  std_logic_vector (31 downto 0);
pc_imm_out          : out std_logic_vector (31 downto 0);
ALU_res_in : in std_logic_vector (31 downto 0);
ALU_res_out : out std_logic_vector (31 downto 0);
pc_4_in : in std_logic_vector (31 downto 0);
pc_4_out : out std_logic_vector (31 downto 0);
control_wb_sel_in : in std_logic_vector (1 downto 0);
control_wb_sel_out : out std_logic_vector (1 downto 0);
control_wb_en_in : in std_logic;
control_wb_en_out : out std_logic;
control_wb_address_in : in std_logic_vector (4 downto 0);
control_wb_address_out : out std_logic_vector (4 downto 0);
MEM_data_in : in std_logic_vector (31 downto 0);
MEM_data_out : out std_logic_vector (31 downto 0);
load_control_in: in std_logic_vector (3 downto 0);
load_control_out: out std_logic_vector(3 downto 0)
) ;
end REG4_MEM_WB;

architecture REG4_MEM_WB_arc of REG4_MEM_WB is

signal pc_imm : std_logic_vector(31 downto 0);
signal pc_4 : std_logic_vector(31 downto 0);
signal control_wb_sel : std_logic_vector(1 downto 0);
signal control_wb_en : std_logic;
signal control_wb_address : std_logic_vector(4 downto 0);
signal ALU_res : std_logic_vector (31 downto 0);
signal MEM_data : std_logic_vector(31 downto 0);
signal load_control : std_logic_vector(3 downto 0);


begin
process(clock, clear)
begin
if clear = '1' then
```

```
ALU_res_out <= "00000000000000000000000000000000";
pc_imm_out <= "00000000000000000000000000000000";
pc_4_out <= "00000000000000000000000000000000";
control_wb_sel_out <= "00";
control_wb_en_out <= '0';
control_wb_address_out <= "00000";
MEM_data_out <= "00000000000000000000000000000000";
load_control_out <= "0000";
elsif rising_edge(clock) then
pc_imm_out <= pc_imm;
pc_4_out <= pc_4;
control_wb_sel_out <= control_wb_sel;
control_wb_en_out <= control_wb_en;
control_wb_address_out <= control_wb_address;
ALU_res_out <= ALU_res;
MEM_data_out <= MEM_data;
load_control_out <= load_control;

end if;
end process;

pc_imm <= pc_imm_in;
pc_4 <= pc_4_in;
control_wb_sel <= control_wb_sel_in;
control_wb_en <= control_wb_en_in;
control_wb_address <= control_wb_address_in;
ALU_res <= ALU_res_in;
MEM_data <= MEM_data_in;
load_control <= load_control_in;



end  REG4_MEM_WB_arc;
```

2 *SIMULACIÓN*

2.3.1 RISCV32I_VHD_TST.VHT

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY RISCV32I_vhd_tst IS
END RISCV32I_vhd_tst;
ARCHITECTURE RISCV32I_arch OF RISCV32I_vhd_tst IS
-- constants
-- signals
SIGNAL clock : STD_LOGIC := '0';
SIGNAL reset_n : STD_LOGIC;
```

```
COMPONENT RISCV32I
        PORT (
        clock : IN STD_LOGIC;
        reset_n : IN STD_LOGIC
        );
END COMPONENT;
BEGIN
        i1 : RISCV32I
        PORT MAP (
-- list connections between master ports and signals
        clock => clock,
        reset_n => reset_n
        );
init : PROCESS
BEGIN
WAIT;
END PROCESS init;
clock <= not clock after 10 ns;
always : PROCESS
BEGIN

reset_n <= '1';
wait for 15 ns;
reset_n <= '0';
wait for 3000 ns;
assert false severity failure;
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
END RISCV32I_arch;
```

2.3.2 R_I.ASM

```
addi x1,x0,10
addi x2,x0,-20
addi x3,x0,15
addi x4,x0,240
lui x5,1
auipc x6,1
slti x7,x1,11
slti x8,x1,9
sltiu x9,x1,11
sltiu x10,x1,9
sltiu x11,x1,-1
xori x12,x1,8
ori x13,x1,5
andi x14,x1,31
slli x15,x1,1
srli x16,x1,1
```

```
srai x17,x1,1
srai x18,x2,1
add x19,x1,x2
sub x20,x1,x2
sub x21,x2,x1
sll x22,x2,x1
slt x23,x1,x2
slt x24,x2,x1
sltu x25,x1,x2
sltu x26,x2,x1
xor x27,x1,x2
srl x28,x2,x1
sra x29,x2,x1
or x30,x1,x2
and x31,x1,x2
```

2.3.3 BRANCH.ASM

```
    addi x1,x0,10
    addi x2,x0,20
    addi x31,x0,-1
    beq x1,x2,B1
    addi x3,x0,1
    addi x3,x0,2
    addi x3,x0,3
    addi x3,x0,4
    addi x3,x0,5
B1: beq x1,x1,B2
    addi x4,x0,1
    addi x4,x0,2
    addi x4,x0,3
    addi x4,x0,4
    addi x4,x0,5
B2: bne x1,x1,B3
    addi x5,x0,1
    addi x5,x0,2
    addi x5,x0,3
B3 : bne x1,x2,B4
    addi x6,x0,1
    addi x6,x0,2
    addi x6,x0,3
B4: blt x2,x1,B5
    addi x7,x0,1
    addi x7,x0,2
    addi x7,x0,3
B5 : blt x1,x2,B6
    addi x8,x0,1
    addi x8,x0,2
    addi x8,x0,3
```

```
B6:bge x1,x2,B7
    addi x9,x0,1
    addi x9,x0,2
    addi x9,x0,3
B7 : bge x2,x1,B8
    addi x10,x0,1
    addi x10,x0,2
    addi x10,x0,3
B8: bgeu x1,x31,B9
    addi x11,x0,1
    addi x11,x0,2
    addi x11,x0,3
B9 : bgeu x31,x1,B10
    addi x12,x0,1
    addi x12,x0,2
    addi x12,x0,3
B10: bltu x31,x1,B11
    addi x13,x0,1
    addi x13,x0,2
    addi x13,x0,3
B11 :bltu x1,x31,B12
    addi x14,x0,1
    addi x14,x0,2
    addi x14,x0,3
B12: bgeu x1,x31,B13
    addi x15,x0,1
    addi x15,x0,2
    addi x15,x0,3
B13 :   bgeu x1,x1,B14
    addi x16,x0,1
    addi x16,x0,2
    addi x16,x0,3
B14 : bgeu x1,x1,B15
    addi x17,x0,1
    addi x17,x0,2
    addi x17,x0,3
B15: bgeu x1,x1,B16
    addi x18,x0,2
    addi x18,x0,3
    addi x18,x0,4
B16 : addi x19,x0,1
    addi x19,x0,2
    addi x19,x0,3
    addi x19,x0,4
    addi x20,x0,1
    addi x20,x0,2
    addi x20,x0,3
    addi x20,x0,4
```

2.3.4 JAL.ASM

```
      addi x1,x0,16
      addi x0,x0,0
      jalr x2,x1,16
      addi x3,x0,1
      addi x3,x0,2
      addi x3,x0,3
      jal  x4,JAL1
       addi x5,x0,1
JAL1: addi x5,x0,2
      addi x5,x0,3
      addi x5,x0,4
```

2.3.5 LOADSTORE.ASM

```
addi x1,x0,128
addi x2,x0,-1
sb x1,(x0)
sb x1,1(x0)
sb x1,2(x0)
sb x1,3(x0)
sb x2,4(x0)
sb x2,5(x0)
sb x2,6(x0)
sb x2,7(x0)
sh x1,8(x0)
sh x1,10(x0)
sw x1,12(x0)
addi x0,x0,0
addi x0,x0,0
addi x0,x0,0
lw x1,0
lw x4,4
lw x5,8
lw x6,12
lb x7,0
lb x8,4
lbu x9,4
lh x10,4
lhu x10,4
```

ANEXO II. ALINEACIÓN CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE

Los objetivos de desarrollo sostenible son 17 metas fijadas por la ONU en 2015 que se consideran de máxima importancia a nivel global. El plazo fijado para cumplirlos es de 15 años y cualquier ingeniero debería tenerlos en mente durante la investigación y desarrollo de nuevas tecnologías.

Este proyecto se alinea con varios objetivos de desarrollo sostenible simultáneamente. El ODS con una relación más evidente es el objetivo 9: "Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación". Está en la propia naturaleza de RISC-V el promover la innovación y la industria sostenible, ya que su naturaleza de código abierto compatible con FPGA permite que la cantidad de personas capaz de innovar y generar nuevas tecnologías no se limite a aquellas capaces de pagar derechos de patente.

El ser implementable en una FPGA permite que el microprocesador sea completamente reutilizable ya que las FPGA pueden ser reprogramadas a voluntad evitando así el gasto de materias primas no renovables y difíciles de reciclar.

Por esto mismo se alinea también con el objetivo 12 "Producción y consumo responsables" ya que al hacer posible que se puedan implementar en FPGA diferentes tipos de hardware se reduce la necesidad de crear circuitos integrados convencionales ahorrando materiales innecesarios.

Otro objetivo muy importante con el que este proyecto está relacionado es el 17: "Revitalizar la Alianza Mundial para el Desarrollo Sostenible"

Al tratarse de un proyecto basado en código open source y cuyo objetivo es colaborar en el ecosistema de desarrollo de RISC-V, el proyecto se alinea con el objetivo 17 sobre la creación de "alianzas para lograr los objetivos". Esto se debe a que el uso compartido de hardware y software open source permite una colaboración mundial entre desarrolladores. Esto puede dotar a mucha gente de la capacidad de innovar generando soluciones más eficaces y eficientes para los problemas que la electrónica digital pretende solucionar.

Además, el proyecto ayuda a alcanzar otros objetivos como el 10 de "Reducción de las desigualdades", al permitir el acceso a hardware de manera gratuita volviéndolo más accesible a personas con escasos recursos económicos y permitiendo que estas tengan herramientas de trabajo de alta tecnología para poder generar riqueza.

Por último, acorde también a la filosofía de RISC-V se persigue el cumplimiento del objetivo 4 que busca garantizar "Educación de calidad", ya que la accesibilidad a microprocesadores y lógica de bajo coste es esencial para facilitar una educación enfocada a STEM accesible a todo el mundo.