Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Ingeniería en Electrónica Industrial y Automática

IMPLEMENTACIÓN DEL PROCESADOR RV32I EN EL PROGRAMA LOGISIM-EVOLUTION: APLICACIÓN A PRÁCTICAS DE LABORATORIO

Autor: Henar Parro Alonso
Tutor: Santiago Cáceres Gómez

Año 2024

Resumen

Este trabajo de fin de grado se centra en la implementación de un procesador basado en la arquitectura RISC-V, específicamente el conjunto de instrucciones RV32I, utilizando la herramienta de simulación Logisim-Evolution. El objetivo principal es implementar un procesador funcional que pueda ser utilizado en hardware real, como una FPGA, o en futuros proyectos que busquen ampliar la capacidad de instrucciones del procesador. A lo largo del documento, se detallan los pasos para la implementación de las instrucciones de tipo R, I, S, B, U y J, así como la integración de todos los componentes en un único sistema. La combinación de RISC-V y Logisim-Evolution ofrece una poderosa herramienta para la enseñanza de la arquitectura de computadores y el diseño de sistemas digitales que se ha aprovechado para desarrollar una serie de prácticas de laboratorio. Estos guían al estudiante a implementar un procesador desde cero abordando la arquitectura del procesador de manera progresiva. La implementación final del procesador permite la simulación de instrucciones básicas y avanzadas, proporcionando una visión integral del funcionamiento de los procesadores modernos basados en la arquitectura RISC-V.

Palabras clave: procesadores, RISC-V, circuitos digitales, instrucciones, laboratorio.

Abstract

This degree dissertation focuses on the implementation of a processor based on the RISC-V architecture, specifically the RV32I instruction set, using the Logisim-Evolution simulation tool. The main objective is to implement a functional processor that can be used in real hardware, such as an FPGA, or in future projects aimed at expanding the processor's instruction capacity. Throughout the document, the steps for implementing R, I, S, B, U, and J type instructions are detailed, as well as the integration of all components into a single system. The combination of RISC-V and Logisim-Evolution offers a powerful tool for teaching computer architecture and digital system design, which has been utilized to develop a series of laboratory practices. These guide students in implementing a processor from scratch, progressively addressing the processor architecture. The final implementation of the processor allows the simulation of both basic and advanced instructions, providing a comprehensive view of the operation of modern processors based on the RISC-V architecture.

Keywords: processors, RISC-V, digital circuits, instructions, laboratory.

Índice general

Introducción **VII**

1. RISC-V **1**
1.1. Orígenes y desarrollo de RISC-V 1
1.1.1. Arquitectura de código abierto 2
1.1.2. Niveles de interpretación 2
1.1.3. Características clave de RISC-V 3
1.1.4. Impacto y adopción 4
1.2. Registros de RISC-V 4
1.3. Extensiones y funcionalidades 6
1.3.1. RV32I: Instrucciones Base para Enteros 6

2. Herramientas **9**
2.1. Características del programa 9
2.2. Uso de la aplicación 11
2.2.1. Uso de los componentes de la biblioteca 12
2.2.2. Creación de un subcircuito 13
2.2.3. Creación de un circuito combinacional 13

3. Implementación de la arquitectura RV32I **17**
3.1. Elementos clave 18
3.1.1. Memoria de programa y contador de programa 18
3.1.1.1. Implementación en Logisim-Evolution 18
3.1.2. Unidad de control 19
3.1.2.1. Implementación en Logisim-Evolution 21
3.1.3. Generador de dato inmediato 22
3.1.3.1. Implementación en Logisim-Evolution 24
3.2. Instrucciones de tipo R 26
3.2.1. Arquitectura necesaria 26
3.2.2. Implementación en Logisim-Evolution 27
3.2.2.1. Banco de registros 27
3.2.2.2. Unidad Aritmético-Lógica (ALU) 30
3.3. Instrucciones de tipo I 32
3.3.1. Arquitectura necesaria 33
3.3.2. Implementación en Logisim-Evolution 33
3.3.2.1. Instrucciones con operaciones aritméticas, lógicas, de desplazamiento y de comparación 33

3.3.2.2. Instrucciones *load* . . . 34
3.3.2.3. Instrucción de salto incondicional . . . 37
3.4. Instrucciones de tipo S . . . 38
3.4.1. Arquitectura necesaria . . . 39
3.4.2. Implementación en Logisim-Evolution . . . 39
3.5. Instrucciones de tipo B . . . 43
3.5.1. Arquitectura necesaria . . . 43
3.5.2. Implementación en Logisim-Evolution . . . 44
3.5.2.1. Unidad Aritmético-Lógica (ALU) . . . 44
3.5.2.2. Circuito principal . . . 45
3.6. Instrucciones de tipo U . . . 46
3.6.1. Arquitectura necesaria . . . 46
3.6.2. Implementación en Logisim-Evolution . . . 46
3.7. Instrucciones de tipo J . . . 47
3.7.1. Arquitectura necesaria . . . 48
3.7.2. Implementación en Logisim-Evolution . . . 49

4. Comprobación de la implementación **51**
4.1. Comprobación de instrucciones que realizan operaciones básicas . . . 52
4.1.1. Realización y ejecución del programa de verificación . . . 52
4.2. Comprobación de instrucciones de carga y almacenamiento . . . 53
4.2.1. Realización y ejecución del programa de verificación . . . 53
4.3. Comprobación de saltos condicionales . . . 55
4.3.1. Realización y ejecución del programa de verificación . . . 55
4.4. Comprobación de saltos incondicionales . . . 57
4.4.1. Realización y ejecución del programa de verificación . . . 57
4.5. Comprobación de instrucciones de tipo U . . . 58
4.5.1. Realización y ejecución del programa de verificación . . . 58

5. Prácticas de laboratorio **59**
5.1. Laboratorio 1 . . . 60
5.1.1. Introducción al laboratorio . . . 60
5.1.2. Enunciado del laboratorio . . . 60
5.1.3. Preguntas a reflexionar sobre la práctica . . . 64
5.2. Laboratorio 2 . . . 65
5.2.1. Introducción al laboratorio . . . 65
5.2.2. Enunciado del laboratorio . . . 65
5.2.3. Preguntas a reflexionar sobre la práctica . . . 67
5.3. Laboratorio 3 . . . 67
5.3.1. Introducción al laboratorio . . . 67
5.3.2. Enunciado del laboratorio . . . 68
5.3.3. Preguntas a reflexionar sobre la práctica . . . 72
5.4. Laboratorio 4 . . . 73
5.4.1. Introducción al laboratorio . . . 73
5.4.2. Enunciado del laboratorio . . . 73
5.4.3. Preguntas a reflexionar sobre la práctica . . . 75
5.5. Laboratorio 5 . . . 75

5.5.1. Introducción al laboratorio 75
5.5.2. Enunciado del laboratorio 76
5.5.3. Preguntas a reflexionar sobre la práctica 78
5.6. Laboratorio 6 78
5.6.1. Introducción al laboratorio 78
5.6.2. Enunciado del laboratorio 79
5.6.3. Preguntas a reflexionar sobre la práctica 80

Conclusiones **81**

Bibliografía **82**

Introducción

Este trabajo se centra en la implementación de un procesador basado en la arquitectura RISC-V. El conjunto de instrucciones a implementar se corresponde con la extensión RV32I de la arquitectura RISC-V, que abarca las instrucciones base para enteros. Esta extensión es obligatoria para todas las implementaciones de RISC-V [1].

Contexto

Los procesadores RISC (*Reduced Instruction Set Computer*) representan un cambio significativo en el diseño de arquitecturas de instrucciones y su uso ha experimentado un notable auge en los últimos años. Una de las características más destacadas de la arquitectura RISC-V es que es de código abierto, lo que permite a cualquier persona utilizar o modificar la arquitectura sin costes de licencia, además de fomentar la innovación y la colaboración de la comunidad tecnológica.

A diferencia de las ISA comerciales existentes, que son propietarias y extremadamente complejas, RISC-V ofrece una alternativa más simple y modular. Las ISA comerciales, como MIPS y SPARC, presentan una complejidad significativa que dificulta su implementación completa en hardware, lo cual incrementa los costos y reduce la eficiencia. RISC-V, en cambio, se diseñó con la intención de ser una ISA sencilla evitando las complicaciones técnicas que afectan a sus predecesoras [2].

La modularidad de la arquitectura facilita la creación de diseños optimizados para aplicaciones específicas, desde dispositivos embebidos de bajo consumo hasta supercomputadoras de alto rendimiento [2]. Un ejemplo notable de esto es el Centro de Supercomputación de Barcelona (BSC), donde se están realizando investigaciones y desarrollos para integrar RISC-V en futuras generaciones de supercomputadoras [3].

Objetivos

En este proyecto se proponen dos objetivos principales ligados, respectivamente, a los ámbitos técnico y docente. Respecto al objetivo técnico, este trabajo persigue implementar un procesador funcional que pueda ser utilizado en hardware real, como una FPGA, o en futuros proyectos que busquen ampliar la capacidad de instrucciones del procesador. Por otro lado, respecto al objetivo docente, se enfoca en lograr la transferencia de conocimiento a los estudiantes y jóvenes investigadores, intentando anticiparse a las posibles dificultades que estos pudieran enfrentar durante su aprendizaje.

Para lograr el cumplimiento de estos objetivos, se define una lista de objetivos parciales:

- Aprender a manejar la herramienta mediante la cual se va a diseñar el procesador.
- Implementar el conjunto de instrucciones que componen la extensión RV32I de la arquitectura RISC-V y verificar el correcto funcionamiento del mismo.
- Comprender la funcionalidad y ejecución de cada una de las instrucciones implementadas, así como los componentes que emplean para ello.
- Elaborar una serie de laboratorios basados en las dificultades experimentadas durante la implementación. De este modo, los estudiantes pueden ser guiados de forma progresiva durante el desarrollo de las partes del procesador.
- Proponer ejercicios prácticos sobre los circuitos diseñados, mediante los cuales los estudiantes puedan realizar comprobaciones de su trabajo.

Estructura del proyecto

En el Capítulo 1 de este proyecto se desarrolla con detalle la historia de este procesador, las características que lo definen y los conocimientos técnicos necesarios para comprender la parte práctica a desarrollar. Además, se presenta el conjunto de instrucciones a implementar y sus respectivos formatos de instrucción, a los cuales se hará referencia repetidas veces a lo largo del desarrollo del trabajo.

Para la implementación de la arquitectura, se ha utilizado una aplicación de simulación y diseño de circuitos digitales conocida como Logisim-Evolution, la cual se introduce en el Capítulo 2. Esta es una herramienta educativa, gratuita y de código abierto que permite a los usuarios crear y probar sus diseños en un entorno virtual antes de implementarlos en hardware real. La facilidad de uso y la versatilidad de Logisim-Evolution lo convierten en una opción ideal para fines educativos y de investigación [4].

El desarrollo de la parte práctica del proyecto comienza por comprender a fondo cada uno de los tipos de instrucciones de la arquitectura, las operaciones que realizan y los componentes que emplean para su ejecución. En el Capítulo 3, inicialmente se explican y se diseñan los circuitos de uso común. Posteriormente, se pasa a realizar un estudio individual de cada tipo de instrucción, en el que se van implementando los componentes que forman la arquitectura de la extensión RV32I de forma progresiva, entendiendo cómo se conectan entre sí.

Tras la implementación de la arquitectura RV32I en el Capítulo 4, se realizan una serie de programas de prueba en los que se ejecutan cada una de las diferentes instrucciones implementadas. Este proceso de verificación es crucial para asegurar el correcto funcionamiento del circuito. Las pruebas incluyen la ejecución de instrucciones aritméticas, lógicas, de control de flujo y de acceso a la memoria, evaluando su comportamiento. La verificación exhaustiva garantiza que el diseño cumple con las especificaciones y

funciona de manera fiable.

A raíz de la implementación de la arquitectura, se elaboran una serie de laboratorios que se presentan en el Capítulo 5. Estos se desarrollan de forma progresiva y teniendo en cuenta las dificultades experimentadas en la implementación de la primera fase del proyecto. Cada práctica se enfoca en uno o varios componentes específicos de la arquitectura, permitiendo a los estudiantes abordar cada elemento de manera individual y comprender su funcionamiento y su interrelación con otros componentes. El objetivo final de estas prácticas es que los estudiantes sean capaces de integrar todos los circuitos en un único sistema, acercándose más a la arquitectura real del RISC-V. Estos laboratorios no solo refuerzan el conocimiento teórico, sino que también proporcionan experiencia práctica en el diseño y la implementación de sistemas digitales complejos.

Por último, este trabajo concluye con un capítulo de conclusiones en el cual se desarrollan los objetivos logrados a lo largo del desarrollo del proyecto así como las dificultades afrontadas. Además, se reflexiona sobre las lecciones aprendidas durante el proceso de implementación y se proponen posibles mejoras y futuras líneas de investigación. Se subraya también la relevancia de la educación en el ámbito de las arquitecturas abiertas, promoviendo un enfoque práctico.

Capítulo 1

RISC-V

En este capítulo se desarrolla la historia del RISC-V y las características principales que definen la arquitectura. También, se muestran las distintas extensiones en las que se divide el procesador y se introduce la extensión base para enteros, la cual se implementa en este proyecto.

1.1. Orígenes y desarrollo de RISC-V

RISC-V, pronunciado «risk-cinco», representa la quinta generación en el diseño RISC y constituye una arquitectura de conjunto de instrucciones (ISA, por sus siglas en inglés) [5]. El concepto de ISA es fundamental en el ámbito de la arquitectura informática, ya que actúa como una interfaz entre el software y el hardware. Esta interfaz define la manera en que un ordenador organiza, representa y ejecuta las instrucciones.

El ISA RISC-V es de código abierto y surgió de un proyecto de investigación en la Universidad de California, Berkeley. El proyecto comenzó en 2010 bajo la dirección de los profesores Krste Asanović y David Patterson, junto con los estudiantes de posgrado Yunsup Lee y Andrew Waterman [6]. El objetivo fue crear un nuevo ISA de código abierto que abordara las limitaciones de los ISA ya existentes, tanto propietarios como abiertos, y proporcionara una base para futuras innovaciones en procesadores.

La creación de una nueva arquitectura como RISC-V se debió a la necesidad de superar las restricciones impuestas por las arquitecturas propietarias como x86 y ARM. Una de las principales ventajas de RISC-V es su naturaleza de código abierto, lo que permite a cualquier persona utilizar, modificar y distribuir la arquitectura sin costos de licencia. Además, su diseño modular permite a los diseñadores de chips personalizar y adaptar la ISA a sus necesidades específicas mediante la inclusión o exclusión de extensiones. Esta flexibilidad y la simplicidad del conjunto de instrucciones reducidas (RISC) facilitan una mayor eficiencia energética y un diseño más sencillo, lo que resulta en procesadores más económicos y eficientes [2].

El primer chip RISC-V fue diseñado en 2011 y las especificaciones del ISA se hicieron públicas en los informes técnicos [5]. La Fundación RISC-V, ahora conocida como *RISC-V International*, se fundó en 2015 para construir una comunidad colaborativa

Figura 1.1: Miembros de la Fundación RISC-V International [7].

basada en la innovación en software y hardware. La fundación es una corporación sin fines de lucro controlada por sus miembros (Fig. 1.1) y dirige el desarrollo para impulsar la adopción inicial del ISA RISC-V [8].

1.1.1. Arquitectura de código abierto

Una arquitectura de código abierto, en el contexto del diseño de procesadores, se refiere a un ISA que está disponible públicamente para que cualquiera lo use, modifique y distribuya sin necesidad de tarifas de licencia o restricciones [9]. Este enfoque fomenta la colaboración e innovación, ya que permite que una comunidad global de desarrolladores contribuya y mejore el ISA. La naturaleza de código abierto de RISC-V implica que no está vinculado a ninguna empresa específica, país o tecnología propietaria, convirtiéndolo en un marco de referencia accesible y universalmente adoptable para el diseño de procesadores.

1.1.2. Niveles de interpretación

Es posible programar todas las instrucciones que es capaz de realizar RISC-V en un lenguaje de programación de alto nivel. El proceso de traducción de lenguajes de alto nivel a instrucciones que pueda ejecutar RISC-V implica varias capas de ejecución y compilación (Fig. 1.2). Existen herramientas para compilar los programas en C y generar el código ensamblador, integrándose algunas de ellas en IDEs como Eclipse.

El primer paso para traducir un lenguaje de alto nivel es compilar el programa en lenguaje ensamblador. Este es un lenguaje de bajo nivel que ofrece una representación simbólica de las instrucciones a ejecutar. Cada línea de código ensamblador representa una instrucción para el ordenador [10].

El siguiente paso es traducir el programa en lenguaje ensamblador a código máquina, que es una representación binaria de las instrucciones que el procesador puede

ejecutar directamente. Esta traducción la realiza un ensamblador. El código máquina de un procesador RISC-V es una secuencia de instrucciones binarias que corresponden directamente a las operaciones definidas en la ISA RISC-V [10].

Una vez que el programa está en forma de código máquina, puede cargarse en la memoria de RISC-V. Este se encarga de descodificar y ejecutar las instrucciones en los recursos de hardware que lo componen, como los registros, la unidad aritmética lógica (ALU), etc.

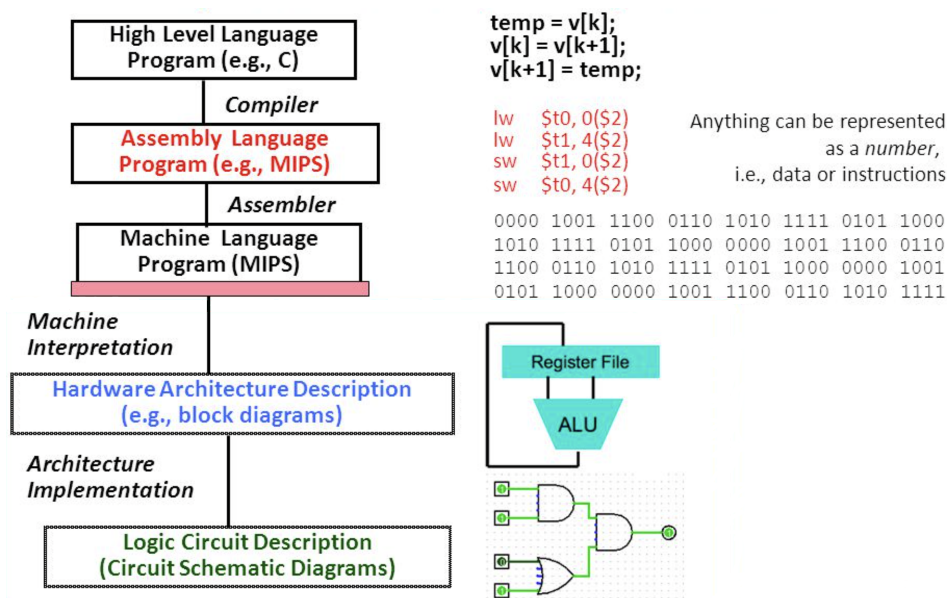


Figura 1.2: Niveles de interpretación.

En el nivel más bajo, para ejecutar las instrucciones se implementan circuitos lógicos. Estos circuitos están diseñados para realizar las operaciones fundamentales especificadas por la ISA RISC-V como sumas, restas, operaciones lógicas y acceso a memoria.

1.1.3. Características clave de RISC-V

Los ISA pueden ser desde muy sencillos, con un número reducido de instrucciones (RISC - *Reduced Instruction Set Computer*), hasta muy complejos, con un gran número de instrucciones y modos de direccionamiento (CISC - *Complex Instruction Set Computer*) [11]. RISC-V se basa en los principios establecidos de cómputo de un conjunto de instrucciones reducido, que enfatizan la simplicidad y eficiencia en el diseño de procesadores. Algunas de las características destacadas de RISC-V son las siguientes:

- Un ISA modular compuesto de una base obligatoria denominada RV32I junto con extensiones opcionales para multiplicar y dividir (RV32M), operaciones atómicas (RV32A), coma flotante de precisión simple (RV32F) y coma flotante de precisión doble (RV32D). La base y estas cuatro extensiones se denominan colectivamente RV32G [12]. Esto permite la personalización y extensibilidad, habilitando a los diseñadores de chips implementar solamente los componentes necesarios para sus aplicaciones específicas.

- Una arquitectura de carga-almacenamiento, donde sólo las instrucciones de carga (*load*) y almacenamiento (*store*) acceden a la memoria, mientras que el resto de instrucciones operan en pequeñas ubicaciones de almacenamiento dentro de la CPU que son accesibles a alta velocidad denominados registros. Esto reduce la complejidad del procesador y mejora su rendimiento [2].

- La totalidad del conjunto de instrucciones es de tamaño fijo para los diferentes formatos de 32, 64 y 128 bits, lo que simplifica la decodificación de las instrucciones y mejora la eficiencia del procesador. Sin embargo, el tamaño de dato predeterminado para cada uno de ellos es 32, 64 y 128 bits, respectivamente, lo que la hace adaptable a una amplia gama de aplicaciones, desde sistemas embebidos hasta servidores de alto rendimiento [13].

- No puede haber patentes relacionadas específicamente con el conjunto de instrucciones RISC-V, ya que no representa ninguna tecnología nueva, sino que se basa en ideas de arquitectura de computadoras que tienen al menos 40 años de antigüedad [6]. Además, una patente es un título de propiedad intelectual que otorga el Estado al inventor. Dado que RISC-V es de código abierto y de dominio público, su estructura básica no es susceptible de ser patentada. Sin embargo, las mejoras o tecnologías adicionales que las empresas creen para mejorar el rendimiento o la eficiencia de los procesadores basados en RISC-V sí pueden ser patentadas.[14]

1.1.4. Impacto y adopción

La naturaleza de código abierto de RISC-V ha llevado a su adopción en una amplia gama de aplicaciones, desde microcontroladores y sistemas integrados hasta cómputo de alto rendimiento e inteligencia artificial. Su ISA ha ido ganando popularidad académica y comercial debido a su potencial de personalización, flexibilidad y rentabilidad [9].

1.2. Registros de RISC-V

RISC-V tiene un total de 32 registros (Fig. 1.3). Para RV32 x tiene un tamaño de 32 bits, mientras que para RV64 la longitud de los datos que almacenan los registros y el PC es de 64 bits [1].

De los 32 registros, el primero, denominado *x0* mantiene un valor fijo de 0. Este registro simplifica ciertas operaciones y reduce el número de instrucciones de un programa, dando lugar a un código más compacto. Por ejemplo, si una operación necesita del valor cero, accede directamente a este registro sin necesidad de cargar un valor cero inmediato en un registro.

Este diseño de los registros difiere de otras arquitecturas, que para obtener el mismo resultado necesitan de alguna operación adicional en el programa.

El registro PC (*Program Counter*) es un registro adicional visible para el usuario que contiene la dirección de la siguiente instrucción a ejecutar. Se actualiza después de cada ciclo para apuntar a la siguiente instrucción de la secuencia, lo que garantiza el flujo

correcto de la ejecución del programa. Su valor solo puede ser modificado a través de algunas instrucciones, como los saltos condicionales o incondicionales.

El rendimiento eficiente de RISC-V se beneficia del uso de registros de propósito general para el almacenamiento temporal de variables, en lugar de utilizar la memoria, lo que agiliza las operaciones. Para gestionar estos valores, RISC-V emplea dos tipos de registros: los *temporary registers*, donde no se garantiza la preservación de los valores después de una llamada a función, y los *saved registers*, que mantienen los valores a lo largo de múltiples llamadas a funciones [15]. No obstante, para el propósito de este trabajo no se ahondará en la función y tipo de cada uno de los registros.

31 — 0	
`x0 / zero`	Alambrado a cero
`x1 / ra`	Dirección de retorno
`x2 / sp`	Stack pointer
`x3 / gp`	Global pointer
`x4 / tp`	Thread pointer
`x5 / t0`	Temporal
`x6 / t1`	Temporal
`x7 / t2`	Temporal
`x8 / s0 / fp`	Saved register, frame pointer
`x9 / s1`	Saved register
`x10 / a0`	Argumento de función, valor de retorno
`x11 / a1`	Argumento de función, valor de retorno
`x12 / a2`	Argumento de función
`x13 / a3`	Argumento de función
`x14 / a4`	Argumento de función
`x15 / a5`	Argumento de función
`x16 / a6`	Argumento de función
`x17 / a7`	Argumento de función
`x18 / s2`	Saved register
`x19 / s3`	Saved register
`x20 / s4`	Saved register
`x21 / s5`	Saved register
`x22 / s6`	Saved register
`x23 / s7`	Saved register
`x24 / s8`	Saved register
`x25 / s9`	Saved register
`x26 / s10`	Saved register
`x27 / s11`	Saved register
`x28 / t3`	Temporal
`x29 / t4`	Temporal
`x30 / t5`	Temporal
`x31 / t6`	Temporal
32	

31 — 0
`pc`
32

Figura 1.3: Registros del RV32I [1].

1.3. Extensiones y funcionalidades

RISC tiene dos variantes principales de base para enteros «I», RV32I y RV64I (RV128I todavía está en desarrollo)y son obligatorias para todas las implementaciones de RISC-V. Estas no se pueden modificar, pero sí ampliar con una o más extensiones opcionales. Las extensiones se dividen en dos grupos, las extensiones estándar y las no estándar. Las estándar proporcionan funcionalidades ligeramente diferentes como multiplicación/división de enteros, operaciones atómicas y aritméticas de coma flotante de precisión simple y doble [1].

La nomenclatura de subconjuntos ISA de RISC-V se define mediante un esquema estructurado. En él, se indica primero la longitud en bits que tendrán los valores a manejar por la arquitectura y después el carácter «I», que define el conjunto de instrucciones base para enteros, seguido de más caracteres que definen las extensiones opcionales que se quieran añadir. Por ejemplo, un RISC de 64 bits de longitud de valor a manejar y con la extensión estándar de multiplicación/división se define como RV64IM.

Todas las extensiones, con el conjunto de instrucciones que ofrece cada una, vienen definidas en el documento de especificaciones oficial [16]. Nosotros nos vamos a centrar únicamente en la variante de base para enteros de 32 bits.

1.3.1. RV32I: Instrucciones Base para Enteros

El ISA RV32I tiene seis formatos de instrucciones básicos (Fig. 1.4): tipo R para operaciones entre registros; tipo I para valores inmediatos, cargas en la memoria (*loads*) y saltos incondicionales en el programa; tipo S para almacenamiento (store); tipo B para saltos condicionales en la memoria de programa; tipo U para valores inmediatos largos; y tipo J para saltos incondicionales en la memoria de programa [1].

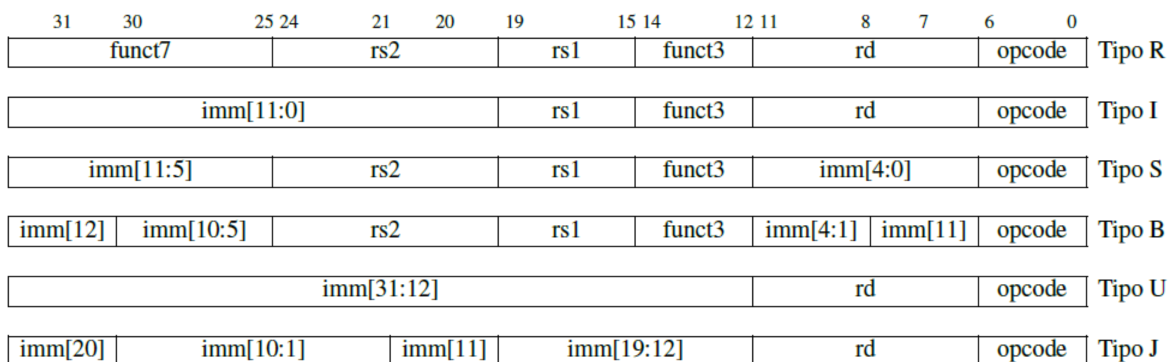


Figura 1.4: Formato de instrucciones RV32I [1].

Para simplificar la decodificación de las instrucciones, los registros de origen (*rs1* y *rs2*) y destino (*rd*) se mantienen, en todos los formatos que los utilizan durante la ejecución, en una posición fija. Además, en los formatos que utilizan inmediatos, el bit de signo está en el bit 31, lo que facilita y acelera el proceso de extensión de signo.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figura 1.5: Tabla con el formato de instrucción de la ISA RV32I [1].

En la Figura 1.5 se muestra desglosado el formato de todas las instrucciones que componen la ISA RV32I. En el software de aplicación que se presenta en el Capítulo 2 se van a implementar todas las instrucciones que aparecen a excepción del último bloque de instrucciones de tipo I mostradas debajo de las instrucciones de tipo R. Estas instrucciones específicas se utilizan para diversas operaciones de control y manejo de estados del procesador. Su implementación no es llevada a cabo en este proyecto debido a que están destinadas a tareas vinculadas con la implementación del sistema operativo.

Capítulo 2

Herramientas

En este capítulo se introduce la herramienta en la cual se va a desarrollar el proyecto y se muestran las características y herramientas que dispone. Además, se ofrece una introducción de cómo emplear cada una de ellas para lograr una mejor comprensión del proyecto desarrollado en el presente trabajo.

En el campo de la educación en electrónica digital, especialmente a un nivel avanzado, existen varias aplicaciones y programas que se utilizan para diseñar, simular y analizar sistemas y circuitos digitales.

Para la implementación y elaboración de laboratorios basados en el procesador, se ha optado por hacer uso del programa Logisim-Evolution. Este es una herramienta educativa para diseñar y simular circuitos lógicos digitales. Es una aplicación gratuita, de código abierto y multiplataforma. Ofrece una amplia gama de funciones que satisfacen tanto a principiantes como a usuarios avanzados, convirtiéndolo en una opción versátil para fines educativos [17].

Logisim-Evolution está disponible a través de varios gestores de paquetes y también ofrece las versiones más recientes del software, generadas automáticamente, conocidas como *nightly builds*. Estas son menos estables que las versiones oficiales, ya que contienen los cambios de código más recientes que aún no han sido probados exhaustivamente.

2.1. Características del programa

A continuación, se muestran algunas de las posibilidades que ofrece este software de aplicación y sus ventajas y debilidades frente a otros programas similares [18].

Código abierto y gratuito

Logisim-Evolution es una herramienta de código abierto y gratuita que facilita el acceso a estudiantes y educadores, contrastando con opciones como Proteus y Multisim, que son comerciales y requieren una inversión económica. LTSpice también es gratuito, pero no es de código abierto, lo que limita el potencial de contribuciones y modificaciones de la comunidad.

Diseño de circuitos

Logisim-Evolution proporciona una interfaz fácil de usar que simplifica el proceso de diseño de circuitos digitales. Su diseño intuitivo permite a los usuarios colocar de forma sencilla componentes y conectarlos entre sí, facilitando el proceso de aprendizaje tanto para estudiantes como para aficionados. Además, admite el diseño jerárquico, lo que permite construir circuitos complejos a partir de subcircuitos más sencillos.

Otros posibles softwares de aplicación, como Proteus, están más orientados al uso profesional e incluyen herramientas de diseño y simulación de PCB. Por otro lado, LTSpice se centra más en la simulación de circuitos analógicos, lo que limita su utilidad para circuitos digitales.

Simulación de circuitos lógicos

La función de simulación de circuitos es una característica fundamental que permite a los usuarios probar y verificar la funcionalidad de sus diseños en un entorno virtual antes de implementarlos en hardware real. Entre las capacidades de simulación se incluye probar varias combinaciones de entradas y observar la salida, lo cual es crucial para comprender y corregir errores o problemas en el diseño de circuitos.

Cronograma

La función de cronograma permite a los usuarios ver la evolución de las señales dentro de su circuito a lo largo del tiempo. Esto es particularmente útil para analizar el comportamiento de circuitos secuenciales y comprender cómo cambian las señales en respuesta a diferentes entradas.

Sin embargo, aunque Logisim-Evolution es una herramienta valiosa para la enseñanza y el aprendizaje del diseño de circuitos lógicos digitales, sus capacidades de cronograma pueden tener limitaciones en comparación con otras herramientas de simulación más avanzadas. Estas limitaciones incluyen problemas con la representación precisa del estado inicial de los circuitos y dificultades para adaptarse a cambios en la configuración del circuito lo que afecta la visualización de los retrasos en los cronogramas.

Componentes VHDL e integración de placas electrónicas

VHDL es un lenguaje de programación utilizado para describir la estructura y comportamiento de sistemas electrónicos digitales. Las siglas de la palabra resultante están compuestas por la combinación de los términos ***VHSIC*** ***H****ardware* ***D****escription* ***L****anguage* (Lenguaje de Descripción de Hardware para Circuitos Integrados de Muy Alta Velocidad). Se utiliza para el diseño, simulación y síntesis de circuitos integrados y sistemas electrónicos en FPGA o ASIC.

Una FPGA (*Field-Programmable Gate Array*) es un dispositivo semiconductor que contiene una matriz de bloques lógicos configurables y una red de interconexiones programables que pueden ser configuradas o “programadas” para realizar una amplia variedad de funciones lógicas. Esto significa que una FPGA puede ser reprogramada después de su fabricación para implementar cualquier diseño lógico, lo que ofrece una gran flexibilidad. Por otro lado, un ASIC (*Application-Specific Integrated Circuit*) es un

circuito integrado diseñado para una aplicación específica y, a diferencia de las FPGAs, no pueden ser reprogramados.

Para usuarios avanzados, Logisim-Evolution ofrece la capacidad de definir el comportamiento de los componentes utilizando VHDL. Esto abre posibilidades para crear componentes más complejos y personalizables. Además, admite la integración de placas electrónicas, lo que significa que los esquemas diseñados en el software pueden simularse en hardware real, en el caso de nuestro programa en FPGAs. Esta característica reduce la brecha entre el diseño teórico y la implementación práctica, proporcionando una experiencia práctica de aprendizaje.

Para la simulación de circuitos VHDL, Logisim-Evolution requiere una aplicación externa como Questa Advanced Simulator. Questa Advanced Simulator es necesario para validar y simular entidades VHDL, lo que permite a los usuarios verificar el comportamiento de sus diseños antes de implementarlos en hardware. Por otro lado, para la implementación en una FPGA, es necesario utilizar aplicaciones específicas de cada fabricante de FPGA. Por ejemplo, para las FPGAs de Xilinx, se utiliza el software Vivado.

Otros programas como LTSpice no disponen de estas funciones. Proteus, por el contrario, ofrece un entorno más completo para el diseño y simulación profesional de FPGAs. Sin embargo, dado que la programación en VHDL y el uso de FPGAs no constituye uno de lo principales objetivos de aprendizaje de la asignatura, el uso de Logisim-Evolution puede ser más que suficiente para una primera iniciación en este campo.

Amplia biblioteca de componentes

Logisim-Evolution dispone de una gran biblioteca de componentes, que incluye LEDs, TTLs (componentes de lógica transistor-transistor), interruptores y SoCs (*System on Chips*). Esta extensa biblioteca permite a los usuarios diseñar una amplia gama de circuitos, desde simples puertas lógicas hasta sistemas digitales más complejos. Sin embargo, a pesar de todos los componentes que ofrece no dispone de un manual completo actualizado en el que poder informarse de los diferentes elementos y sus propiedades así como de otras funcionalidades que ofrece el programa.

Otro de los problemas que plantea es que no permite copiar circuitos entre ventanas de diferentes proyectos, por lo que hay que utilizar métodos alternativos para incorporar un circuito previamente realizado a un nuevo programa.

2.2. Uso de la aplicación

Logisim-Evolution es una aplicación escrita en Java, por lo que puede ejecutarse en cualquier sistema operativo que contenga el entorno de ejecución de Java. En Github [4] se proporciona tanto el enlace para la descarga de Java, como para la descarga del programa en cualquier sistema operativo.

La apariencia del programa al abrirlo se muestra en la Fig. 2.1. En esta se puede

apreciar en (1) la barra del menú, en (2) los componentes lógicos más simples junto con las entradas y salidas de tipo pin, un puntero y una mano, con la cual se modifican los valores del circuito. En (3), en la parte superior, se almacenan con su nombre los circuitos que hayan sido creados, en este caso únicamente el circuito principal (*main*), y más abajo se ve el despliegue del tipo de componentes que tiene la biblioteca del programa.

En la parte inferior (4), en la pestaña *Properties*, se muestran las propiedades del componente seleccionado, las cuales se pueden modificar según el número de bits que se vayan a manejar, la apariencia del elemento, la orientación, etc. En la pestaña *State* se muestra, en caso de que haya bancos de registros, la información que haya almacenada en cada uno de ellos durante la simulación del circuito. Por último, la barra inferior del programa (5) es la barra correspondiente al *zoom*, ya que los tamaños de los posibles circuitos a implementar pueden diferir mucho.

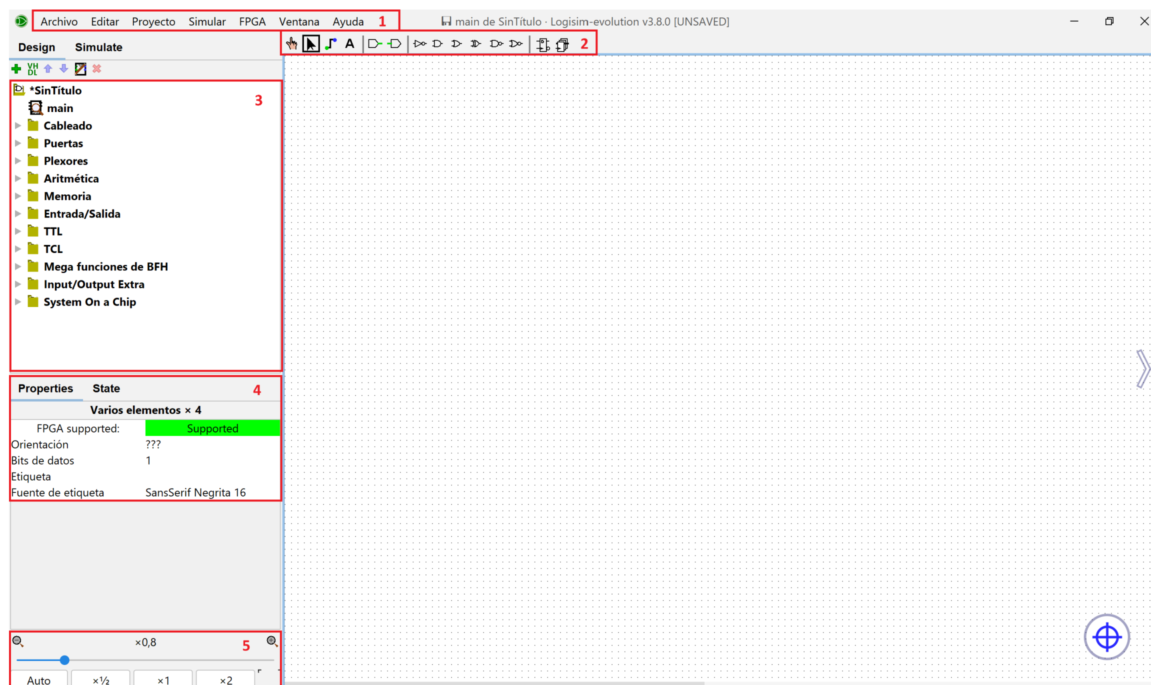


Figura 2.1: Pantalla de inicio de Logisim-Evolution.

A continuación, se muestran algunos circuitos simples y cómo se han construido a partir de elementos y herramientas del programa que se van a utilizar a lo largo del desarrollo del trabajo. El objetivo principal consiste en entender correctamente cómo se han creado los circuitos que forman parte de la estructura de RISC-V.

2.2.1. Uso de los componentes de la biblioteca

Para crear circuitos simples a raíz de los componentes de la biblioteca únicamente hace falta seleccionar los elementos a utilizar como se muestra a través de un ejemplo, en Fig. 2.2.

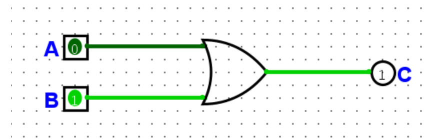


Figura 2.2: ejemplo de circuito simple: puerta OR.

2.2.2. Creación de un subcircuito

En más de una ocasión es necesario encapsular una parte de un circuito más grande en un elemento más pequeño y manejable. Esto es conocido como un subcircuito. Para crearlo hay que seguir los pasos que muestra la Fig. 2.3. Como ejemplo se ha creado un subcircuito de una puerta AND con dos entradas de cuatro bits. Para que las entradas y salidas del subcircuito puedan ser utilizadas desde el circuito principal, estas deben tener un cableado de tipo túnel en lugar de tipo pin.

Una entrada de tipo túnel es una herramienta que permite conectar diferentes puntos de un circuito sin necesidad de dibujar explícitamente los cables que los unen. Esto es especialmente útil cuando se necesita conectar puntos que están muy separados en el diseño del circuito, ya que evita que el diagrama se vuelva confuso y desorganizado. Esta herramienta se utiliza a lo largo del proyecto en múltiples ocasiones.

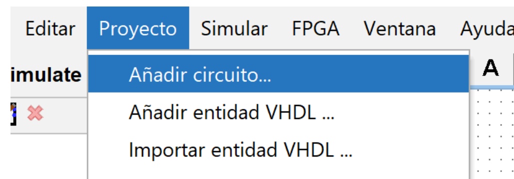
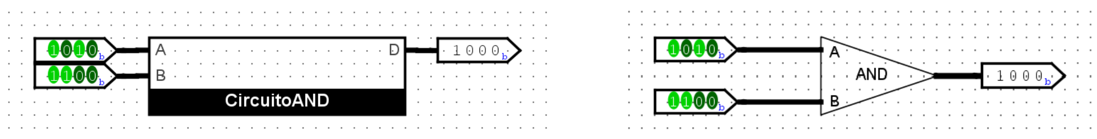


Figura 2.3: Añadir subcircuito.

La apariencia de un subcircuito se puede elegir de entre las opciones que ofrece el programa (Fig. 2.4a) o diseñar uno a medida (Fig. 2.4b). Para los distintos elementos que componen la estructura de RISC-V se ha diseñado a medida su apariencia.



(a) Subcircuito AND con apariencia sugerida por Logisim-Evolution.

(b) Subcircuito AND con apariencia a medida.

Figura 2.4: Subcircuito AND con diferentes apariencias.

2.2.3. Creación de un circuito combinacional

Una forma rápida de crear un circuito que cumpla una tabla de verdad es a través de la opción de circuito combinacional (Fig, 2.5). Su creación se basa en definir previamente las variables de salida y de entrada, y cuántos bits las componen, y después definir el valor que toma la salida en la tabla de verdad.

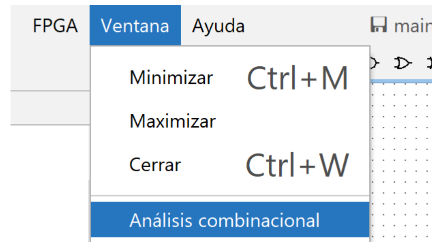
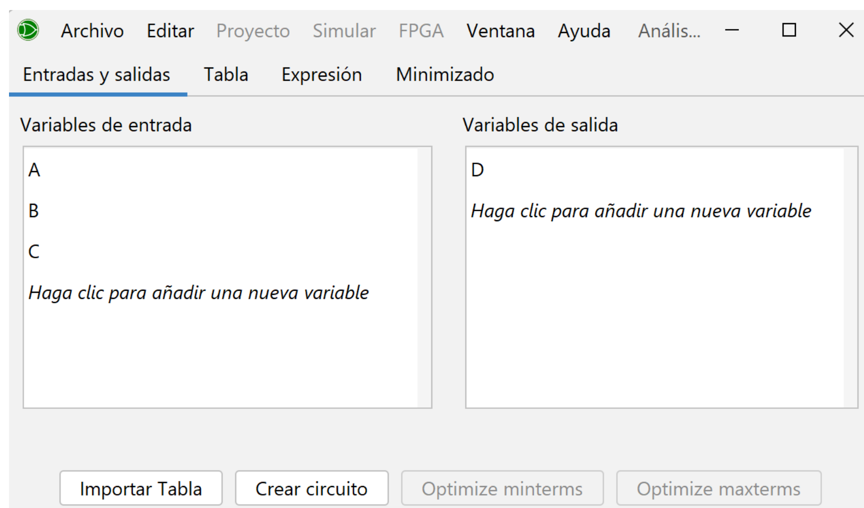
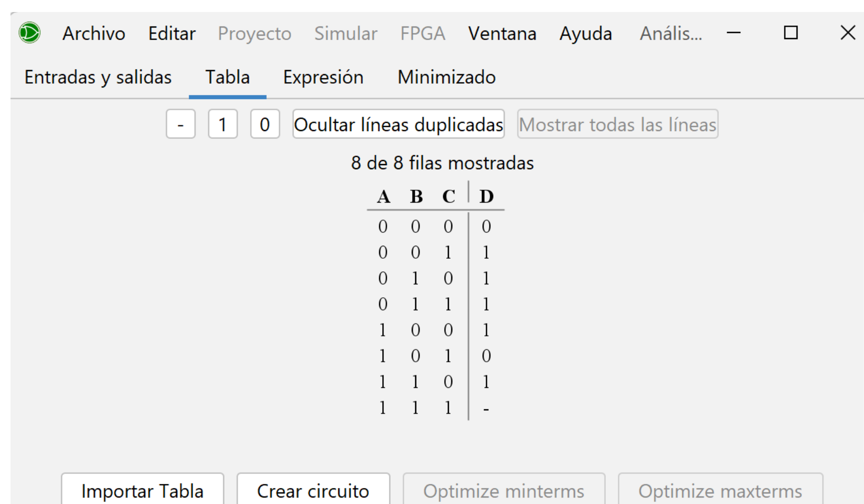


Figura 2.5: Añadir circuito combinacional.

En el ejemplo mostrado (Fig. 2.6) tenemos tres variables de entrada de un bit cada una y una sola salida cuyos posibles valores son: 0, 1 y DC (*Don't Care*). Para casos de circuitos con muchas más variables de entrada y salida antes de crear el circuito resultante es necesario optimizar la función de salida.



(a) Variables de entrada y salida del circuito combinacional.



A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	-

(b) Tabla de verdad del circuito combinacional.

Figura 2.6: Creación paso a paso del circuito combinacional.

El programa realiza la optimización del circuito resultante de la tabla de verdad introducida por el usuario a través del método seleccionado: optimización por mintems o por maxterms. La diferencia entre optimizar una función digital utilizando minterms o maxterms radica en cómo se estructura la función. La optimización por minterms se utiliza para expresar la función como una suma de productos, mientras que la optimización por maxterms se utiliza para expresar la función como un producto de sumas.

Independientemente de por qué método se optimice el objetivo es realizar ajustes y mejoras en el diseño para cumplir con requisitos específicos de rendimiento, eficiencia y costo.

Finalmente, el circuito combinacional resultante (Fig. 2.7) se guarda como un subcircuito del circuito principal, el cual puede ser manejado y cambiado de aspecto como se menciona anteriormente.

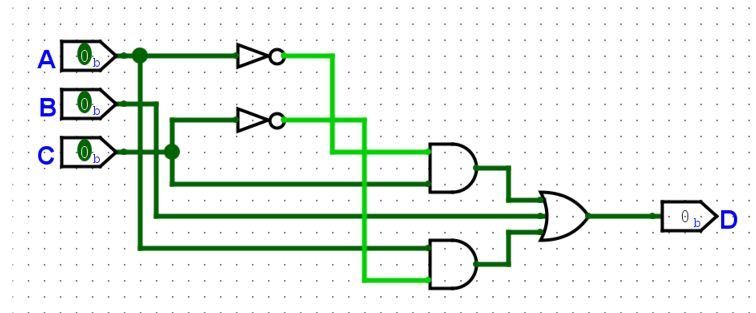


Figura 2.7: Circuito combinacional resultante.

Capítulo 3

Implementación de la arquitectura RV32I

En este capítulo, se procede a la implementación de la arquitectura RV32I utilizando la aplicación Logisim-Evolution. En el capítulo anterior, se presenta el programa junto con los conocimientos fundamentales y las herramientas necesarias para el uso de esta aplicación, lo que sienta las bases para el trabajo que se desarrolla a continuación.

La arquitectura RV32I se implementa de manera modular, abordando cada uno de los tipos de instrucciones en secciones separadas. Este enfoque modular no solo facilita una comprensión más clara y detallada de cada componente, sino que también permite una integración progresiva de todos los elementos de la arquitectura.

Cada sección de este capítulo se centra en un tipo específico de instrucción, proporcionando una descripción detallada de su formato de instrucción, los elementos necesarios de la arquitectura para su ejecución, su función a desarrollar y su implementación en Logisim-Evolution. Sin embargo, la primera sección de este capítulo es una excepción, ya que se enfoca en explicar la función y la implementación de las partes del circuito que son más generales y que se utilizan en todas o casi todas las instrucciones.

A la hora de desarrollar la implementación de cada tipo de instrucción se comienza implementando las instrucciones más simples que emplean menos partes de la arquitectura, pero utilizan componentes básicos esenciales para el resto de las instrucciones. De esta manera, se va incrementando tanto la estructura del circuito con cada nueva instrucción como la complejidad de las funciones de las instrucciones, comenzando con operaciones lógicas y aritméticas y avanzando hacia saltos en la memoria.

Es importante destacar que, a lo largo de este capítulo, se adopta una convención específica para la nomenclatura de los campos y señales. Los nombres de los campos que forman los formatos de instrucción se presentan en cursiva, lo que facilita su identificación y diferenciación del resto del texto. Asimismo, todas las señales que controlan el flujo de datos a través de la arquitectura también aparecen en cursiva. Por otro lado, las entradas y salidas a los circuitos correspondientes al resto de señales (que no son de control) se escriben en formato normal, sin cursiva, para mantener una clara distinción

entre los diferentes tipos de elementos.

En resumen, este capítulo proporciona una guía detallada para la implementación de la arquitectura RV32I en Logisim-Evolution, desglosando cada tipo de instrucción y explicando cómo se utilizan y se implementan los diferentes componentes de la arquitectura.

3.1. Elementos clave

Para iniciar la implementación de la arquitectura de instrucciones RV32I, este capítulo presenta los circuitos comunes a todas o casi todas las instrucciones. El objetivo es proporcionar una comprensión más clara de la implementación que se desarrollará en los capítulos siguientes de la arquitectura RV32I.

3.1.1. Memoria de programa y contador de programa

La arquitectura RISC-V incluye componentes esenciales como la memoria de instrucciones y el contador de programa (PC), que trabajan en conjunto para la ejecución de instrucciones. La memoria de instrucciones almacena el código del programa que el procesador necesita ejecutar. Cada instrucción en RISC-V tiene un tamaño fijo de 32 bits (4 bytes), lo que facilita la gestión y el acceso a las instrucciones.

El contador de programa (PC) es un registro que contiene la dirección de la siguiente instrucción a ejecutar. Después de que se ejecuta una instrucción, el PC se incrementa automáticamente para apuntar a la siguiente instrucción en la secuencia. Este proceso asegura que las instrucciones se ejecuten en el orden correcto, a menos que una instrucción de control de flujo (de tipo J o de tipo B) modifique el valor del PC para cambiar la secuencia de ejecución [19].

En cuanto a la alineación de la memoria, cabe destacar que las instrucciones son de 4 bytes, por lo que las direcciones de las instrucciones deben estar alineadas en múltiplos de 4. Esta alineación es crucial para asegurar un acceso eficiente y correcto a las instrucciones en la memoria.

3.1.1.1 Implementación en Logisim-Evolution

En la Figura 3.1 se muestra el circuito que incluye el contador de programa (PC) y la memoria de instrucciones. El PC se incrementa en 4 (004 en hexadecimal) después de cada ciclo de reloj (CLK), lo que refleja la alineación de las instrucciones en la memoria. Las señales de control como WE (*Write Enable*) y CLR (*Clear*) también están presentes para gestionar el funcionamiento del PC y asegurar que las instrucciones se ejecuten correctamente y en el orden adecuado.

El contador de programa se implementa mediante un registro idéntico a los que se usan en el banco de registros explicado en la Sección 3.2. Este registro recibe el valor de la dirección en la que se almacena la siguiente instrucción a ejecutar. Aunque la ejecución lógica de las instrucciones es de una en una y, por eso, cada vez se incrementa en

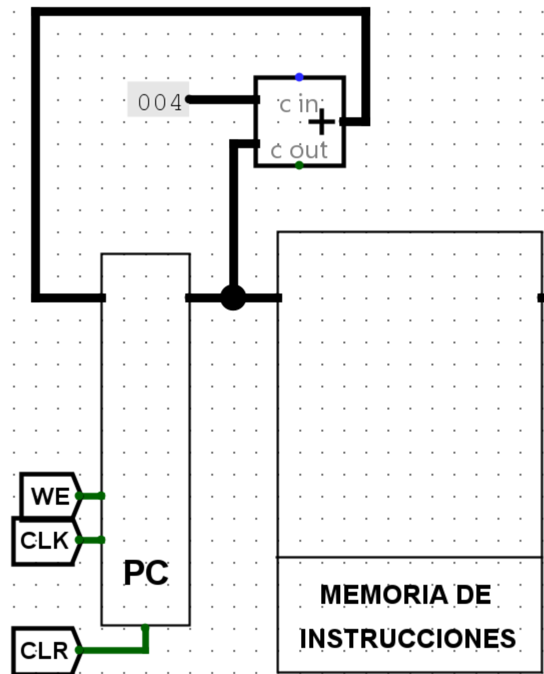


Figura 3.1: Apariencia a medida del contador de programa y la memoria de programa.

un valor de 4, también hay ejecuciones de salto de instrucción. Estas instrucciones calculan la dirección de la siguiente instrucción a ejecutar con valores de 32 bits, pues es el tamaño que maneja la arquitectura.

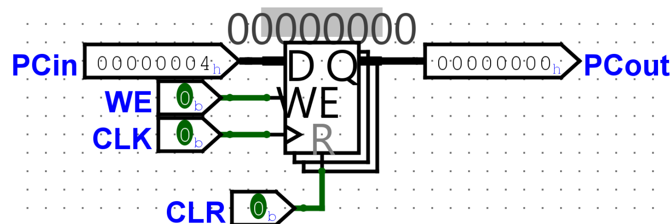


Figura 3.2: Registro que realiza la función de contador de programa. Almacena la dirección de la memoria de programa que contiene la siguiente instrucción a ejecutar.

La memoria ROM de 1K, utilizada como memoria de programa en el circuito, tiene un total de 1024 direcciones (2^{10}). Eso hace que solo necesite 10 de los 32 bits que salen del contador de programa para indicar la dirección de la siguiente instrucción a ejecutar. Además, las instrucciones deben estar alineadas en direcciones múltiplos de 4, lo que requiere que las dos últimas cifras de la dirección en binario sean ‘00’.

3.1.2. Unidad de control

La unidad de control (CU) es un componente esencial en la arquitectura de los procesadores RISC-V, así como en otros tipos de arquitecturas de procesadores. Su función principal es dirigir y coordinar las operaciones del procesador, actuando como el cerebro del sistema al controlar el flujo de datos y las operaciones de ejecución de instrucciones

dentro de la CPU (Unidad Central de Procesamiento) [20]. La unidad de control recibe instrucciones de la memoria del programa, las decodifica para entender qué operaciones deben realizarse y luego genera señales de control que dirigen el funcionamiento de otras partes del procesador, como la Unidad de Lógica Aritmética (ALU), los registros y la memoria de datos [21].

El propósito de la unidad de control es asegurar que todas las operaciones dentro del procesador se realicen de manera ordenada y sincronizada. Esto incluye la secuencia de pasos en el ciclo de ejecución de instrucciones, que comprende la búsqueda de instrucciones, la decodificación, la ejecución, el acceso a la memoria y el almacenamiento de resultados. Al controlar estos pasos, la unidad de control permite que el procesador ejecute programas de manera eficiente, interpretando las instrucciones del software y convirtiéndolas en acciones específicas realizadas por el hardware.

Para la implementación de la unidad de control de la arquitectura RV32I se ha construido una tabla de verdad (Tabla 3.1) con todas las instrucciones con valores distintos del campo *opcode* de la instrucción. En ella se muestran, para los diferentes tipos de instrucción, los distintos valores que toman las señales de control que dirigen el flujo de datos de la arquitectura y que se implementan en los siguientes capítulos.

Tipo	**opcode**	**ALU Fnt**	**Esc Reg**	**Leer Mem**	**Esc Mem**	**Salto Cond**	**Salto Incond**	**D_IN_BR**
I (load)	0000011	1	1	1	0	0	00	100
I	0010011	1	1	0	0	0	00	011
U (auipc)	0010111	x	1	x	0	0	00	001
S	0100011	1	0	x	1	0	xx	xxx
R	0110011	0	1	0	0	0	00	011
U (lui)	0110111	x	1	x	0	0	00	000
B	1100011	0	0	x	0	1	00	xxx
I (jalr)	1100111	x	1	0	0	0	10	010
J	1101111	x	1	0	0	0	01	010

Tabla 3.1: Tabla de verdad del subcircuito Unidad de control.

Cabe destacar, que no todas las señales de control están incluidas en esta tabla, ya que hay señales que necesitan más valores además del almacenado en el campo *opcode* para ser generadas. Este es el caso, por ejemplo, de la señal de control que maneja el resultado de la operación que realiza la unidad aritmético-lógica ($ALUop$). Además del *opcode*, hace uso del campo *funct3* y del bit en la posición 30 de la instrucción. Este último es necesario debido a que las instrucciones **sub** y **sra** comparten los valores de los campos *funct3* y *opcode* con las instrucciones **add** y **srl** de tipo R (Sec. 3.2), respectivamente. La tabla de verdad que sigue esta señal es la Tabla 3.2.

Por último, a partir de los tres bits del campo *funct3* del formato de la instrucción también se obtienen señales de control. Este campo sirve para manejar flujos de datos en las instrucciones de carga y almacenamiento de datos en memoria y en las instrucciones de tipo B.

Instrucción	Bit 30	funct3	opcode	ALUop
add / addi	0	000	0110011 / 0010011	0000
sll / slli	0	001	0110011 / 0010011	0001
slt / slti	0	010	0110011 / 0010011	0010
sltu / sltiu	0	011	0110011 / 0010011	0011
xor / xori	0	100	0110011 / 0010011	0100
srl / srli	0	101	0110011 / 0010011	0101
or / ori	0	110	0110011 / 0010011	0110
and / andi	0	111	0110011 / 0010011	0111
sub	1	000	0110011	1000
sra / srai	1	101	0110011 / 0010011	1001
sb	x	000	0100011	0000
sh	x	001	0100011	0000
sw	x	010	0100011	0000
lb	x	000	0000011	0000
lh	x	001	0000011	0000
lw	x	010	0000011	0000
lbu	x	100	0000011	0000
lhu	x	101	0000011	0000

Tabla 3.2: Tabla de verdad del subcircuito ControlALU integrado en el subcircuito Unidad de control.

3.1.2.1 Implementación en Logisim-Evolution

Para la implementación del subcircuito Unidad de control, al igual que para la implementación del subcircuito ControlALU integrado, se siguen los pasos indicados en la Subsección 2.2.3. Primero se definen las tablas de verdad y después se optimiza el circuito con ayuda del programa Logisim-Evolution.

La unidad de control (Figura 3.3) recibe los 32 bits de la instrucción en ejecución y mediante un separador, se obtienen los bits pertenecientes a los campos *opcode* y *funct3* además del bit en la posición 30 de la instrucción. Las señales de control generadas por la unidad de control son obtenidas a partir del *opcode* de cada instrucción. Estas señales son más básicas y se fundamentan en controlar qué partes de la arquitectura utiliza cada tipo de instrucción. Esto implica poder diferenciar qué tipo de instrucción puede leer o almacenar un dato en la memoria o escribir en un registro del banco de registros, entre otras funciones. Cada una de estas señales con sus respectivas funciones se explican en la implementación de la arquitectura a lo largo del proyecto.

Por otro lado, el circuito generador de la señal que controla el resultado de la operación que sale de la ALU (ControlALU) se puede ver integrado en el circuito Unidad de control. Para generar la señal *ALUop* hace uso de los campos *opcode*, *funct3* y el bit en la posición 30 decodificados de la instrucción. Este circuito, es un circuito combinacional simplificado gracias al diseño eficiente del formato de instrucciones que requieren operaciones lógicas o aritméticas. Esto significa que, a pesar de procesar más bits de

entrada, su diseño optimizado lo hace considerablemente menos complejo (Figura 3.4).

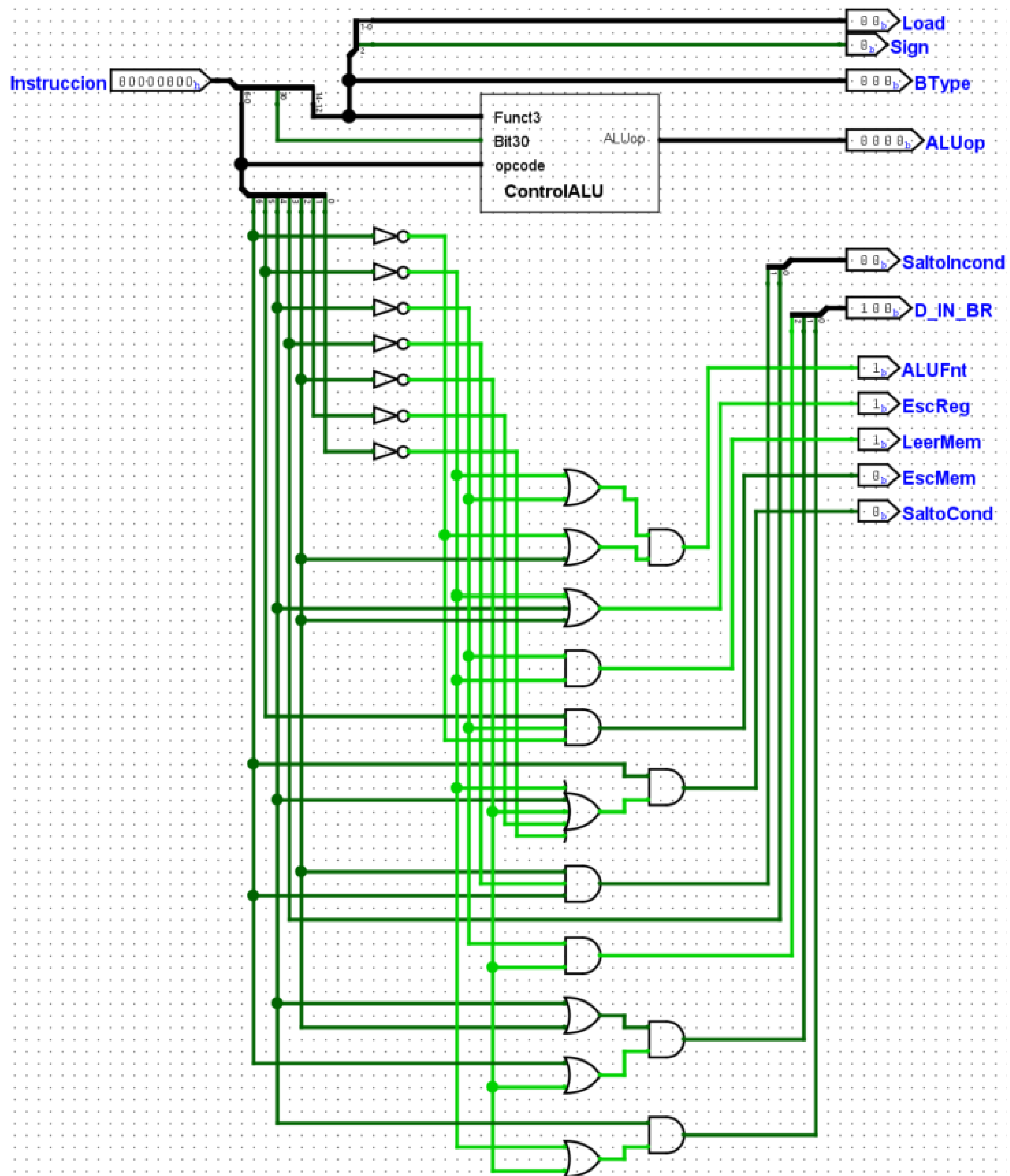


Figura 3.3: Subcircuito combinacional Unidad de control resultante de la tabla de verdad 3.1. El subsubcircuito ControlALU se encuentra dentro ya que también genera señales de control.

3.1.3. Generador de dato inmediato

El circuito generador del valor inmediato es una parte crucial de la arquitectura RISC-V, especialmente cuando se trabaja con instrucciones que contienen campos inmediatos

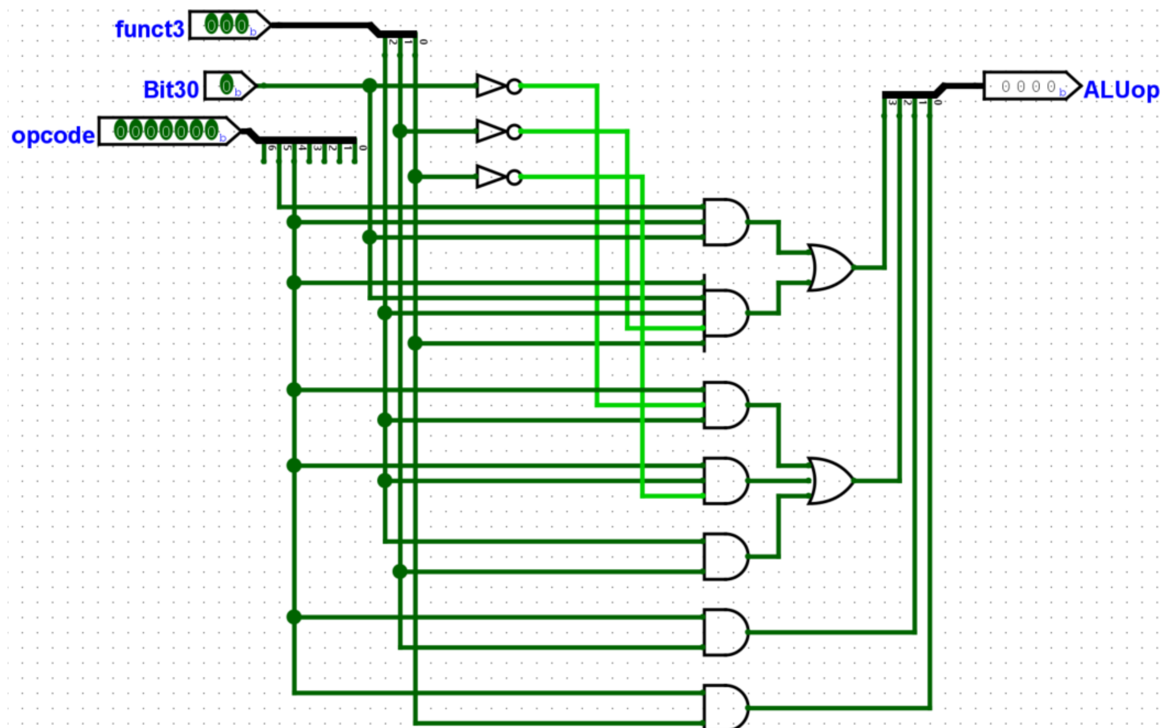


Figura 3.4: Subcircuito combinacional cuya salida es la señal de control que selecciona el resultado de la operación a realizar en la unidad aritmético-lógica.

de tamaño menor al de los registros de propósito general. En RISC-V, los registros son de 32 o 64 bits, dependiendo de la variante (RV32I o RV64I), pero los campos inmediatos en las instrucciones pueden ser de 12, 20 o incluso menos bits. El propósito del circuito generador del dato inmediato es aumentar el número de bits de estos valores inmediatos mientras se mantiene su signo y valor original, permitiendo así que puedan ser utilizados en operaciones aritméticas y lógicas con registros de tamaño completo.

El funcionamiento del circuito generador del dato inmediato es relativamente sencillo pero esencial para la correcta ejecución de las instrucciones. Cuando se recibe un valor inmediato, el circuito verifica el bit más significativo del inmediato (MSB), el cual determina su signo. Si este bit es '0', los bits adicionales se llenan con ceros, manteniendo el número positivo. Por otro lado, si el MSB es '1', los bits adicionales se llenan con unos, manteniendo el número negativo [22]. Este proceso se realiza de manera automática permitiendo que las operaciones aritméticas y lógicas se realicen correctamente sin importar el tamaño original del inmediato.

En la arquitectura de un procesador RISC-V, las instrucciones pueden tener diferentes formatos, tales como R, I, S, B, U, y J, cada uno con propósitos específicos y estructuras distintas. Los formatos de instrucción que incluyen campos inmediatos, como I, S, B, y J, requieren del circuito generador del dato inmediato para procesar correctamente los valores inmediatos que forman parte de la instrucción [23]. La Figura 3.5 muestra cómo se forma el valor del dato inmediato a partir de cada formato de los distintos tipos de instrucción. Los inmediatos producidos por cada uno de los formatos de instrucción

base están etiquetados para mostrar qué bit de instrucción produce cada bit del valor inmediato [24].

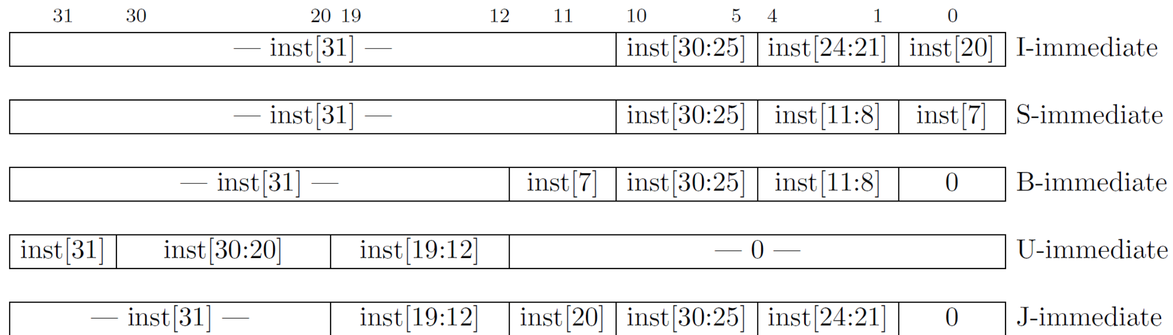


Figura 3.5: Tipos de inmediatos producidos por las instrucciones RISC-V. Los campos están etiquetados con los bits de instrucción utilizados para construir su valor [24].

3.1.3.1 Implementación en Logisim-Evolution

Para integrar el subcircuito generador del valor inmediato en el resto de la arquitectura RV32I, se ha diseñado una apariencia a medida muy simple (Figura 3.6) en la cual, como entrada, se introducen los 32 bits de la instrucción y, como salida, se extrae el valor del inmediato en un formato extendido de 32 bits para poder operar con el.

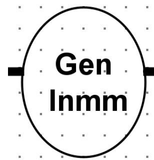


Figura 3.6: Subcircuito generador del valor inmediato con apariencia a medida.

El circuito está diseñado para manejar diferentes tipos de instrucciones. Cada tipo de instrucción tiene un formato específico para el campo inmediato, como se muestra en la Figura 3.5. Los 32 bits que forman la instrucción recibidos a la entrada del circuito, son desglosados con la ayuda de un separador. De la misma forma, con ayuda de un separador, los bits necesarios para la construcción del valor inmediato son recogidos en el orden y posición correspondiente para cada tipo de instrucción. Una vez recogidos los bits del campo que almacena el valor inmediato de la instrucción, se utiliza un extensor de signo que alarga dichos valores hasta un total de 32 bits.

Finalmente, el valor inmediato extendido se selecciona mediante el multiplexor y se envía a la salida del circuito. La señal de control *SelecImm* determina cuál de los valores inmediatos extendidos se debe utilizar, asegurando que el circuito funcione correctamente para cualquier tipo de instrucción de la arquitectura RV32I.

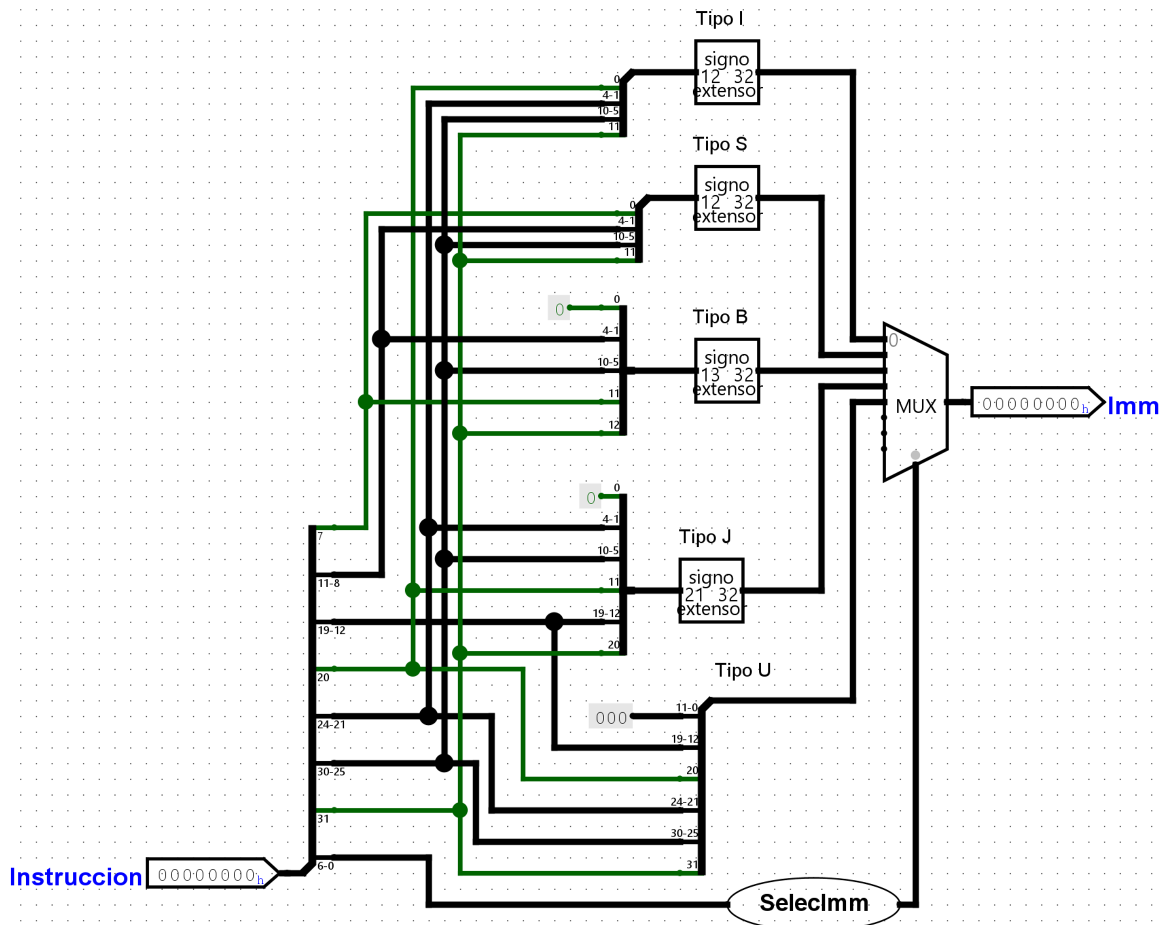


Figura 3.7: Interior del subcircuito generador del valor inmediato.

La señal de control *SelectImm* proviene de un pequeño circuito combinacional (Figura 3.8). En él, a partir de los bits almacenados en el campo *opcode* del formato de la instrucción, se distingue qué tipo de instrucción está en ejecución y, por tanto, qué valor de entrada al multiplexor debe seleccionarse como salida del circuito. La tabla de verdad correspondiente a este circuito combinacional es la Tabla 3.3.

Tipo	opcode	SelectImm
I	0010011 / 0000011	000
S	0100011	001
B	1100011	010
J	1101111	011
U	0110111 / 0010111	100

Tabla 3.3: Tabla de verdad del subcircuito selector del valor inmediato extendido en signo.

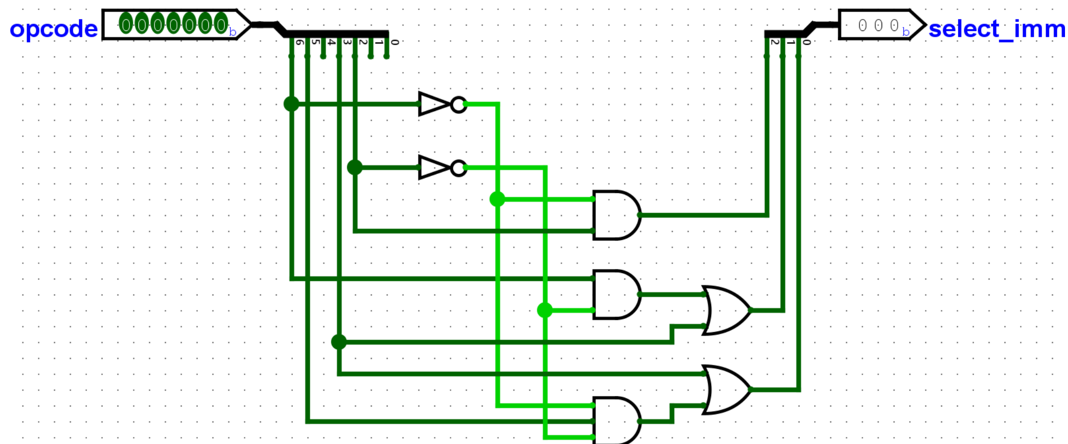


Figura 3.8: Subcircuito combinacional cuya salida es la señal selectora del dato extendido en signo.

3.2. Instrucciones de tipo R

Las instrucciones de tipo R realizan operaciones de diferentes tipos: aritméticas, lógicas, de comparación y desplazamientos, tanto aritméticos como lógicos. Independientemente de la operación que realicen, todas tienen el mismo formato de instrucción (Sección 1.3.1) con dos direcciones para los registros origen (*rs1* y *rs2*), puesto que las operaciones se realizan entre registros, y otra para el registro destino (*rd*), donde se almacena el resultado de la operación.

El resto de campos que completan el formato que define cada instrucción son el *opcode*, el cual es el mismo para todo el conjunto de instrucciones de tipo R, el *funct3* y el *funct7*, los cuales se encargan de diferenciar las instrucciones y, por lo tanto, la operación a realizar en cada una de ellas.

En código ensamblador las instrucciones del mismo tipo se escriben con el mismo formato. A continuación, como ejemplo, se muestra el código correspondiente a la instrucción de suma y las operaciones que realiza: **add** *rd*, *rs1*, *rs2* $\rightarrow rd = rs1 + rs2$.

3.2.1. Arquitectura necesaria

Las instrucciones de tipo R realizan un recorrido bastante reducido a través del hardware necesario para todo el RV32I. Para su implementación, solo se requiere de la memoria de programa, el banco de registros, la unidad aritmético-lógica (comúnmente conocida como ALU) y el circuito combinacional ControlALU que, como su nombre indica, se encarga de controlar la operación a realizar en la ALU.

La instrucción a ejecutar sale de la memoria de programa e indica al banco de registros la dirección de los registros que almacenan los valores con los cuales se va a operar. La ALU se encarga de realizar la operación correspondiente con dichos valores y devuelve el resultado de la operación a la dirección destino del banco de registros especificada en la instrucción.

3.2.2. Implementación en Logisim-Evolution

Para implementar la arquitectura necesaria para la ejecución de las instrucciones hemos diseñado cada parte por separado. A la hora de explicar cada parte del circuito, tanto en este capítulo como en los posteriores, la fuente utilizada para nombrar los campos de los formatos de instrucciones será itálica. Por el contrario, para los nombres de entrada y salida de los circuitos y subcircuitos se utilizará una fuente normal.

3.2.2.1 Banco de registros

En el diseño realizado a medida para el subcircuito del banco de registros (Fig. 3.9), se pueden apreciar dos entradas que indican las direcciones de los registros que contienen los valores a operar (rs1 y rs2), una entrada con la dirección del registro destino que almacenará el resultado de la operación (w_address) y la entrada a través de la cual se introduce dicho resultado (d_in). También tiene dos salidas (d_out1 y d_out2) que van conectadas a la entrada de la ALU con los valores almacenados en *rs1* y *rs2*, respectivamente.

Para obtener la dirección de los registros necesarios a la entrada del subcircuito se emplea un separador, de forma que los 32 bits de la instrucción a la salida de la memoria de programa puedan ser decodificados. Como resultado se obtienen tres salidas, de cinco bits cada una, con una dirección de registro almacenada.

La entrada situada en la parte superior (WE) proviene del circuito combinacional Unidad de control (Subsección 3.1.2), en el cual no vamos a hacer hincapié, ya que para la totalidad de las instrucciones R tomará un valor lógico de 1.

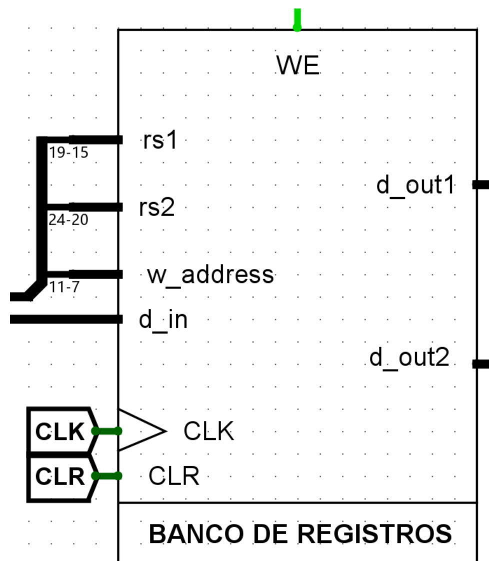


Figura 3.9: Subcircuito Banco de registros con apariencia a medida.

Por último, en la parte baja del banco de registros tenemos una entrada de reloj y una de reset que se utilizan en el interior del subcircuito para controlar los registros.

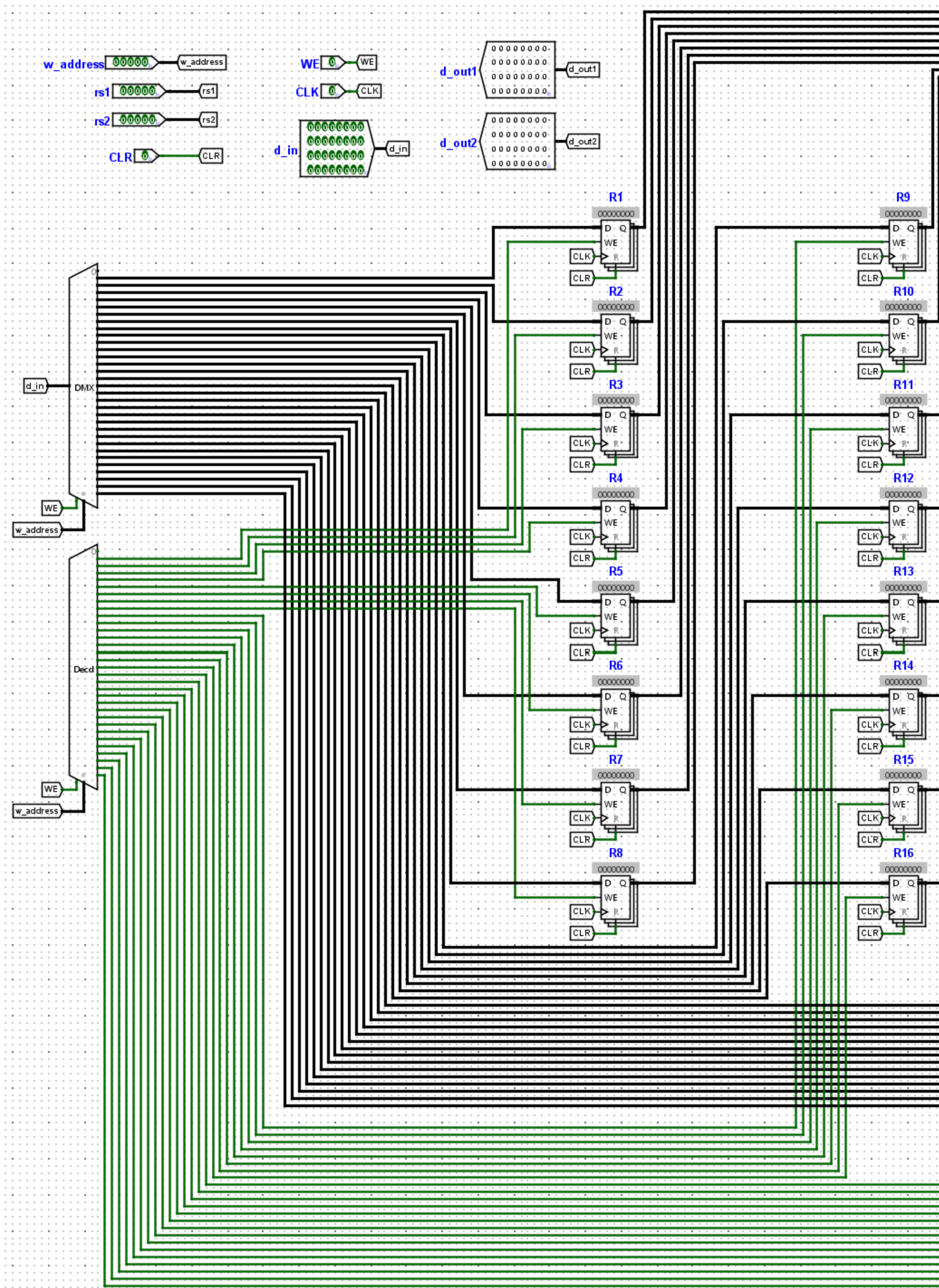
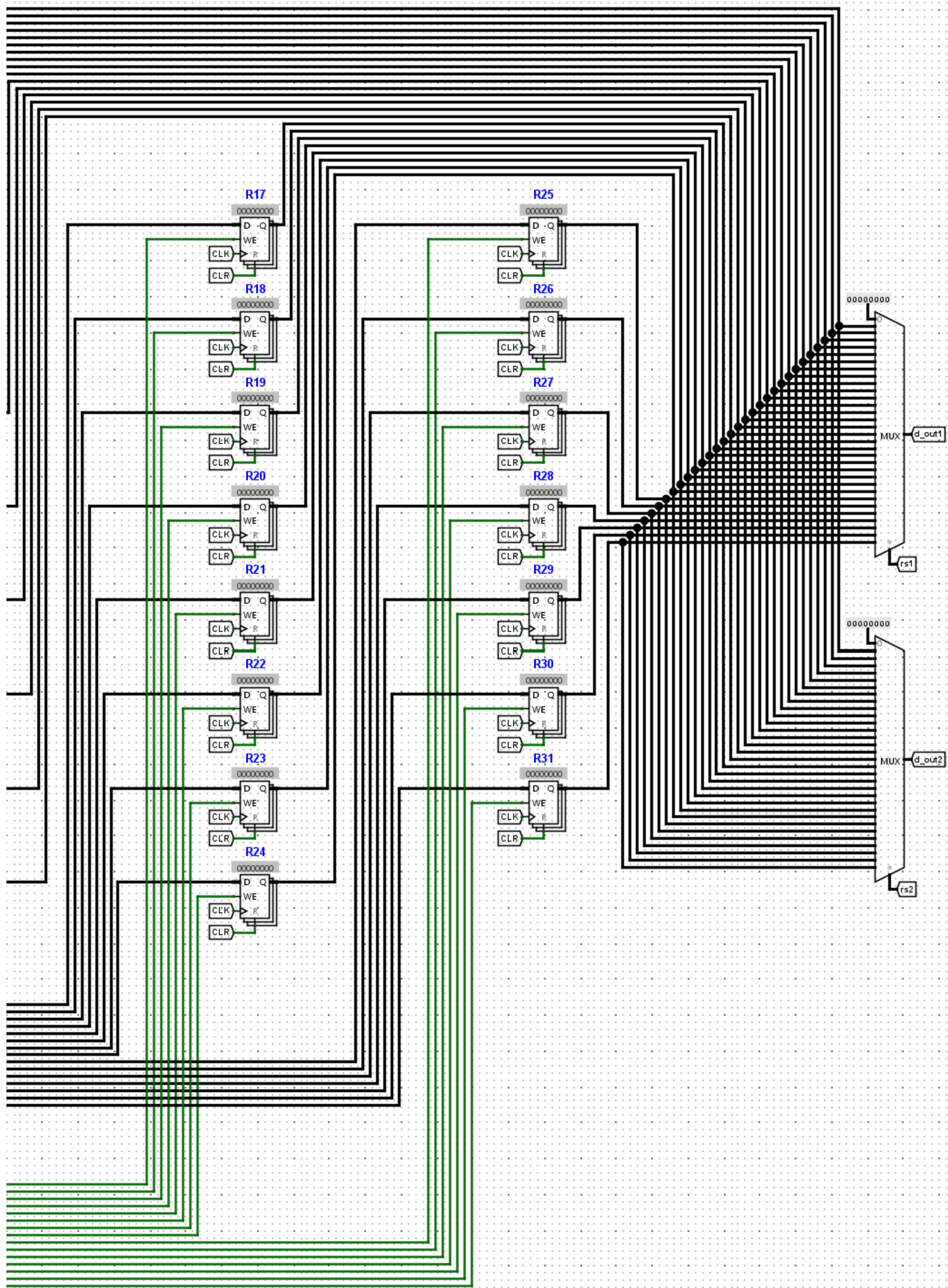


Figura 3.10: Interior del subcircuito Banco de registros.


R17
R18
R19
R20
R21
R22
R23
R24
R25
R26
R27
R28
R29
R30
R31
MUX
d_out1
rs1
MUX
d_out2
rs2

La arquitectura RV32I especifica un conjunto de 32 registros con 32 bits de ancho cada uno. Estos registros se utilizan para el almacenamiento de tanto los operandos como de los resultados de las operaciones con números enteros.

En este caso, el diseño del subcircuito del banco de registros dispone de 31 registros en lugar de 32 (Fig. 3.10). Esto se debe a que el valor del registro x0 siempre es cero.

Todos los registros disponen de una entrada de reloj (CLK) y una entrada que reinicia el valor de los registros a cero (CLR). Los valores almacenados por cada uno de los registros se pueden visualizar en hexadecimal encima del registro modificado o, como se menciona en el Capítulo 2, en la pestaña *State* a la izquierda de la ventana de edición del circuito.

Para controlar a qué registro entra el dato de entrada (d_in), se emplea un decodificador y un demultiplexor. Cada una de las salidas del decodificador llega a la entrada de habilitación de escritura (***W****rite* ***E****nable*) del registro correspondiente. La señal de selección indica qué salida del decodificador se debe activar y, por tanto, qué registro debe habilitar la escritura permitiéndole así almacenar el valor de entrada al banco de registros, es la señal w_address.

El demultiplexor empleado introduce el valor de entrada al banco de registros. Cada una de sus salidas conecta con la entrada de datos del registro correspondiente y, empleando también la señal de control w_address, se traslada el dato entrante al banco de registros a la dirección o registro indicado.

Para controlar los datos de salida del subcircuito se emplean dos multiplexores que funcionan de la misma forma. Todas las entradas de los multiplexores están conectadas a la salida de datos del registro correspondiente, a excepción de la entrada cero. Esta entrada corresponde con el registro x0 y, por tanto, se le asigna una constante de valor nulo para aumentar la simplicidad del subcircuito.

El dato de salida de cada uno de los multiplexores es habilitado por la señal de selección, que corresponde con la dirección de los registros *rs1* y *rs2* que proporciona la instrucción.

Por último, para hacer el subcircuito más limpio y visual, ya que muchas de las entradas se utilizan repetidamente a lo largo de este, se han utilizado túneles que están conectados a las respectivas entradas o salidas del exterior del banco de registros.

3.2.2.2 Unidad Aritmético-Lógica (ALU)

La unidad aritmético-lógica es la encargada de realizar todas las operaciones que requiera el conjunto de instrucciones. Las operaciones implementadas por las instrucciones de tipo R son muy básicas y serán utilizadas a posteriori también por otros tipos de instrucciones.

Como entrada tenemos dos túneles (rs1 y rs2_imm) que introducen los valores con

los que operar. El túnel rs2_imm puede recibir un número almacenado en un registro, como es en el caso de las instrucciones de tipo R, o un valor inmediato, para otro tipo de instrucciones.

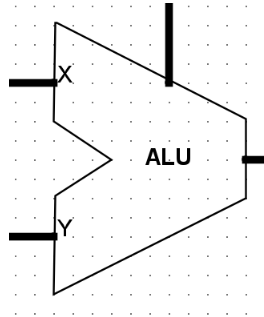


Figura 3.11: Subcircuito ALU con apariencia a medida.

Ambos datos entran en los diferentes bloques dispuestos para cada una de las posibles operaciones, de las cuales, solo será elegida una como salida. Para ello se utiliza un multiplexor controlado por el circuito combinacional Unidad de control, cuya señal proviene de la entrada situada en la parte superior de la ALU (Fig. 3.11).

Los bloques elegidos para cada tipo de operación se desarrollan a continuación.

Operaciones aritméticas

Dentro de las instrucciones con operaciones aritméticas tenemos la **add** (*addition*) y **sub** (*subtraction*). Estas se pueden implementar directamente a partir de los bloques sumador y restador, respectivamente, de la aplicación Logisim-Evolution.

Operaciones lógicas

Las instrucciones **and**, **or** y **xor** manejan los números a nivel de bits con sus respectivas puertas lógicas.

Comparaciones

Hay dos tipos de comparaciones: **slt** (*set less than*) y **sltu** (*set less than unsigned*). La primera instrucción compara los valores de los registros *rs1* y *rs2* en complemento a 2 y si el valor de *rs1* es menor devuelve un valor unitario al registro destino *rd*, de lo contrario, devuelve un valor de cero. La segunda instrucción realiza la misma operación, pero en lugar de comparar los valores en complemento a 2, lo realiza sin tener en cuenta el signo.

Esto es posible debido a que en los bloques de comparación de la aplicación, existe la posibilidad de cambiar la propiedad de interpretar los números de diferentes maneras.

Cabe destacar que estos comparadores devuelven una salida de un solo bit y la salida de la ALU debe devolver un número de 32 bits, por lo que, para ambas comparaciones es necesaria la aplicación de un extensor de signo a la salida del comparador (Fig. 3.12). De esta forma rellenamos los 31 bits restantes con ceros.

Desplazamientos

Las instrucciones de desplazamiento pueden realizar desplazamientos lógicos como **sll** (*shift left logic*) y **srl** (*shift right logic*) y aritméticos como **sra** (*shift rigt arithmetic*).

En la Figura 3.12, en el bloque de cada una de las instrucciones se puede apreciar que una de las dos entradas se realiza a través de un separador, el cual deja paso únicamente a los cinco primeros bits (del 0 al 4), que son los que indican las posiciones a desplazar. Esto se debe a que el banco de registros almacena números de 32 bits y, por lo tanto, no tendría sentido desplazar la posición de un número más de 32 bits o, lo que es lo mismo, 2^5, en ninguna dirección porque a partir de ese número el resultado siempre sería el mismo: todos los bits a cero.

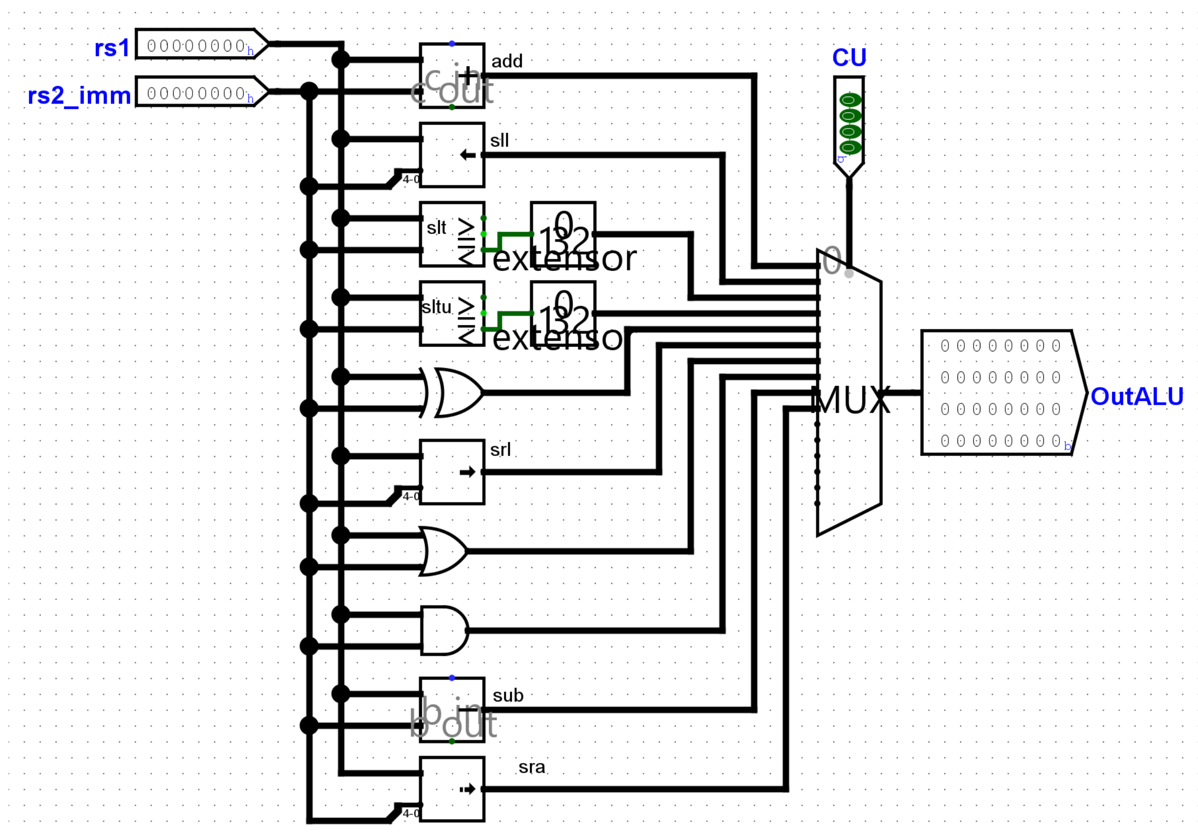


Figura 3.12: Interior del subcircuito ALU.

3.3. Instrucciones de tipo I

Dentro de las instrucciones de tipo I se distinguen tres funcionalidades diferentes: operaciones de carga de datos de la memoria al banco de registros, operaciones lógicas y aritméticas paralelas a las instrucciones de tipo R y una instrucción de salto incondicional.

Las distintas funcionalidades del tipo I se diferencian entre sí por los siete bits que

forman el campo *opcode*, aunque el formato que siguen todas las instrucciones se estructura de la misma forma. Todas disponen de un campo registro destino (*rd*), una dirección de memoria del banco de registros (*rs1*), un campo para distinguir las operaciones a realizar en la ALU (*func3*) y un campo de doce bits, formado por la agrupación de los campos *func7* y *rs2*, que almacena un valor inmediato (*imm*).

AL igual que el formato de la instrucción es igual para todas ellas, el comando de la instrucción en código ensamblador también se escribe siguiendo la misma estructura. A continuación, como ejemplo, se muestra el código correspondiente a la instrucción de suma y las operaciones que realiza: **addi** *rd*, *rs1*, *imm* $\rightarrow rd = rs1 + imm$.

3.3.1. Arquitectura necesaria

Debido a que las funciones que realiza cada subbloque de instrucciones es diferente, el recorrido a través de la arquitectura tampoco es el mismo. Para la ejecución total de las instrucciones de este tipo es necesario ampliar la arquitectura implementada previamente.

Las instrucciones paralelas a las vistas en el capítulo anterior utilizan las mismas partes del circuito y, a mayores, un multiplexor el cual selecciona el tipo de dato de una de las entradas de la ALU. También es necesaria una memoria de la cual se van a cargar datos al banco de registros con otras instrucciones y un multiplexor, cuya señal de control se llama *MemaReg*, que se explicará más adelante. Además, para la instrucción de salto incondicional se requiere de otro multiplexor que va conectado en serie con el multiplexor controlado por *MemaReg* y cuya señal de control se llama *saltoIncond*.

Por último, cabe destacar que todas las instrucciones de tipo I hacen uso del subcircuito *SingExtender* (Subsección 3.1.3). Este, como su propio nombre indica, se encarga de extender en signo, hasta 32 bits, el valor inmediato almacenado en la instrucción. De esta forma, los valores resultantes pueden ser operados mediante la ALU.

3.3.2. Implementación en Logisim-Evolution

Debido a que el recorrido a través de la arquitectura es diferente para cada una de las funciones de este tipo de instrucción, la explicación se divide en tantas partes como funciones hay.

3.3.2.1 Instrucciones con operaciones aritméticas, lógicas, de desplazamiento y de comparación

El recorrido de estas instrucciones es idéntico al visto previamente para las instrucciones de tipo R. Las operaciones realizadas son también las mismas y, por ello, la estructura de la ALU permanece igual. La única diferencia es que el segundo valor a operar lo genera el subcircuito generador del dato inmediato y no se obtiene a través del banco de registros.

Esta diferencia genera un pequeño conflicto, ya que se juntan dos señales con distintos

valores almacenados en ellas a la entrada del dato "Y" de la unidad aritmético-lógica. La solución, como se puede observar en la Figura 3.13, se basa en colocar un multiplexor con dos entradas que seleccione qué valor debe pasar.

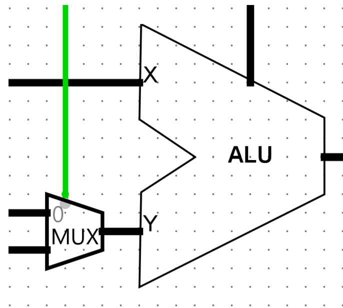


Figura 3.13: Subcircuito ALU con un multiplexor seleccionando el dato Y de entrada.

El multiplexor está controlado por una señal de control (*ALUFnt*) generada en el subcircuito combinacional Unidad de control. Cuando la señal de control toma un valor lógico de cero, el dato seleccionado como entrada a la ALU es el proporcionado por el banco de registros y, por tanto, la instrucción en ejecución no es la de tipo I. Por otro lado, si el valor de la señal es un uno lógico, el dato de entrada a la ALU es el valor inmediato extendido en signo.

3.3.2.2 Instrucciones *load*

Las instrucciones de carga de datos de la memoria al banco de registros, además de la ALU y del banco de registros hacen uso de una memoria en la que se almacenan datos y un multiplexor a su salida. Este multiplexor, controlado por la señal de control *MemaReg*, selecciona si el dato de entrada al banco de registros proviene de la operación realizada en la ALU o de la memoria, como es en el caso de las instrucciones *load*.

En la parte exterior del subcircuito (Fig. 3.14) se pueden ver varias entradas de control provenientes del subcircuito Unidad de control como *LeerMem* y *EscrMem* que permiten la lectura y escritura de datos, respectivamente. También están las entradas de control *Sign*, que permite diferenciar si la instrucción tiene en cuenta o no el signo del dato a cargar, y *Load*, que indica la longitud del dato a cargar. Esta última señal de control hace referencia a que se puede cargar un byte, dos bytes o cuatro bytes dependiendo de la instrucción de carga en ejecución.

Los accesos DataIN y DataOUT sirven para la entrada y salida de datos, respectivamente. Por último, la entrada dir recibe la dirección de la memoria de la cual se quiere, en el caso de las instrucciones de carga, extraer el valor para almacenarlo en la dirección del registro destino indicado en la instrucción. Esta dirección es el resultado de la suma realizada en la ALU del valor almacenado en el registro indicado por el campo *rs1* y el valor inmediato almacenado en *imm* extendido en signo.

En la parte interior del subcircuito podemos apreciar una memoria RAM, en este caso de 1024 direcciones, con un circuito auxiliar a su salida (Fig. 3.17). El direccionamiento

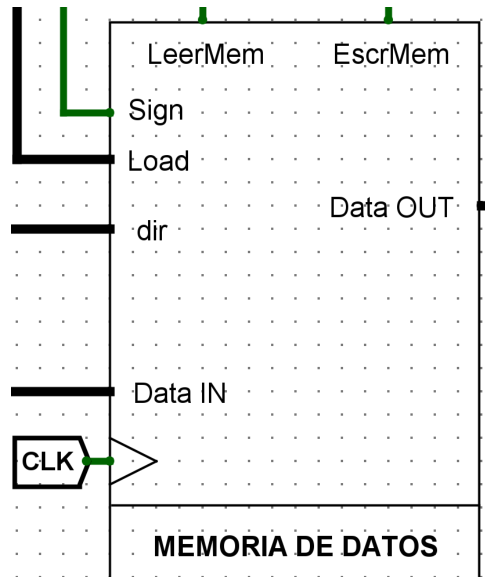


Figura 3.14: Subcircuito Memoria de datos por la parte exterior.

de la memoria de datos al almacenar valores de 32 bits, al igual que el de la memoria de programa, va de cuatro en cuatro bytes. Por este motivo, los dos últimos bits del resultado de la suma realizada en la ALU se omiten, obteniendo así una dirección múltiplo de cuatro.

El funcionamiento del circuito a la salida de la memoria se explica a continuación según los diferentes tipos de instrucciones *load* [25].

Load byte

Load byte, **lb**, como su nombre indica carga un único byte de la memoria de datos al registro destino. Esto se consigue con ayuda de un separador a la salida de la memoria que divide los 32 bits almacenados en la dirección calculada en cuatro partes de 8 bits cada una.

Los bytes resultantes de la separación entran a un multiplexor controlado por los dos bits omitidos previamente para acceder a la dirección de memoria. De esta forma si los dos bits son cero se accederá a los ocho bits más bajos del dato. Si el bit más bajo es uno y el segundo bit es cero se seleccionan los siguientes ocho bits, del 8 al 15, y así hasta completar las combinaciones lógicas posibles que son reflejadas en la Figura. 3.15.

La salida seleccionada por el multiplexor se extiende de 8 bits a 32 bits de nuevo, ya que el resto del circuito solo maneja datos de esa longitud. La extensión se realiza en signo o en ceros y ambos valores son recibidos a la entrada de otro multiplexor. Este nuevo multiplexor está controlado por el tercer bit del campo *funct3*, cuyo valor es 0 para las instrucciones que tienen en cuenta el signo y 1 para las instrucciones que no (*unsigned*).

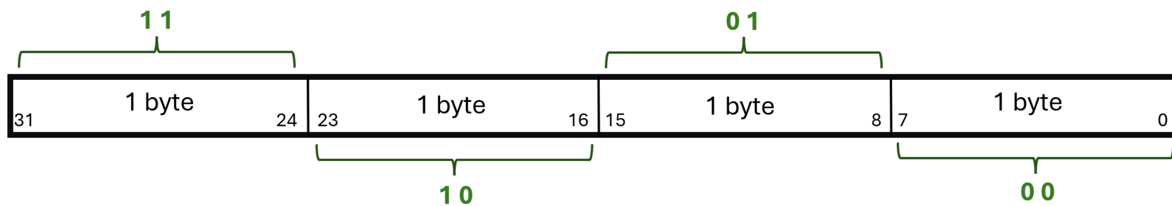


Figura 3.15: Byte cargado de la memoria de datos según los dos bits más bajos de la dirección de memoria calculada en la ALU.

Load halfword

Load halfword, **lh**, funciona de forma prácticamente idéntica a *load byte*, pero la longitud de los datos a seleccionar es de 16 bits. Las diferentes opciones de selección en este caso se reducen a tres en vez de cuatro (Fig. 3.16). Esto es debido a que la cuarta opción seleccionaría los ocho bits más altos de la dirección calculada más los ocho bits más bajos de la siguiente dirección en la memoria a la cual no se tiene acceso.

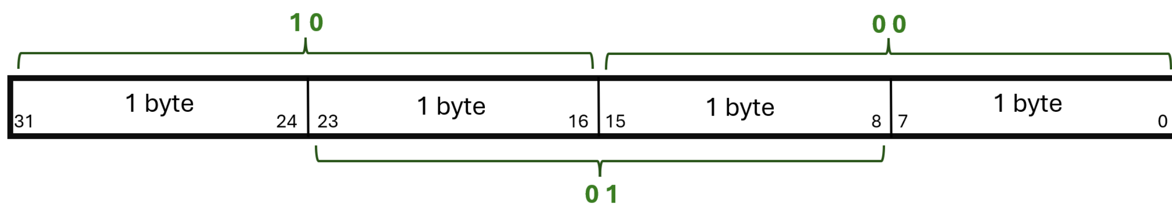


Figura 3.16: Bytes cargados de la memoria de datos según los dos bits más bajos de la dirección de memoria calculada en la ALU.

La salida del primer multiplexor se extiende ahora de 16 a 32 bits y, de la misma forma que para *load byte*, se selecciona el valor extendido en signo o sin signo con ayuda del segundo multiplexor.

Load word

Por último, la instrucción *load word*, **lw**, toma el valor completo almacenado en la dirección de memoria, por lo que la salida de la memoria se lleva directamente al último multiplexor.

Este multiplexor recibe también por sus entradas los valores seleccionados en los multiplexores anteriores manejados por la señal de control *Sign*. De esta forma la primera entrada recibe el valor de un byte extendido hasta 32 bits, la segunda entrada recibe dos bytes del dato de la memoria extendido también hasta 32 bits y la tercera entrada recibe el dato de la memoria completo.

El valor de salida del multiplexor y, por tanto de la memoria de datos, es seleccionado por la señal de control *Load* generada en la unidad de control, que corresponde a los dos bits más bajos del campo *funct3*. Finalmente, el búfer situado a la salida del último multiplexor, manejado por la señal *LeerMem*, habilita la salida del dato del subcircuito Memoria de datos.

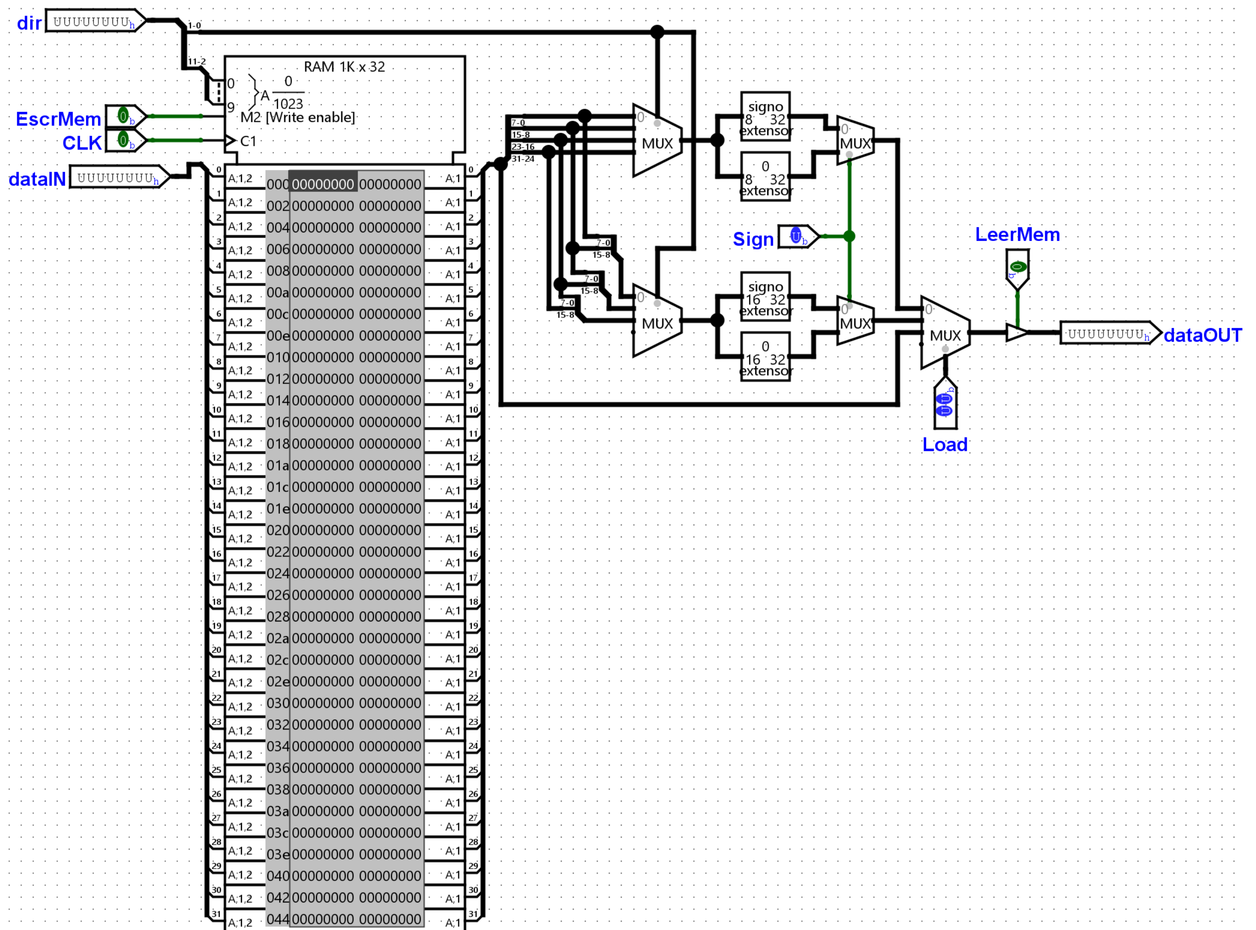


Figura 3.17: Subcircuito Memoria de datos, formado por la memoria RAM y un circuito auxiliar, para la correcta ejecución de las distintas instrucciones *load*.

3.3.2.3 Instrucción de salto incondicional

La instrucción **jalr** (*jump and link register*) permite realizar saltos a direcciones calculadas en tiempo de ejecución, lo cual es útil para implementar llamadas a funciones y retornos de funciones en lenguajes de programación de alto nivel. Además, almacena el valor de la que sería la siguiente instrucción de no haber un salto en el programa (PC + 4) en el registro destino indicado en la instrucción.

Para almacenar en el banco de registros el valor del contador de programa más cuatro, por los cuatro bytes que abarca una instrucción, se hace uso de un multiplexor controlado por la señal de control *SaltoIncond* y puesto en serie con el multiplexor a la salida de la memoria de datos controlado por la señal *MemaReg*.

Este nuevo multiplexor dispone de dos entradas (Figura 3.18). La primera recibe la salida del multiplexor anterior y es seleccionada cuando la señal de control toma un valor de cero. La segunda recibe el resultado de la suma a guardar en el registro destino y es seleccionada cuando la señal de control es uno. Para la realización de dicha suma es necesario un extensor de signo, debido a que el contador de programa se almacena

en 12 bits, pero el banco de registros solo admite datos de una longitud de 32 bits.

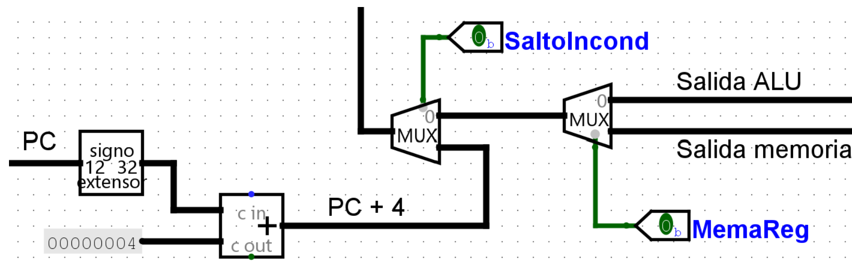


Figura 3.18: Multiplexores encargados de seleccionar el dato entrante al banco de registros.

Las señales de control *SaltoIncond* y *MemaReg* son generadas en el subcircuito Unidad de control. En la subsección correspondiente a la explicación de la unidad de control (Subsec. 3.1.2), estas señales no se incluyen. Esto es debido a las modificaciones del circuito realizadas en la Sección 3.6.

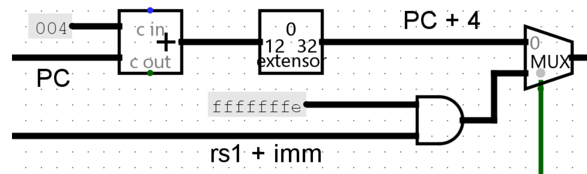


Figura 3.19: Subcircuito ALU con un multiplexor seleccionando el dato Y de entrada.

Por otro lado, la parte correspondiente al salto de esta instrucción se realiza también con la ayuda de un multiplexor con dos entradas controlado por la señal *SaltoIncond* (Fig. 3.19). La primera recibe el valor del Contador de programa más cuatro, que es la dirección en la memoria de programa de la instrucción inmediatamente posterior a la instrucción en ejecución. La segunda entrada corresponde al valor de la dirección de la memoria de programa a la que se va a saltar.

Este valor es el resultante de sumar el valor almacenado en el registro indicado por el campo *rs1* y el valor inmediato almacenado en el campo *imm* extendido en signo. Además, al resultado de esta operación se le aplica una máscara, con la puerta lógica AND, para enmascarar el último valor a cero. Esto se debe a que en RISC-V todas las instrucciones tienen una longitud de 32 bits (4 bytes) y deben estar alineadas en direcciones múltiplos de 4 bytes en la memoria [26].

3.4. Instrucciones de tipo S

Las instrucciones de tipo S, como su propio nombre indica (*store*), se utilizan para guardar en la memoria datos que están almacenados en el banco de registros. Dependiendo de la operación específica, una instrucción de tipo S puede guardar un byte, dos bytes o incluso el tamaño completo del dato, asegurando así la flexibilidad y eficiencia en la gestión de la memoria y la adaptabilidad a diferentes necesidades de almacenamiento de datos dentro de un programa.

El formato de las instrucciones de tipo S comienza con un campo *opcode* que indica que la instrucción es de almacenamiento, dos direcciones de registro *rs1* y *rs2*, un *func3* y un valor inmediato *imm*. Los doce bits que forman el campo que almacena el valor inmediato están separados. El campo, que en otras instrucciones almacenaba la dirección del registro destino, ahora almacena los primeros cinco bits del inmediato. Los siete bits restantes, en cambio, están al principio de la instrucción.

El comando en código ensamblador para ejecutar una instrucción de tipo S es muy parecido al visto en la sección anterior. La única diferencia se debe a que ahora en lugar tener un registro destino (*rd*) donde cargar el dato extraído, tenemos un segundo registro fuente (*rs2*) del que se obtiene el valor a almacenar en memoria. A continuación, se muestra el código correspondiente a la instrucción de almacenamiento de una palabra y las operaciones que realiza: **sw** *rs2*, *imm(rs1)* $\rightarrow Memoria[rs1{+}imm] = rs2$.

3.4.1. Arquitectura necesaria

Para el correcto funcionamiento de las instrucciones de tipo S hace falta la implementación de un pequeño subcircuito a la entrada de la memoria de datos. Este resuelve la implementación de las instrucciones **sb** (*store byte*) y **sh** (*store halfword*). Además del subcircuito, estas instrucciones hacen uso del banco de registros, la unidad aritmético-lógica, el subcircuito generador de dato inmediato y la memoria de datos.

La dirección de la memoria en la cual se va a guardar un dato es calculada en la ALU. Esta corresponde al valor resultante de sumar el dato inmediato extendido en signo con el valor almacenado en el registro indicado por el campo *rs1*. Por último, el valor del dato a almacenar en la memoria está guardado en el banco de registros en la dirección indicada por el campo *rs2* de la instrucción.

3.4.2. Implementación en Logisim-Evolution

La ejecución de la instrucción se divide en dos partes. En la primera se calcula la dirección de memoria en la cual se va a almacenar un valor y en la segunda se realiza el proceso de almacenamiento del mismo. La implementación de la primera parte ya está realizada, ya que corresponde a una simple operación de suma igual que en la instrucción **addi** de tipo I.

Para proceder a la implementación de la segunda parte de las instrucciones de tipo S es necesario conocer previamente su funcionamiento. Estas se basan en tomar datos que están en el banco de registros y almacenarlos en la memoria de datos. El tamaño de los datos a almacenar depende de la instrucción en ejecución **sb**, **sh** y **sw** que seleccionan un byte, dos bytes o los cuatro bytes del registro indicado, respectivamente (Figura 3.20).

Las instrucciones *store byte* y *store halfword* están diseñadas para almacenar específicamente los bytes menos significativos de un registro en la memoria. Para la instrucción **sb**, se almacena el byte menos significativo del registro especificado en la dirección

de memoria calculada. De la misma forma, la instrucción **sh** almacenan los dos bytes menos significativos del registro en la memoria [26].

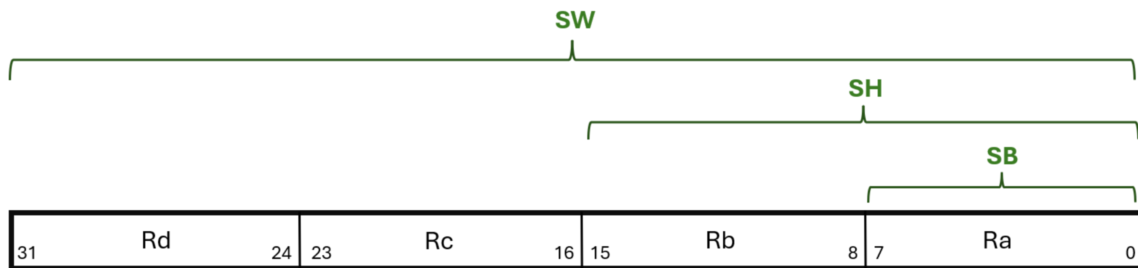


Figura 3.20: Selección de bytes del valor almacenado en el registro en función de la instrucción *store*.

Por otro lado, el RISC-V utiliza direccionamiento por bytes, por lo que cada byte en la memoria tiene su propia dirección única [27]. Esto significa que el dato tomado del banco de registros puede almacenarse en la dirección de memoria exacta del byte calculado. Cabe destacar, que el RISC-V no requiere que los datos estén naturalmente alineados, pero el acceso a datos no alineados puede no ser soportado por todas las implementaciones. En nuestro caso, por simplicidad del circuito, solo se permite el acceso en la memoria de datos alineados.

La alineación de memoria en el conjunto de instrucciones RISC-V se refiere a cómo se organizan y acceden los datos en la memoria para optimizar el rendimiento y evitar errores. En nuestro circuito, debido a que la longitud de una palabra es de cuatro bytes, cada palabra debe estar alineada en una dirección múltiplo de cuatro y es únicamente a esos cuatro bytes de la memoria a los que se permite el acceso. Esto significa que las dos últimas cifras de la dirección en binario deben ser 00, por esta razón, en nuestro circuito omitimos esos dos últimos bits a la entrada de la dirección de la memoria.

Por último, para terminar de entender el funcionamiento de las instrucciones de tipo *store*, cabe destacar que, en caso de almacenar un byte o dos en una dirección de memoria, los bytes de la dirección de memoria que no son cubiertos por el dato entrante del banco de registros permanecen intactos. Esto nos ofrece cuatro posibilidades de almacenamiento para la instrucción **sb**, tres para la instrucción **sh** y una para **sw**.

Para la realización del subcircuito necesitamos introducir el valor almacenado en el registro utilizando las entradas Ra-Rd, y la palabra almacenada en la dirección de memoria a la que se apunta con las entradas Ma-Md. Estos valores entran divididos en cables de 8 bits cada uno para la posible realización de las combinaciones de almacenamiento de cada instrucción.

En la parte exterior del subcircuito (Figura 3.21) hay también cuatro entradas de control. Las entradas D0 y D1 corresponden con los dos bits más bajos de la dirección a apuntar, es decir, los bits omitidos a la entrada de la dirección de memoria debido a la alineación de las direcciones. Estos bits nos indican en qué posición, dentro de los

cuatro bits que ocupa una palabra en memoria, se va a colocar el valor extraído del banco de registros. Las entradas F31 y F30, correspondientes a los dos bits más bajos del campo *funct3* de la instrucción, sirven para saber cuál de las tres instrucciones *store* es la que está en ejecución. Por último, el nuevo valor entrante a la memoria de datos sale a través de la salida MemIN.

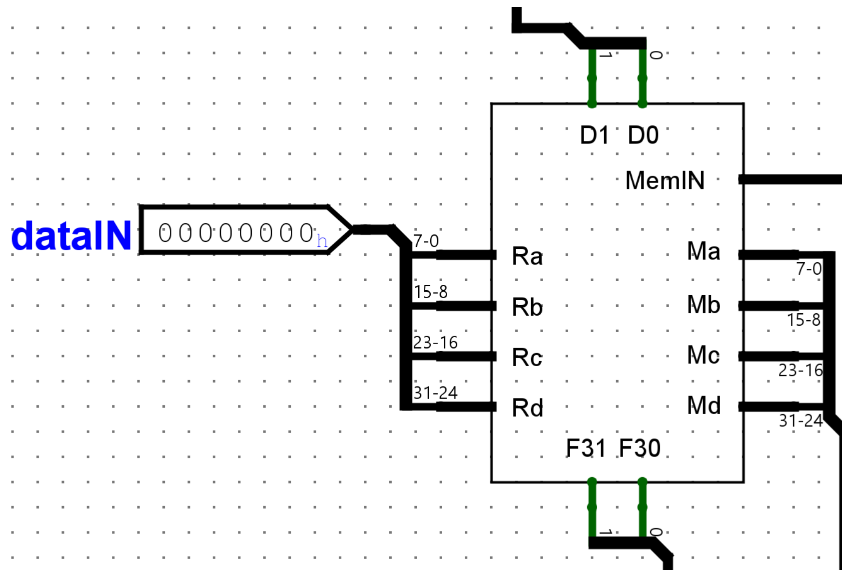


Figura 3.21: Apariencia a medida del subcircuito a la entrada de la memoria de datos para la implementación de las instrucciones *store*.

Debido a la complejidad que generan las diferentes posibles salidas, se ha planteado una tabla de verdad (Tabla 3.4). En ella se muestran todas las combinaciones de las entradas de control y los valores que toma cada byte de la palabra entrante a la memoria de datos para cada una de ellas. Cabe destacar que, para este tipo de instrucción, los dos últimos bits del *funct3* nunca van a tomar un valor lógico alto de forma simultánea.

Instrucción	**F31**	**F30**	**D1**	**D0**	**MemIN A**	**MemIN B**	**MemIN C**	**MemIN D**
SB	0	0	0	0	Ra	Mb	Mc	Md
	0	0	0	1	Ma	Ra	Mc	Md
	0	0	1	0	Ma	Mb	Ra	Md
	0	0	1	1	Ma	Mb	Mc	Ra
SH	0	1	0	0	Ra	Rb	Mc	Md
	0	1	0	1	Ma	Ra	Rb	Md
	0	1	1	0	Ma	Mb	Ra	Rb
	0	1	1	1	x	x	x	x
SW	1	0	x	x	Ra	Rb	Rc	Rd

Tabla 3.4: Tabla de verdad del subcircuito realizado para las instrucciones *store*.

Además, para el caso de la instrucción **sw**, independientemente del valor que tengan D0 y D1 solo hay un posible valor entrante en la memoria de datos, el valor completo

almacenado en el registro. Por último, para la instrucción **sh**, no es posible acceder al último byte de la palabra de la dirección apuntada. Esto se debe a que, al tener que almacenar dos bytes en memoria, requiere acceder también al primer byte de la siguiente palabra de la memoria de datos, lo cual no es posible en nuestra implementación por la alineación de la memoria.

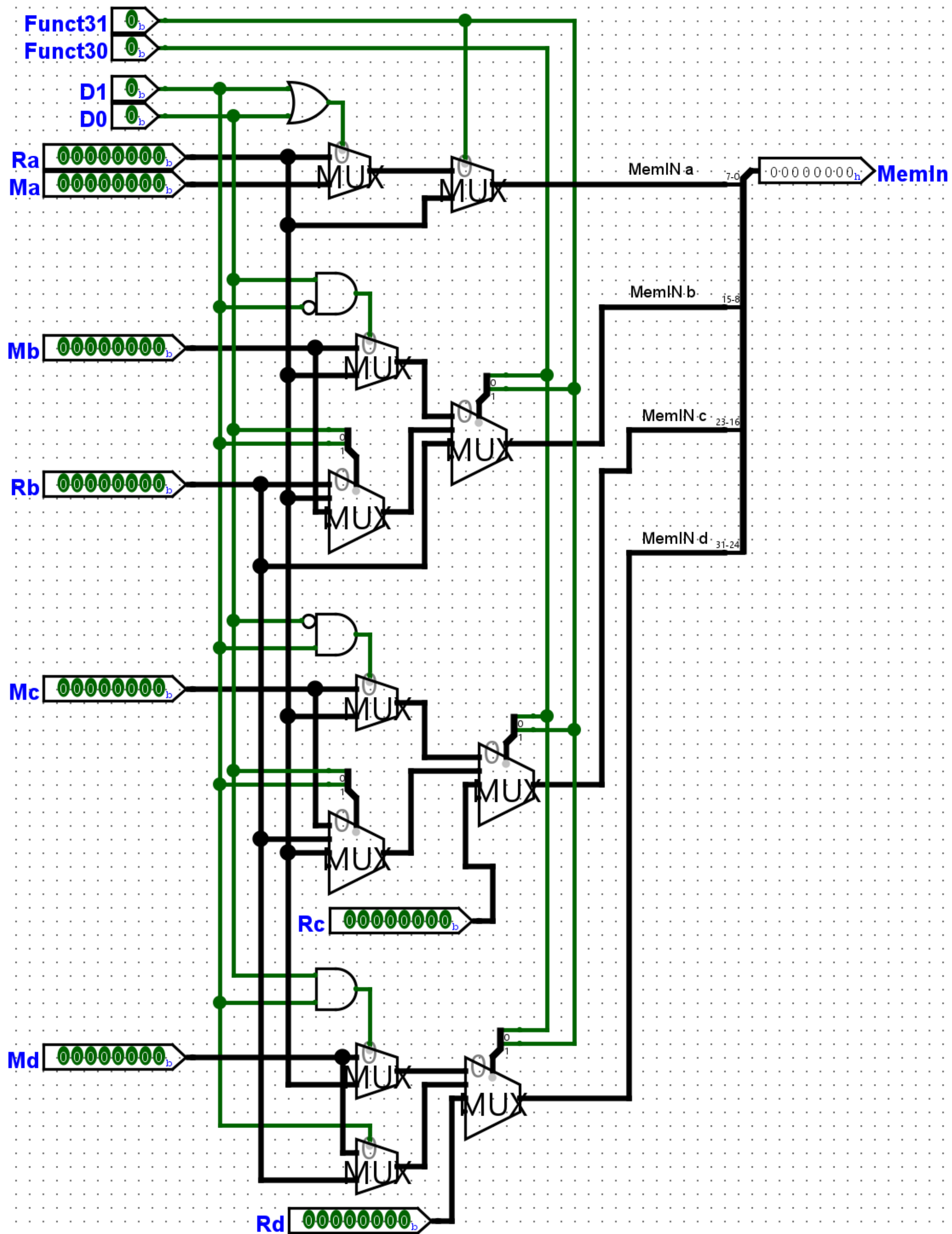


Figura 3.22: Subcircuito basado en la tabla de verdad para las instrucciones de tipo S.

Partiendo de la tabla de verdad, se ha desarrollado un circuito a base de multiplexores. Cada uno de los cuatro bytes que forman la palabra entrante a la memoria de datos sigue el mismo esquema de selección: dos multiplexores controlados por D0 y D1 cuyas salidas llegan a un tercer multiplexor que depende de los valores del campo *funct3*. Tras simplificar el circuito resultante, el circuito final es el que se puede ver en la Figura 3.22.

3.5. Instrucciones de tipo B

Las instrucciones de tipo B en la arquitectura RISC-V son utilizadas principalmente para operaciones de salto condicional, esenciales para controlar el flujo de ejecución en programas. Estas instrucciones permiten que el procesador tome decisiones basadas en comparaciones entre registros y, en función del resultado, puede saltar a otra parte del código. Este tipo de instrucciones es crucial para la implementación de estructuras de control como bucles y condicionales (*if-else*), permitiendo que el programa ejecute diferentes secciones de código dependiendo de las condiciones evaluadas en tiempo de ejecución.

El formato de las instrucciones de tipo B en RISC-V incluye varios campos específicos que determinan su operación. Estas instrucciones están compuestas por el campo *opcode* de 7 bits que indica el tipo de operación a realizar, seguido por los campos *rs1* y *rs2* que especifican los registros fuente utilizados para la comparación. El campo *func3* determina la condición específica bajo la cual se realizará el salto (por ejemplo, igualdad, menor que, etc.). Finalmente, el campo *imm* (inmediato) se utiliza para determinar el desplazamiento del salto si la condición es verdadera. Este campo *imm* está dividido en varios bits distribuidos a lo largo de la instrucción, lo que permite un direccionamiento más flexible y eficiente dentro del código.

El comando en código ensamblador para ejecutar una instrucción de tipo B necesita de dos registros fuente y un valor inmediato para calcular la dirección del salto, que puede ser hacia delante o hacia atrás. El valor inmediato, se puede representar como una etiqueta. A continuación, se muestra el código correspondiente a la instrucción de salto en caso de que los valores a comparar sean iguales:
beq *rs1*, *rs2*, *imm* $\rightarrow if(rs1 = rs2) \Rightarrow Memoria\ de\ programa[PC + imm]$.

3.5.1. Arquitectura necesaria

Para implementar las instrucciones de tipo B en la arquitectura RISC-V, se requieren varios componentes adicionales más allá de los ya presentes en el circuito. En el subcircuito de la unidad aritmético-lógica (ALU), se incorporan dos comparadores y varias puertas lógicas, junto con un multiplexor, para facilitar la evaluación de condiciones. Adicionalmente, en el circuito principal se necesita implementar un sumador para calcular la dirección de salto, una puerta lógica AND y un multiplexor adicional que determina la dirección efectiva de la siguiente instrucción a ejecutar.

Una vez sabemos los componentes adicionales necesarios, vemos el recorrido de las instrucciones de tipo B. Estas salen de la memoria de programa, atraviesan el banco

de registros del que obtienen los valores a comparar y realizan dicha comparación en la ALU. La salida de la comparación es una de las entradas de la puerta adicional AND, cuya otra entrada es controlada por la unidad de control. La salida de esta puerta AND influye directamente en el multiplexor que determina la dirección de la próxima instrucción a ejecutar, basándose en el resultado de la comparación y las señales de control, modificando así el flujo normal de ejecución del programa si la condición se cumple.

3.5.2. Implementación en Logisim-Evolution

Para la ejecución de las instrucciones de tipo B es necesario realizar modificaciones en la ALU, donde se realiza la comparación. Además, se deben realizar ajustes en la ruta que conduce a la entrada del contador de programa (PC). Esto es debido a que, en caso de que la comparación resulte verdadera, la próxima instrucción a ejecutar no seguirá el flujo secuencial habitual. En su lugar, se seleccionará una nueva instrucción ubicada en una posición específica de la memoria de programa. Esta posición se determina sumando el valor inmediato, proporcionado por el campo *imm* de la instrucción, al valor actual del contador de programa.

3.5.2.1 Unidad Aritmético-Lógica (ALU)

Hay un total de seis instrucciones de comparación (Fig. 3.23) de las cuales cada una tiene su opuesta dentro de este mismo grupo. Para facilitar su ejecución, se han implementado dos comparadores en el diseño del circuito. Uno de ellos se encarga de las comparaciones de números en complemento a dos, que son aquellas que consideran el signo, mientras que el otro comparador realiza comparaciones sin tenerlo en cuenta.

Branches	Branch =	B	BEQ
	Branch ≠	B	BNE
	Branch <	B	BLT
	Branch ≥	B	BGE
	Branch < Unsigned	B	BLTU
	Branch ≥ Unsigned	B	BGEU

Figura 3.23: Instrucciones de tipo B con sus correspondientes operaciones de comparación.

A la salida de los bloques comparadores se disponen las puertas lógicas necesarias para construir las operaciones de comparación a realizar (Fig. 3.24). El valor del resultado de la comparación es seleccionado a través del multiplexor y sacado como otra salida a mayores de la ya existente en la ALU, ya que se trata de una salida de un solo bit (*Branch*) y no de 32 bits.

Para optimizar el rendimiento y reducir los costos computacionales y los posibles retrasos, la salida de cada operación de comparación se une con la entrada del multiplexor equivalente al valor decimal almacenado por el campo *funct3* de cada una de las instrucciones. La razón de esta configuración es simplificar la generación de la señal de

control para el multiplexor dentro de la unidad de control, haciendo que esta señal sea directamente equivalente al valor del campo *funct3*.

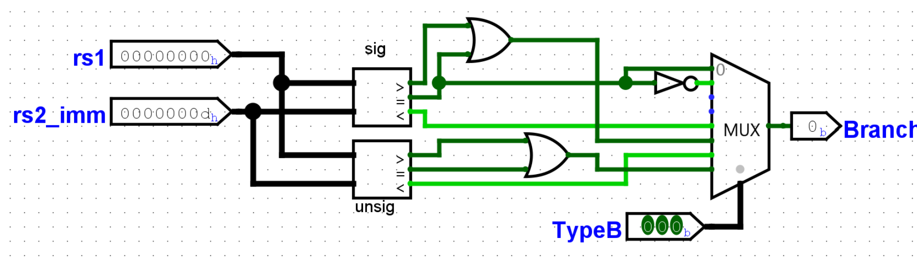


Figura 3.24: Bloques de comparación añadidos al subcircuito de la ALU.

3.5.2.2 Circuito principal

Los resultados de las comparaciones definen la dirección de la siguiente instrucción a ejecutar, por lo que, los cambios en el circuito se realizan a la entrada del bloque contador de programa.

Las dos posibilidades son: que se transfiera la dirección de la siguiente instrucción o que se ejecute otra instrucción cuya dirección es calculada con otro valor diferente a 4 (por los 4 bytes que ocupa una instrucción). De esta forma, en la Figura 3.25 se puede apreciar un multiplexor con dos entradas, una para cada una de las opciones.

La salida del multiplexor elegida cuando el valor de selección es un 0 lógico corresponde a la dirección de la instrucción inmediatamente posterior a la que se acaba de ejecutar. Esta es obtenida del valor del contador del programa en el momento actual más 4 bytes. Por otro lado, cuando el valor de selección es 1, la dirección de la siguiente instrucción a ejecutar es la resultante de sumar el valor inmediato extendido en signo y el valor almacenado por el PC en el momento de ejecución.

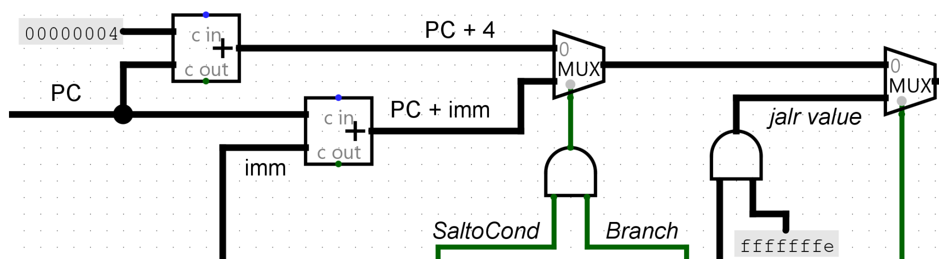


Figura 3.25: Cambios a la entrada del bloque contador de programa (PC).

Cabe destacar que la señal de selección del multiplexor se controla mediante una puerta lógica AND. Esta puerta tiene como entradas la señal *SaltoCond*, la cual es activada por la unidad de control cuando se ejecuta una instrucción de tipo B, y la señal *Branch*, que se activa cuando la comparación correspondiente se cumple. De esta forma, solo cuando las dos señales están activas a la vez se realiza un salto en el programa en ejecución.

3.6. Instrucciones de tipo U

Las instrucciones de tipo U en la arquitectura RISC-V son utilizadas para operaciones que involucran un valor inmediato de 20 bits que se carga directamente en un registro. Este tipo de instrucción es esencial para configurar registros con valores altos que no caben en un inmediato más pequeño, típicamente usado en otras instrucciones. Por ejemplo, se utilizan para establecer los bits superiores de un registro mientras se configura la máquina o se realizan cálculos que requieren constantes grandes [29].

El formato de las instrucciones de tipo U en RISC-V es relativamente simple y consta de pocos campos. Estas instrucciones incluyen un campo de *opcode* de 7 bits que especifica el tipo de instrucción a realizar. El campo inmediato (*imm*) de 20 bits se utiliza para proporcionar el valor constante que se cargará en el registro. No hay campos *funct3*, *rs1* o *rs2* en las instrucciones de tipo U, ya que estas instrucciones no realizan operaciones entre registros o no requieren una función adicional especificada por *funct3*. El campo *rd* de 5 bits especifica el registro destino donde se almacenará el valor inmediato. Este diseño simplificado facilita la carga de valores grandes directamente en los registros sin intervención de otros registros o cálculos adicionales.

El comando en código ensamblador para ejecutar este tipo de instrucción es también muy simple. Se muestra a continuación el código correspondiente a la instrucción de carga en los 20 dígitos más significativos: **lui** *rd*, $imm \rightarrow rd = imm$.

3.6.1. Arquitectura necesaria

El recorrido de las instrucciones de tipo U por la arquitectura RISC-V es bastante escueta. Únicamente utiliza el subcircuito generador del dato inmediato, el banco de registros donde almacena el dato en la dirección indicada por *rd* y un multiplexor donde se selecciona el dato de entrada al banco de registros.

3.6.2. Implementación en Logisim-Evolution

Algunas de las instrucciones previamente vistas a las instrucciones de tipo U, como las de tipo I y R también devuelven un valor al registro destino del banco de registros. La decisión de qué valor almacenar en dicho registro es tomada por unos multiplexores explicados en la Subsección 3.3.2.3.

Dentro de las instrucciones de tipo U existe una diferencia en el dato a almacenar. La instrucción **lui** carga en los 20 dígitos más significativos del registro destino los 20 dígitos del dato inmediato contenidos en la instrucción, rellenando los dígitos menos significativos con ceros. Por otro lado, la instrucción **auipc** carga en el registro destino el valor resultante de sumar el dato inmediato generado más la dirección de la propia instrucción contenida en el contador de programa.

Debido a estas dos nuevas instrucciones, el total de multiplexores para seleccionar cuál de todos los valores se almacena en el registro destino pasa de dos a cuatro (Fig. 3.26). Cada uno de ellos es controlado por una señal de control generada en el subcircuito

Unidad de control.

Esta solución presentada, aunque es correcta, no es muy efectiva debido al costo computacional por la generación de más señales de control y es más lenta debido a los retardos de propagación acumulados que generan cuatro multiplexores en serie. La solución propuesta, y definitiva, es sustituir los cuatro multiplexores de dos entradas por un multiplexor de, al menos, cinco entradas.

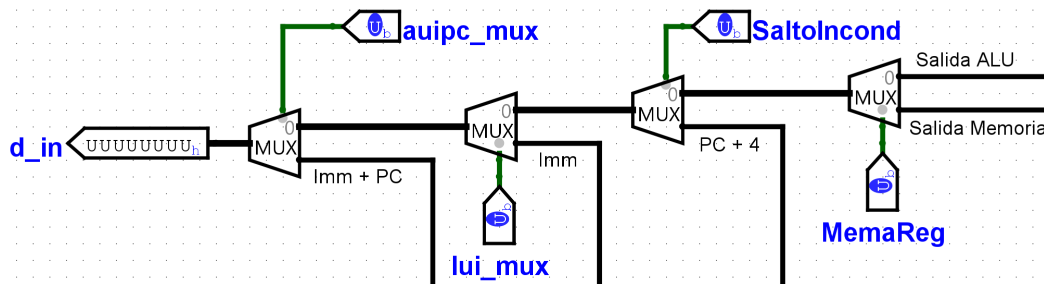


Figura 3.26: Selección del dato de entrada al banco de registros con cuatro multiplexores en serie.

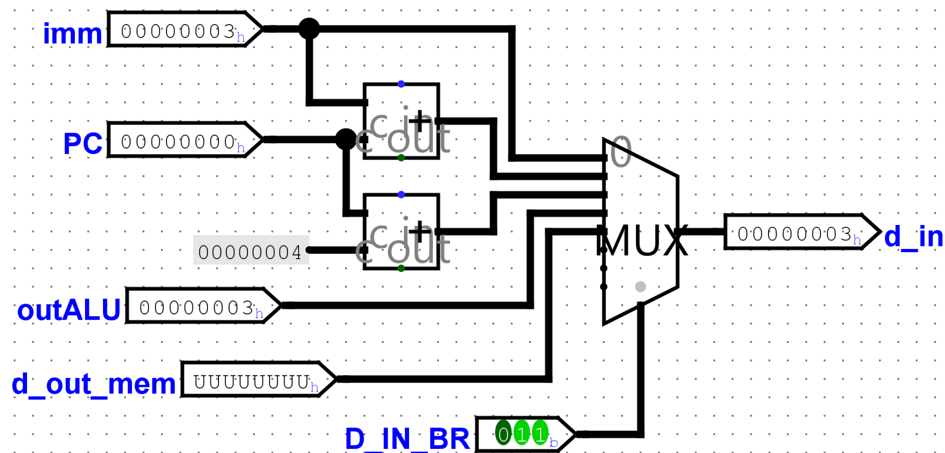
Este multiplexor requiere de una señal de control de tres bits, *D_IN_BR*, generada en la unidad de control que sustituye las señales de control para los cuatro multiplexores. En la Figura 3.27 se pueden ver las diferentes posibles entradas al registro destino según la instrucción en ejecución. De esta forma, el único retardo de propagación que sufre esta parte del circuito es el tiempo que tarda el multiplexor en responder a un cambio en la señal de control.

3.7. Instrucciones de tipo J

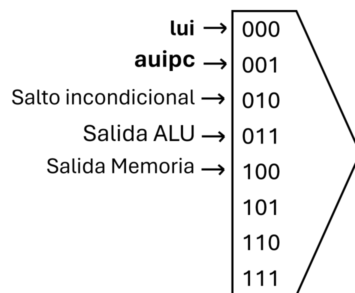
Dentro de las instrucciones de tipo J en la arquitectura RISC-V solo existe la instrucción **jal** (*jump and link*). Esta es utilizada para realizar saltos incondicionales a una dirección específica dentro del programa. Esta nueva dirección se calcula sumando un valor inmediato al contador de programa (PC). Además, esta instrucción guarda la dirección de la siguiente instrucción en un registro (generalmente x1 o ra), lo que permite regresar al punto de ejecución original después de completar la función llamada. Esto es esencial para la implementación de llamadas a funciones y la realización de saltos largos dentro de un programa, facilitando así la estructuración del código.

El formato de las instrucciones de tipo J en RISC-V se caracteriza por tener un campo de *opcode* de 7 bits que identifica la operación como un salto, un campo de 20 bits inmediato (*imm*) que se utiliza para calcular la dirección de salto relativa al contador de programa actual, y un campo *rd* de 5 bits que especifica el registro donde se almacenará la dirección de retorno. No se utilizan los campos *func3*, *rs1*, ni *rs2* en las instrucciones de tipo J, ya que estas instrucciones no realizan operaciones entre registros ni requieren una función adicional especificada por *func3*.

El comando en código ensamblador para ejecutar esta instrucción es prácticamente



(a) Circuito del multiplexor a la entrada del banco de registros.



(b) Esquema del uso de cada una de las entradas del multiplexor.

Figura 3.27: Multiplexor selector del dato de entrada al banco de registros.

idéntico al visto en la sección anterior. A continuación, se muestra el código correspondiente a la instrucción de salto incondicional:
jal rd, $imm \rightarrow rd = PC + 4$ & $Memoria\ de\ programa[PC + imm_{x2}]$

3.7.1. Arquitectura necesaria

La instrucción de salto incondicional de tipo J para su ejecución hace uso del subcircuito generador del dato inmediato. En este subcircuito, además de extender en signo el valor almacenado en *imm* es multiplicado por dos (Subsec. 3.1.3). También hace uso del banco de registros, donde almacena, en la dirección indicada en *rd*, el valor del contador de programa más cuatro unidades, es decir, la siguiente instrucción.

Por último, para transferir el valor correcto de la dirección de salto al contador de programa se utiliza un multiplexor de cuatro entradas controlado por la señal de control *SaltoIncond*. Esto es debido a que la instrucción **jalr** de tipo I, al ser otra instrucción de salto incondicional, también se controla con esa señal. Por lo tanto hay dos entradas para dos instrucciones de tipo salto incondicional y una tercera para otros tipos de instrucciones.

3.7.2. Implementación en Logisim-Evolution

Para implementar la parte del salto en la memoria de programa de la instrucción se utiliza el multiplexor de cuatro entradas mostrado en la Figura 3.28. Este multiplexor está situado en serie con el multiplexor de dos entradas utilizado para la realización de las instrucciones de tipo B. La salida del multiplexor de dos entradas es recibida por la primera entrada del multiplexor de cuatro entradas. La segunda entrada recibe la dirección de salto para la instrucción *jal* y la tercera recibe la dirección de salto calculada para el caso de la otra instrucción de salto incondicional de la arquitectura RV32I, **jalr** (Subsec. 3.3.2.3).

La señal de control *SaltoIncond* generada en el subcircuito Unidad de control se encarga de seleccionar el valor de salida del multiplexor dependiendo de la instrucción en ejecución.

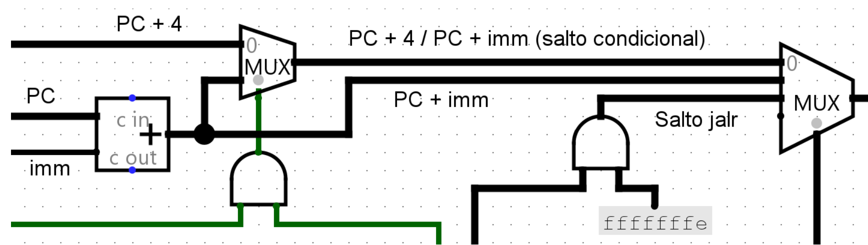


Figura 3.28: Selección de la dirección de memoria entrante al contador de programa.

Por otro lado, la parte de la instrucción que se encarga de almacenar el valor de la dirección de la siguiente instrucción, el valor del contador de programa más cuatro, en el registro destino ya ha sido implementada. La instrucción *jalr* también almacenaba dicho valor en el registro destino. El multiplexor selector del dato de entrada al banco de registros se explica en la Sección 3.6. Las únicas modificaciones realizadas en este multiplexor son las que conciernen a la señal de selección *D_IN_BR* generada en el subcircuito Unidad de control. Esta tiene que seleccionar como salida del multiplexor la entrada correspondiente al salto incondicional para esta instrucción además de para la instrucción de tipo I **jalr**.

Capítulo 4

Comprobación de la implementación

En este capítulo, nos enfocamos en la verificación y demostración del correcto funcionamiento de la arquitectura del procesador implementada en el capítulo anterior utilizando la aplicación Logisim-Evolution. Tras haber completado la implementación de todas las instrucciones que conforman el conjunto de la ISA RV32I, salvo las que indicamos en la Subsección 1.4, es crucial asegurar que cada componente del procesador opera de la forma esperada. Este proceso de verificación es esencial para identificar y corregir posibles errores en el diseño antes de proceder a etapas más avanzadas de desarrollo, en caso de haberlas.

Para llevar a cabo esta verificación, se realizan comprobaciones por bloques de instrucciones. Cada bloque agrupa instrucciones del mismo tipo o aquellas que requieren el uso de las mismas partes de la arquitectura. Por ejemplo, las instrucciones de carga y almacenamiento son verificadas conjuntamente, ya que ambas interactúan con el banco de registros y la memoria. Este enfoque permite una evaluación detallada y precisa de cada componente del procesador, asegurando que todas las partes funcionen de manera correcta.

Cada bloque de instrucciones es sometido a pruebas exhaustivas mediante el desarrollo de secuencias de instrucciones específicas que contemplen diversas alternativas para cada operación. Estas secuencias son diseñadas para cubrir una amplia gama de escenarios posibles, garantizando así una verificación completa y robusta del funcionamiento del procesador. Además, se utiliza la herramienta auxiliar Ripes, una aplicación que simula el comportamiento de un procesador RISC-V, para ejecutar programas de manera paralela en ambas plataformas. Esta comparación permite validar el correcto funcionamiento del circuito en Logisim-Evolution al contrastar los resultados obtenidos con los de Ripes [28].

Los programas de prueba diseñados para cada comprobación están escritos en código ensamblador, y para la escritura de cada instrucción es necesario indicar los distintos registros a utilizar. A lo largo de las diversas comprobaciones, se encuentran los registros mencionados de distinta forma. Cuando el registro se indica mediante una x seguida de un número, este número señala cuál de los 32 registros que forman el banco de registros es. En cambio, si se indica de otra forma, la intención es destacar si es un

registro temporal o un registro guardado. La relación entre ambas nomenclaturas se puede consultar en la Sección 1.2.

4.1. Comprobación de instrucciones que realizan operaciones básicas

La primera sección del capítulo dedicado a realizar comprobaciones se enfoca en todas las operaciones cuya función principal es ejecutar alguna operación lógica, aritmética, de comparación o de desplazamiento, centrándose así, casi exclusivamente en la unidad aritmético-lógica (ALU) y en el banco de registros. En este último componente, se verifica que se pueda leer y escribir en los registros que lo forman a excepción del registro cero que tiene un valor constante de cero. Estas operaciones son fundamentales, ya que también se emplean en la ejecución de otras instrucciones, por lo que es esencial verificar su correcto funcionamiento en primer lugar.

En consecuencia, las instrucciones a verificar en esta sección incluyen todas las instrucciones de tipo R y las instrucciones de tipo I que realizan instrucciones paralelas a las de tipo R. La principal diferencia entre ambos tipos de instrucciones es que las de tipo I utilizan un valor inmediato como uno de los dos operandos de la operación.

4.1.1. Realización y ejecución del programa de verificación

El programa diseñado ejecuta dos veces cada tipo de instrucción, una de ellas con dos números positivos como operandos y otra con uno positivo y otro negativo para probar todas las combinaciones. Para simplificar la realización del programa se han utilizado los tres mismos números para todas las instrucciones.

El programa comienza inicializando los registros de los cuales se van a obtener los operandos mediante tres instrucciones **addi**. Tras estas, se desarrollan todas las instrucciones de tipo R y tipo I a comprobar.

```
1  addi x5, x0, 2      # x5 = 2
2  addi x7, x0, -7     # x7 = -7
3  addi x28, x0, 20    # x28 = 20
4  add x10, x5, x28    # x10 = x5 + x28 = 2 + 20 = 22
5  add x11, x5, x7     # x11 = x5 + x7 = 2 + (-7) = -5
6  sub x12, x28, x5    # x12 = x28 - x5 = 20 - 2 = 18
7  sub x13, x5, x7     # x13 = x5 - x7 = 2 - (-7) = 9
8  sll x14, x5, x28    # x14 = x5 << x28 = 2 << 20
9  sll x15, x7, x5     # x15 = x7 << x5 = (-7) << 2
10 slt x16, x5, x28    # x16 = (x5 < x28) = (2 < 20) = 1
11 slt x17, x5, x7     # x17 = (x5 < x7) = (2 < -7) = 0
12 sltu x18, x5, x28   # x18 = (x5 < x28) = (2 < 20) = 1
13 sltu x19, x5, x7    # x19 = (x5 < x7) = (2 < -7 u) = 1
14 xor x20, x5, x28    # x20 = x5 ^ x28 = 2 ^ 20
15 xor x21, x5, x7     # x21 = x5 ^ x7 = 2 ^ (-7)
16 srl x22, x28, x5    # x22 = x28 >> x5 = 20 >> 2 = 5
17 srl x23, x7, x5     # x23 = x7 >> x5 = (-7) >> 2
18 sra x24, x28, x5    # x24 = x28 >> x5 = 20 >> 2 = 5
```

```
19 sra x25, x7, x5    # x25 = x7 >> x5 = -7 >> 2
20 or x26, x5, x28    # x26 = x5 | x28 = 2 | 20
21 or x27, x5, x7     # x27 = x5 | x7 = 2 | (-7)
22 and x29, x5, x28   # x28 = x5 & x28 = 2 & 20
23 and x30, x5, x7    # x29 = x5 & x7 = 2 & (-7)
24 addi x10, x5, 20   # x10 = x5 + 20 = 2 + 20 = 22
25 addi x11, x5, -7   # x11 = x5 + (-7) = 2 - 7 = -5
26 slli x12, x5, 2    # x12 = x5 << 2 = 2 << 2 = 8
27 slli x13, x7, 1    # x13 = x7 << 1 = -7 << 1
28 slti x14, x5, 20   # x14 = (x5 < 20) = (2 < 20) = 1
29 slti x15, x5, -7   # x15 = (x5 < -7) = (2 < -7) = 0
30 sltiu x16, x5, 20 # x16 = (x5 < 20) = (2 < 20) = 1
31 sltiu x17, x5, -7 # x17 = (x5 < -7) (2 < -7 u) = 1
32 xori x18, x5, 20   # x18 = x5 ^ 20 = 2 ^ 20
33 xori x19, x5, -7   # x19 = x5 ^ (-7) = 2 ^ (-7)
34 srli x20, x28, 2   # x20 = x28 >> 2 = 20 >> 2 = 5
35 srli x21, x7, 1    # x21 = x7 >> 1 = 2 >> 1 = 1
36 srai x22, x28, 2   # x22 = x28 >> 2 = 20 >> 2 = 5
37 srai x23, x7, 1    # x23 = x7 >> 1 = -7 >> 1
38 ori x24, x5, 20    # x24 = x5 | 20 = 2 | 20
39 ori x25, x5, -7    # x25 = x5 | (-7) = 2 | (-7)
40 andi x26, x5, 20   # x26 = x5 & 20 = 2 & 20
41 andi x27, x5, -7   # x27 = x5 & (-7) = 2 & (-7)
```

Código 4.1: Programa de prueba para las operaciones básicas.

La ejecución del bloque de instrucciones en la aplicación Logisim-Evolution se realiza en paralelo con la ejecución del programa en la aplicación Ripes. Este permite realizar comprobaciones de manera más rápida y eficaz, asegurando así el correcto funcionamiento de las operaciones básicas que realiza el procesador.

4.2. Comprobación de instrucciones de carga y almacenamiento

La segunda sección se dedica a las instrucciones que se centran en el uso de la memoria de datos. Estas incluyen las instrucciones de carga (*load*), de tipo I, y las de almacenamiento (*store*), de tipo S. Ambos tipos de instrucciones pueden realizar múltiples combinaciones con los datos, lo que hace necesario llevar a cabo una comprobación exhaustiva de cada una de ellas. Además, las instrucciones de tipo *load* también manejan números negativos, por lo que es crucial verificar que esta operación se realice correctamente.

4.2.1. Realización y ejecución del programa de verificación

El programa a ejecutar para la comprobación del funcionamiento del circuito precisa del uso de alguna de las operaciones básicas probadas anteriormente. Estas operaciones se realizan al principio del programa (Código 4.2) para almacenar en el banco de registros el dato a manipular con las instrucciones *load* y *store*. Además, también se almacenan distintas direcciones de memoria consecutivas que se van a emplear en el programa.

```
1 addi t1, zero, 0x518
```

```
2  slli t1, t1, 16
3  addi t1, t1, 0x782       # Dato a manipular por las load y store
4  addi s0, zero, 0x5C2     # Calculo de la primera direccion de memoria
5  add s0, s0, s0
6  add s0, s0, s0
7  addi s1, s0, 4           # Segunda direccion de memoria
8  addi s2, s1, 4           # Tercera direccion de memoria
9  addi s3, s2, 4           # Cuarta direccion de memoria
10 addi s4, s3, 4           # Quinta direccion de memoria
11 addi s5, s4, 4           # Sexta direccion de memoria
12 addi s6, s5, 4           # Septima direccion de memoria
13 addi s7, s6, 4           # Octaba direccion de memoria
```

Código 4.2: Programa de prueba para las operaciones de carga y almacenamiento.

A continuación se procede a probar el conjunto de instrucciones *store*. Esta sección del programa emplea las diferentes direcciones de memoria definidas antes para almacenar en cada una de ellas el byte o bytes correspondientes al dato almacenado en t1. Una vez se ha comprobado que el direccionamiento se realiza de forma correcta se comprueba si hay algún problema en caso de que el offset de la instrucción sea negativo.

```
14 sw t1, 0x000, s0        # Almacena plabra, media palabra
15 sb t1, 0x000, s1        # o un byte de t1 en las direcciones
16 sb t1, 0x001, s2        # de memoria definidas en s0 - s7
17 sb t1, 0x002, s3
18 sb t1, 0x003, s4
19 sh t1, 0x000, s5
20 sh t1, 0x001, s6
21 sh t1, 0x002, s7
22 sh t1, -4, s5
```

Código 4.3: Programa de prueba para las operaciones de carga y almacenamiento.

La siguiente sección del programa se centra en verificar la correcta ejecución de las instrucciones *load*. Inicialmente carga la palabra completa y luego procede a realizar la instrucción **lb** de dos formas distintas. La primera vez que se ejecuta, accede a cada byte de la palabra que almacena una dirección concreta de la memoria. La siguiente vez, accede a las diferentes direcciones de memoria empleadas anteriormente por las instrucciones *store* y carga en el banco de registros el valor que estas habían almacenado anteriormente. Después, ejecuta la instrucción **lh** y, tras asegurar que el acceso a las direcciones de memoria se realiza sin ningún problema, se comprueba que el uso de un número negativo como offset en la instrucción no altera el comportamiento del procesador.

```
23 lw t2, 0x000, s0        # Carga palabra almacenada en s0 en t2
24 lb t3, 0x000, s0        # Carga byte almacenado en s0 en t3
25 lb t4, 0x001, s0        # Carga byte almacenado en s0+1 en t4
26 lb t5, 0x002, s0
27 lb t6, 0x003, s0
28 lb a1, 0x000, s1        # Carga byte almacenado en s1 en a1
29 lb a2, 0x001, s2        # Carga byte almacenado en s2+1 en a2
```

```
30 lb a3, 0x002, s3
31 lb a4, 0x003, s4
32 lh t3, 0x000, s0          # Carga media palabra almacenada en s0
33 lh t4, 0x001, s0
34 lh t5, 0x002, s0
35 lh a5, -4, s1
```

Código 4.4: Programa de prueba para las operaciones de carga y almacenamiento.

Finalmente, se ejecutan las instrucciones de carga que no tienen en cuenta el signo del dato (**lbu** y **lhu**). Esta parte del programa repite prácticamente la misma secuencia que las operaciones que sí consideran el signo, permitiendo observar las diferencias entre estas instrucciones y comprobar su funcionamiento.

```
36 lbu t3, 0x000, s0         #Instrucciones de carga sin signo
37 lbu t4, 0x001, s0
38 lbu t5, 0x002, s0
39 lbu t6, 0x003, s0
40 lhu t3, 0x000, s0
41 lhu t4, 0x001, s0
42 lhu t5, 0x002, s0
```

Código 4.5: Programa de prueba para las operaciones de carga y almacenamiento.

4.3. Comprobación de saltos condicionales en la memoria de programa

Las siguientes instrucciones a comprobar comprenden todas las instrucciones de tipo *branch*. Por lo que la principal intención de esta sección es asegurar que se realizan correctamente las comparaciones y su consecuente salto en la memoria de programa.

4.3.1. Realización y ejecución del programa de verificación

Para comprobar las instrucciones de tipo *branch* se realiza un pequeño programa para cada una de ellas. De esta forma, se asegura un análisis más profundo y se evita la realización de un programa más extenso y complejo. Estas instrucciones sirven para realizar estructuras de control, como bucles, por lo que la estructura de los distintos programas es muy similar entre sí.

Se comienza la verificación del funcionamiento con la instrucción **beq**. Para esta se realiza un pequeño programa con un bucle que se ejecuta dos veces hasta finalizar. De esta forma se revisa la comparación y la acción que desencadena en caso de que la comparación se cumpla o no.

```
1 addi s0, zero, 2           # Inicializacion de las variables
2 addi s1, zero, 0
3 FOR:                       # Inicio del bucle
4     beq s0, s1, FIN
```

```
5     addi s1, s1, 1
6     beq zero, zero, FOR   # Vuelta al punto inicial del bucle
7     FIN:
8         beq s0, s1, FIN   # Fin del programa
```

Código 4.6: Programa de prueba para la instrucción **beq**.

La siguiente instrucción a examinar (**bne**) realiza la comparación opuesta a la anterior, por lo que lo único que cambia con respecto al programa previo es la lógica del bucle.

```
1 addi s0, zero, 2          # Inicializacion de las variables
2 addi s1, zero, 2
3 FOR:                      # Inicio del bucle
4     bne s0, s1, FIN
5     addi s1, s1, 1
6     beq zero, zero, FOR   # Vuelta al punto inicial del bucle
7     FIN:
8         bne s0, s1, FIN   # Fin del programa
```

Código 4.7: Programa de prueba para la instrucción **bne**.

El siguiente programa, en el que se emplea la instrucción **blt**, se ejecuta el bucle tres veces en los que el valor almacenado en s1 es menor e igual que el valor almacenado en s0 antes de saltar a la instrucción final. De esta forma se asegura una correcta ejecución de la comparación. También se ejecuta el mismo programa sustituyendo la instrucción **blt** por **bltu**. Esto provoca que el procesador no ejecute la parte correspondiente al bucle principal debido a que el valor almacenado en s1, sin tener en cuenta el signo, es mucho mayor que el valor almacenado en s0.

```
1 addi s0, zero, 2          # Inicializacion de las variables
2 addi s1, zero, -2
3 FOR:                      # Inicio del bucle
4     blt s0, s1, FIN
5     addi s1, s1, 2
6     beq zero, zero, FOR   # Vuelta al punto inicial del bucle
7     FIN:
8         blt s0, s1, FIN   # Fin del programa
```

Código 4.8: Programa de prueba para las instrucciones **blt** y **bltu**.

Para comprobar las dos instrucciones de salto condicional restantes se cambia el orden de los operandos de la comparación, como se muestra en el código. En este caso, cuando se ejecuta empleando la instrucción **bge**, el bucle principal se realiza dos veces hasta que ambos valores son iguales y salta a la instrucción final. Por el contrario, si se emplea la instrucción **bgeu** el bucle principal no se ejecuta debido a la misma razón expuesta anteriormente.

```
1 addi s0, zero, 2          # Inicializacion de las variables
2 addi s1, zero, -2
3 FOR:                      # Inicio del bucle
```

```
4       bge s1, s0, FIN
5       addi s1, s1, 2
6       beq zero, zero, FOR   # Vuelta al punto inicial del bucle
7       FIN:
8           bge s1, s0, FIN   # Fin del programa
```

Código 4.9: Programa de prueba para las instrucciones **bge** y **bgeu**.

4.4. Comprobación de saltos incondicionales en la memoria de programa

Las instrucciones de salto incondicional, al igual que las instrucciones *branch* también realizan saltos en la memoria pero, en este caso, no precisan que se cumpla una condición previa para realizar dicho salto. En esta sección se comprueba que las dos instrucciones restantes realizan de forma correcta los saltos en la memoria de programa.

4.4.1. Realización y ejecución del programa de verificación

Para la comprobación se emplean programas muy similares. En el caso de la ejecución del programa para la instrucción **jal**, se verifica que el banco de registros almacena el valor del contador de programa más cuatro unidades al registro definido. Además, ejecuta el salto en la memoria realizando el bucle definido.

```
1 addi t0, zero, 3        # t0 = 3 (numero de iteraciones)
2 addi t1, zero, 0        # t1 = 0 (contador de bucle)
3 FOR:
4     addi t1, t1, 1        # Incremento del contador del bucle
5     bge t1, t0, FIN       # Si t1 >= t0, salto a FIN
6     jal t2, FOR        # Salto incondicional a FOR
7 FIN:
8     beq zero, zero, FIN
```

Código 4.10: Programa de prueba para la instrucción **jal**.

Por otro lado, la instrucción **jalr** almacena el valor del contador de programa más cuatro unidades en el registro especificado y calcula la dirección de la siguiente instrucción a ejecutar. Esta dirección se determina sumando en la ALU el valor del inmediato generado y el valor almacenado en otro registro. En el programa diseñado, la dirección apunta a una subrutina que realiza la suma de los valores almacenados en dos registros. Tras su finalización, el comando *ret* retorna de la subrutina utilizando la dirección almacenada en *x1*, continuando así con el bucle en ejecución.

```
1 addi s0, zero, 3        # Inicializa s0 con el valor 3
2 addi a1, a1, 2
3 FOR:
4     beq s0, zero, FIN    # Si s0 es cero, salta a FIN
5     addi s0, s0, -1      # Decrementa s0
6     jalr x1, zero, SUMA # Salta a la subrutina SUMA
7     beq zero, zero, FOR # Salta a FOR
```

```
8
9  # Subrutina SUMA
10 SUMA:
11     add a0, a0, a1       # Suma a0 y a1, guarda en a0
12     ret                  # Retorna de la subrutina
13
14 FIN:
15     beq zero, zero, FIN # Bucle infinito
```

Código 4.11: Programa de prueba para la instrucción **jalr**.

4.5. Comprobación de instrucciones de tipo U

Las dos instrucciones restantes de comprobar son esenciales para manejar constantes grandes y direcciones relativas al PC en la arquitectura RISC-V. Cuando se trabaja con direcciones absolutas en memoria, **lui** es preferible, mientras que **auipc** es más útil cuando se necesita calcular direcciones relativas al contador de programa.

4.5.1. Realización y ejecución del programa de verificación

El inmediato de las instrucciones de tipo U en la arquitectura RISC-V no puede ser negativo. Este inmediato de 20 bits se interpreta como un valor sin signo, lo que significa que siempre es positivo o cero [29]. Por ello, el programa a ejecutar para la comprobación es muy sencillo, ya que no hay muchas variaciones posibles a realizar en estas instrucciones.

```
1 lui t0, 0x12345         # t0 = 0x12345000
2 auipc t1, 0x1           # t1 = PC + 0x1000 (PC actual + 0x1000)
3 addi t1, t1, 0x234      # t1 = t1 + 0x234
4 srli t1, t1, 3
5 sw t0, 0(t1)            # Mem[t1] = t0
```

Código 4.12: Programa de prueba para las instrucciones de tipo U **lui** y **auipc**.

Este programa simulado en el circuito implementado inicialmente carga el valor inmediato en los bits superiores de un registro. A continuación, almacena en otro registro el valor del inmediato más el valor del contador del programa en el momento de ejecución, en este caso 4. Por último, ejecuta un par de instrucciones ya previamente testadas para almacenar en la memoria de datos el valor cargado previamente por la instrucción **lui** en el banco de registros.

Capítulo 5

Prácticas de laboratorio

En este capítulo, se presentan una serie de prácticas de laboratorio consistentes en la elaboración de circuitos digitales basados en la arquitectura RISC-V. Estas prácticas están orientadas a estudiantes de ingeniería y tienen como objetivo principal proporcionar una comprensión profunda y práctica de los componentes fundamentales de la arquitectura RISC-V. A través de la construcción y simulación de diferentes partes de la arquitectura, los estudiantes podrán adquirir habilidades esenciales en el diseño y análisis de sistemas digitales.

Las prácticas de laboratorio se estructuran de manera que cada sesión se enfoque en uno o varios componentes específicos de la arquitectura RISC-V, como el banco de registros, la unidad aritmético-lógica (ALU), y el generador del dato inmediato, entre otros. En algunos casos, estos componentes se presentan de forma simplificada para facilitar el entendimiento de los conceptos básicos. Esta metodología permite a los estudiantes abordar cada elemento de manera individual, comprendiendo su funcionamiento y su interrelación con otros componentes de la arquitectura.

El objetivo final de estas prácticas es que, tras desarrollar y entender cada parte de la arquitectura de forma individual, los estudiantes sean capaces de integrar todos los circuitos en un único sistema cohesivo. Este sistema se acercará más a la arquitectura real del RISC-V, permitiendo la simulación de algunas instrucciones básicas. De esta manera, los estudiantes no solo aprenderán sobre el diseño de hardware, sino que también adquirirán una visión integral de cómo se construyen y operan los procesadores modernos.

Además, el proceso de realización de estos laboratorios está diseñado para ser progresivo. Los conocimientos y habilidades adquiridos en cada práctica serán fundamentales para el desarrollo de las siguientes, asegurando un aprendizaje continuo. Esta estructura no solo refuerza los conceptos aprendidos, sino que también prepara a los estudiantes para enfrentar desafíos más complejos en el ámbito de la electrónica digital.

5.1. Laboratorio 1

En esta sección, se propone la realización de dos circuitos fundamentales en la arquitectura de computadores: un banco de registros y una unidad aritmético-lógica (ALU). Estos componentes son esenciales para el funcionamiento de cualquier procesador y su comprensión es crucial para los estudiantes de ingeniería electrónica.

A través de la construcción y simulación de estos dos circuitos, los estudiantes podrán adquirir una comprensión práctica y más profunda de los componentes clave de la arquitectura RISC-V, sentando las bases para el diseño y análisis de sistemas más complejos.

5.1.1. Introducción al laboratorio

El primer circuito a desarrollar es un banco de registros, similar al empleado en la arquitectura RISC-V. En la arquitectura RISC-V estándar, el banco de registros consta de 32 registros de propósito general. Sin embargo, para facilitar el entendimiento y manejo por parte de los estudiantes, se ha simplificado a un banco de 8 registros. Este banco de registros permitirá almacenar y manipular datos de manera eficiente, replicando las operaciones básicas de lectura y escritura que se realizan en un procesador RISC-V real.

El segundo circuito es una unidad aritmético-lógica (ALU), que, como su nombre indica, se encarga de realizar operaciones aritméticas y lógicas. La ALU propuesta en este laboratorio será capaz de ejecutar tantas operaciones como las instrucciones de tipo R (Sec 3.2) requieran para su correcta ejecución. De esta forma se realizan una variedad amplia de operaciones y se obtiene la base necesaria para el resto de instrucciones de la arquitectura.

El objetivo de esta práctica es que el estudiante comprenda de manera individual el funcionamiento de cada circuito y adquiera la capacidad de conectarlos entre sí una vez completados. Esta integración permitirá al estudiante observar y analizar la interacción entre el banco de registros y la ALU, proporcionando una comprensión profunda de su funcionamiento conjunto. Este conocimiento es esencial, ya que sienta las bases para futuros laboratorios en los que se desarrollarán y se integrarán otras partes de la arquitectura, permitiendo una comprensión más completa y detallada de un procesador basado en la arquitectura RISC-V.

La extensión del laboratorio está diseñada para ser completada en sesiones de una hora y media, con un plazo límite de entrega de dos semanas desde la sesión inicial. Esta estructura permitirá a los estudiantes trabajar de manera progresiva y consolidar sus conocimientos a medida que avanzan en la implementación de estos circuitos.

5.1.2. Enunciado del laboratorio

En esta práctica de laboratorio, se abordan dos circuitos fundamentales: la implementación de un banco de registros y una unidad aritmético-lógica (ALU). Estos componentes son esenciales en el diseño y funcionamiento de microprocesadores modernos, y su com-

prensión es crucial para el desarrollo posterior del resto de laboratorios.

Ambos circuitos se deben implementar como subcircuitos del mismo programa y proporcionarles la apariencia indicada por el enunciado. La finalidad del laboratorio consiste en conectar ambos subcircuitos en la vista principal del programa y comprobar su correcta implementación.

- **Banco de Registros**

El primer circuito consiste en diseñar y construir un banco de registros paralelo al utilizado en la arquitectura RISC-V, pero en este caso formado por 8 registros de 32 bits cada uno. Los registros son componentes clave en la arquitectura de un procesador, ya que almacenan datos temporales y resultados intermedios de las operaciones realizadas por la ALU. El registro a utilizar en la implementación del circuito es el indicado en la Figura 5.1.

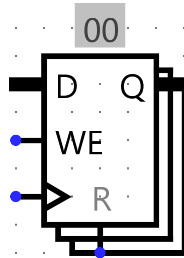


Figura 5.1: Registro de la biblioteca de elementos de Logisim-Evolution.

Para su implementación también es necesario emplear un demultiplexor, un decodificador y dos multiplexores de la biblioteca de 'Plexores' del programa.
El banco de registros debe cumplir con las siguientes especificaciones:

1. El banco de registros debe permitir la lectura de dos registros simultáneamente (uso de los multiplexores).

2. La escritura en un registro debe ser posible solo cuando la señal *WE* esté habilitada (uso del decodificador y el demultiplexor).

3. Todos los registros deben tener señal de reloj y señal de inicialización.

4. Los datos leídos deben estar disponibles en las salidas dOut1 y dOut2.

5. Apariencia a medida requerida para el subcircuito indicada por la Figura 5.2 con las siguientes conexiones:

    - Entradas:
        - rs1: Dirección del primer registro a leer.
        - rs2: Dirección del segundo registro a leer.
        - w_address: Dirección del registro en el que se desea escribir.
        - dIn: Datos a escribir en el registro especificado por w_address.

- WE (Write Enable): Señal de habilitación de escritura.
- Salidas:
  - dOut1: Dato leído del registro especificado por rs1.
  - dOut2: Dato leído del registro especificado por rs2.

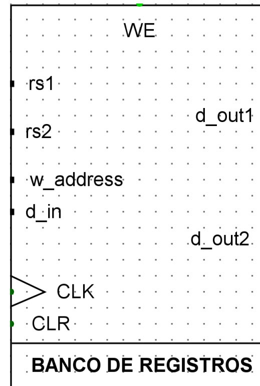


Figura 5.2: Apariencia requerida para el subcircuito Banco de registros.

- **Unidad Aritmético-Lógica (ALU)**

El segundo circuito se centra en la implementación de una ALU capaz de realizar todas las operaciones requeridas por las instrucciones de tipo R de la arquitectura RISC-V. La ALU es un componente fundamental y debe ser capaz de realizar operaciones aritméticas, lógicas, de comparación y de desplazamiento. El resultado obtenido de cada operación a realizar es recogido por la entrada de un multiplexor que es controlado por una señal de control manual. La ALU diseñada en este laboratorio debe tener las siguientes características:

1. El resultado de la operación correspondiente a la instrucción indicada debe ser recogida por la siguiente entrada del multiplexor:

operación	**entrada del multiplexor (valor decimal)**
add	0
sll	1
slt	2
sltu	3
xor	4
srl	5
or	6
and	7
sub	8
sra	9

Tabla 5.1: Tabla guía del diseño de la ALU.

2. Debe tener la apariencia a medida indicada en la Figura 5.3.

   - Entradas:
     - X: Primer operando (32 bits).
     - Y: Segundo operando (32 bits).
     - Control: Señal de control que permite seleccionar la operación a realizar introduciendo dígitos de forma manual.
   - Salidas:
     - OutALU: Resultado de la operación seleccionada (32 bits).

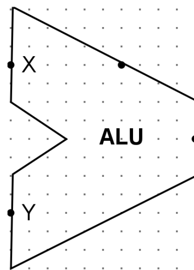


Figura 5.3: Apariencia requerida para el subcircuito ALU.

- **Ejercicio práctico de comprobación**

Para comprobar la correcta implementación de los circuitos se realiza un pequeño ejercicio práctico de simulación. Para ello se requiere que ambos circuitos estén conectados entre sí de forma que los resultados de las operaciones de la ALU se almacenen en el banco de registros y puedan ser utilizados en operaciones posteriores.

Para la simulación se colocan pines de tres bits en las entradas rs1, rs2 y w_address, otro pin con cuatro bits en la señal de control de la ALU y dos pines y un botón en las señales indicadas en la Figura 5.4. Estas últimas se construyen mediante pines, donde se introduce la señal, y túneles que comunican la señal con el resto del circuito.

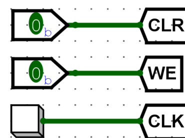


Figura 5.4: Señales de inicio, escritura y reloj del circuito.

Antes de iniciar la simulación, se introducen los valores decimales 13 y 20 a los registros tres y seis, respectivamente. Después, se deben proporcionar los valores indicados en la Tabla 5.2 a los pines correspondientes para cada señal de reloj y completar el resto de la tabla con los valores de la simulación obtenidos.

rs1	rs2	w_address	ControlALU	dOut1	dOut2	OutALU
110	011	010	0000			
110	010	001	1000			
010	011	100	0111			
110	100	101	0001			
010	011	111	0010			
010	110	110	0110			
001	100	111	1001			
101	010	101	0100			
001	100	111	0101			
010	011	111	0011			
010	001	111	0011			

Tabla 5.2: Tabla a completar durante la realización del laboratorio.

5.1.3. Preguntas a reflexionar sobre la práctica

Tras el desarrollo de los circuitos propuestos en el enunciado, se plantea a continuación una serie de situaciones y preguntas al estudiante con el objetivo de que reflexione sobre su trabajo.

- Se quiere realizar una instrucción que requiere de lectura y escritura en un solo ciclo y en el mismo registro. Piensa, por ejemplo, en una instrucción de tipo R y, basándote en el circuito realizado, explica por qué esta acción no genera ningún conflicto.

- La dirección cero del banco de registros de la arquitectura RISC-V, correspondiente al registro cero, siempre almacena un valor constante e inmutable de cero. ¿Qué solución o soluciones propones para que el circuito realizado tenga el mismo comportamiento?

- Para implementar la operaciones de desplazamiento de la ALU se ha de utilizar el bloque desplazador del programa. ¿Por qué motivo la segunda entrada de este bloque no toma el mismo número de bits que la primera entrada? Si la ALU hiciese operaciones con datos de 8 bits, ¿cuántos bits tomaría como entrada la segunda entrada del bloque desplazador? Razona la respuesta.

- Las instrucciones de tipo I realizan una serie de instrucciones paralelas a las de tipo R, pero el segundo valor entrante a la ALU corresponde al valor generado en el circuito generador de dato inmediato. ¿Qué solución propones para introducir el dato inmediato en vez de un valor almacenado en el banco de registros?

- ¿Cuál es la diferencia entre un desplazamiento lógico y un desplazamiento aritmético?

- ¿Cómo se implementa la comparación de números con signo y sin signo?¿Para qué tipo de comparación se usa complemento a dos y por qué?

5.2. Laboratorio 2

En esta sección del laboratorio, se propone la realización de un circuito fundamental en la arquitectura RISC-V, el circuito generador del dato inmediato. Este circuito es utilizado por la mayoría de las instrucciones de la arquitectura del procesador, por lo que resulta imprescindible para poder seguir implementando y conectando el resto de partes de la arquitectura.

5.2.1. Introducción al laboratorio

Dentro de este segundo laboratorio el primer circuito a desarrollar es el generador del dato inmediato. Este circuito tiene la función de tomar el valor inmediato almacenado en el campo *imm* de cada instrucción y extenderlo a un valor de 32 bits, manteniendo su signo. Esto es crucial para operaciones que requieren trabajar con valores inmediatos que pueden ser tanto positivos como negativos. El generador del dato inmediato garantiza que el valor inmediato se interprete correctamente en todas las operaciones aritméticas y lógicas que lo involucren.

Además, en este laboratorio se proporcionan los circuitos correspondientes a la memoria de programa junto con el contador de programa y al banco de registros, ahora sí, con 32 registros. El circuito obtenido del generador del dato inmediato debe unirse a los otros dos circuitos proporcionados y a la ALU implementada en el anterior laboratorio con el objetivo de poder realizar también operaciones en la ALU con el dato inmediato como uno de los operandos.

El objetivo de la realización de esta práctica es que el estudiante comprenda cómo se genera el dato inmediato a partir del formato de la instrucción para cada tipo de instrucción del procesador. Además, se espera que el estudiante adquiera un entendimiento más profundo de la estructura interna y de las interconexiones entre los distintos circuitos que forman la arquitectura.

La extensión del laboratorio está diseñada para ser realizada en dos sesiones de una hora cada una, distribuidas en dos semanas consecutivas. Los estudiantes tendrán un plazo límite de entrega de dos semanas desde la sesión inicial para completar y presentar sus circuitos.

5.2.2. Enunciado del laboratorio

En esta práctica de laboratorio se desarrolla un circuito esencial que forma parte de la arquitectura RISC-V, el generador del dato inmediato. Además, se realiza un ejercicio práctico en el que se unifican el circuito a implementar con dos circuitos proporcionados y la unidad aritmético-lógica implementada en el laboratorio anterior. Estos dos circuitos proporcionados son la memoria de programa junto con el contador de programa y un banco de registros formado por 32 registros.

- **Circuito generador del dato inmediato**

El circuito se centra en la implementación de un circuito generador del dato inmediato que extienda los valores inmediatos almacenados en el campo *imm* de las instrucciones de la arquitectura RV32I hasta 32 bits. Este circuito consta de dos partes: un circuito en el que se decodifica la instrucción y se construye el valor inmediato para cada tipo de instrucción y un subcircuito dentro de este que, a través de un pequeño circuito combinacional, selecciona el inmediato a utilizar según la instrucción en ejecución.

La entrada del circuito debe recibir los 32 bits que salen de la memoria de programa, es decir, cada una de las instrucciones con su formato. Con ayuda del elemento separador de la biblioteca de cables del programa Logisim-Evolution se ha de decodificar la instrucción generando así el valor inmediato para cada tipo de instrucción. El valor inmediato generado debe salir por la única salida del circuito hacia el resto de la arquitectura. La Figura 5.5 muestra como se generan los valores inmediatos de cada tipo de instrucción a partir de la posición de los bits del campo *imm* en el formato de la instrucción.

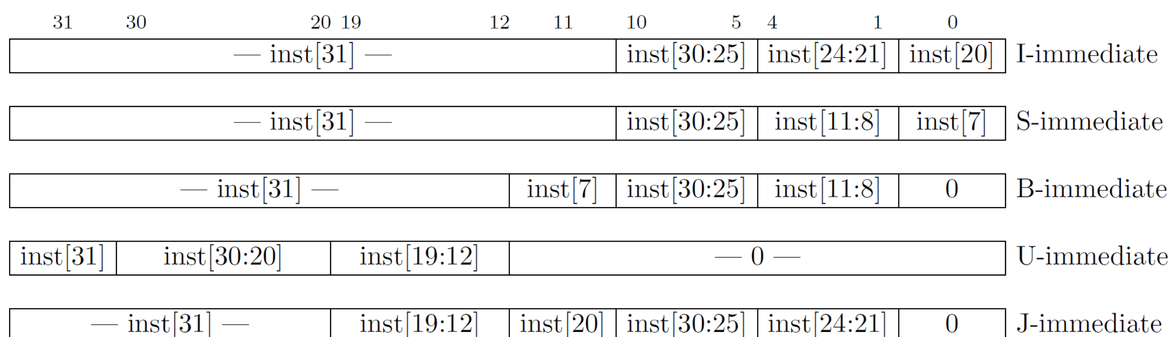


Figura 5.5: Tipos de inmediatos producidos por las instrucciones RISC-V. Los campos están etiquetados con los bits de instrucción utilizados para construir su valor.

Para seleccionar el valor inmediato a utilizar se ha de emplear un multiplexor cuya señal de selección corresponde a la salida del subcircuito combinacional. Para la generación de este subcircuito se precisa de la elaboración de una tabla de verdad.

1. ¿Son necesarios los 32 dígitos de la instrucción para generar la señal de selección?¿Qué campo del formato permite diferenciar unas instrucciones de otras?

- **Ejercicio práctico de comprobación**

Para comprobar la correcta generación del valor inmediato para cada tipo de instrucción, se han de unificar los circuitos proporcionados junto con la ALU implementada en el laboratorio anterior y el circuito generador del dato inmediato diseñado en un solo circuito. Después, se ha de colocar un pin de salida en el cable que une la salida del circuito generador del dato inmediato con una de las entradas de la ALU. Esta entrada a la ALU origina un conflicto que se ha de solucionar debido a que se juntan la salida del circuito generador del valor inmediato con una de las salidas del banco de

registros.

Una vez se tenga un único circuito se introduce un fichero *.txt* en formato binario en la memoria de programa. Este fichero contiene una instrucción de cada tipo que hacen uso del valor inmediato. Con el fichero introducido en la memoria se ha de completar la Tabla 5.3 en la que se indique a cada ciclo de reloj el valor inmediato generado por el circuito. Este valor puede verse reflejado en el pin de salida conectado. Para comprobar que los valores inmediatos generados son correctos se han de analizar los valores obtenidos y contrastarlos con el formato de las instrucciones proporcionadas.

Tipo	**Valor en binario (32 bits)**	**Valor en decimal**
I		
S		
B		
U		
J		

Tabla 5.3: Tabla a completar durante la realización del laboratorio.

5.2.3. Preguntas a reflexionar sobre la práctica

Tras el desarrollo de la práctica, se plantea a continuación una serie de preguntas al estudiante con el objetivo de que reflexione sobre alguna de las características de los circuitos.

- Si una instrucción tiene una longitud de 32 bits, ¿en cuánto hay que incrementar el contador de programa para que la dirección de la memoria apunte a la siguiente instrucción?
- ¿Qué diferencia hay entre una memoria RAM y una ROM?¿Cuál es más adecuada para desempeñar la función de memoria de programa y por qué?
- ¿Por qué crees que es necesario extender el valor inmediato hasta 32 bits?

5.3. Laboratorio 3

Este laboratorio se enfoca en el direccionamiento de la memoria y la alineación de datos, aspectos que introduce ligeramente la práctica anterior con la memoria de programa. Al abordar estos temas, se facilita una comprensión más profunda de cómo se gestionan y acceden los datos en la memoria. Los circuitos propuestos a resolver son los circuitos necesarios para la implementación de la totalidad de las instrucciones de carga y almacenamiento de datos.

5.3.1. Introducción al laboratorio

El primer circuito a desarrollar en esta práctica es el que implementa las instrucciones de tipo *load*. Estas instrucciones son responsables de cargar datos desde la memoria

a los registros del procesador. El circuito se ubicará a la salida de la memoria y se encargará de interpretar las instrucciones de carga, tales como **lw** (*load word*), **lh** (*load halfword*), **lhu** (*load halfword unsigned*), **lb** (*load byte*) y **lbu** (*load byte unsigned*). La correcta alineación de los datos y el manejo de los diferentes tamaños de datos (8, 16 y 32 bits) son aspectos clave que se abordarán en este circuito.

El segundo circuito a implementar es el que maneja las instrucciones de tipo *store*. Estas instrucciones permiten almacenar datos desde los registros del procesador hacia la memoria. El circuito se ubicará a la entrada de la memoria de datos, ya que son los datos a almacenar los que se ven afectados. Las instrucciones de almacenamiento incluyen **sw** (*store word*), **sh** (*store halfword*) y **sb** (*store byte*). Este circuito debe asegurar que los datos se almacenen correctamente en la memoria, respetando la alineación y el direccionamiento de la memoria.

Ambos circuitos se realizarán tras contestar unas preguntas de reflexión planteadas a los estudiantes para guiarles hacia su construcción. Estas preguntas están diseñadas para fomentar la comprensión de los conceptos clave antes de proceder a la implementación práctica. En el caso del circuito de tipo *store*, los estudiantes deberán realizar una tabla de verdad previa a la implementación del circuito, lo que les permitirá planificar y verificar el comportamiento esperado del circuito.

El objetivo de esta práctica de laboratorio es que el estudiante entienda cómo funcionan el direccionamiento y la alineación de la memoria en un procesador RISC-V, así como la manera en que se gestionan las operaciones de carga y almacenamiento de datos. A través del desarrollo de los circuitos para las instrucciones de tipo *load* y *store*, los estudiantes aprenderán a manejar distintos tamaños de datos (8, 16 y 32 bits) y a asegurar su correcta alineación en la memoria.

La extensión del laboratorio está diseñada para ser completada en dos sesiones de una hora cada una, distribuidas en dos semanas consecutivas. Los estudiantes tendrán un plazo límite de entrega de tres semanas desde la sesión inicial para completar y presentar sus circuitos.

5.3.2. Enunciado del laboratorio

Este laboratorio práctico se enfoca en la implementación y análisis de circuitos relacionados con la memoria de datos en la arquitectura RISC-V. En particular, se va a trabajar con las instrucciones de tipo *load* y *store*, fundamentales para la operación de cualquier procesador. Para este laboratorio, se requiere que la memoria esté alineada en direcciones múltiplos de cuatro, lo que significa que solo se permite el acceso a cada uno de los cuatro bytes que forma una palabra.

La memoria a utilizar en esta práctica debe ser una memoria RAM de 1K (1024 direcciones) que contenga las propiedades mostradas en la Figura 5.6. La memoria resultante tras asignar las propiedades mostradas tiene dos entradas, una para la señal de reloj y otra para habilitar la escritura (Escr). A mayores de estas dos señales necesita un búfer a su salida controlado por una señal que habilite la lectura del dato de la memoria

(Leer). Todas estas señales son pines que se accionan manualmente.

RAM (410,200)	
FPGA supported:	Supported
Número de bits de dirección	10
Número de bits de datos	32
Habilita:	La línea de uso permite
Tipo de carnero	volátil
Usar el pasador transparente	No
Tamaño de la línea	Individual
Allow misaligned?	No
Flanco	Flanco de subida
Implementación del bus de datos	Buses de datos separados para lectura y escritura
Etiqueta	HDL Required
Fuente de etiqueta	SansSerif Negrita 16
Etiqueta visible	No
Apariencia	Logisim-Evolution

Figura 5.6: Propiedades requeridas para la memoria de datos.

Estas propiedades indican entre otras cosas, que la memoria a utilizar no utiliza direccionamiento por bytes, sino que solo se permite el acceso en la memoria de datos alineados. Esto quiere decir que solo se permite el acceso a una palabra entera. Por ello, si se quiere acceder a algún byte concreto de la palabra, la forma de acceder a él es a partir de los dos bits más bajos de la dirección señalada a la entrada de la memoria y deberá de realizarse en los circuitos respectivos a cada tipo de instrucción.

- **Circuito para Instrucciones de Tipo *Load***

Realiza un pequeño circuito a la salida de la memoria de datos de la arquitectura que permita la ejecución de todas las instrucciones de tipo *load*. Para su realización es preciso entender como funciona el direccionamiento de la memoria.

Las instrucciones *load* pueden cargar 1 byte, 2 bytes o la palabra entera de la memoria al banco de registros indicado en la instrucción. La complejidad de estas instrucciones reside en que se puede cargar cualquiera de los cuatro bytes que forman la palabra almacenada o cualquiera de las tres combinaciones de dos bytes consecutivos. En caso de tener que almacenar un byte o dos de la memoria en un registro estos se almacenan en los bits más bajos y, depende de la instrucción en ejecución, si se considera el signo o no.

A continuación, se enumeran una serie de consideraciones a tener en cuenta a la hora de elaborar el circuito:

1. La memoria debe estar alineada en múltiplos de cuatro.
2. El valor de salida de la memoria siempre tiene una longitud de 32 bits.
3. Se recomienda utilizar los separadores empleados en anteriores laboratorios.
4. Los dos bits más bajos del campo *funct3* de la instrucción indican el tipo de instrucción *load* en ejecución. (Se simulará activando un pin de forma manual.)

5. El tercer bit del campo *funct3* de la instrucción indica si se considera el signo del valor a cargar en el registro. (Se simulará activando un pin de forma manual.)

- **Circuito para Instrucciones de Tipo *store***

La segunda implementación a realizar en este laboratorio corresponde al circuito necesario para la correcta ejecución de las instrucciones de tipo *store*. La implementación de este circuito ha de realizarse a la entrada de la memoria de datos, pues maneja los datos que se van a almacenar en ella.

Es importante saber que las instrucciones *store byte* y *store halfword* están diseñadas para almacenar específicamente los bytes menos significativos de un registro en la memoria. Sin embargo, estos bytes pueden ir almacenados en cualquiera de los cuatro bytes de la palabra. Los datos de la dirección de la memoria que no son sobrescritos por nuevos datos cargados deben permanecer intactos, por lo que hay múltiples combinaciones que dependen del byte exacto al que apunta la dirección entrante a la memoria.

Por último, al igual que para el circuito anterior es necesario reconocer qué bits del formato de cada una de las instrucciones *store* permite distinguirlas entre sí. En este caso hay que tener en cuenta los dos bits más bajos del campo *funct3* de la instrucción, que sirven para saber cuál de las tres instrucciones *store* es la que está en ejecución.

A continuación, se enumeran también una serie de consideraciones a tener en cuenta a la hora de elaborar el circuito:

1. La memoria debe estar alineada en múltiplos de cuatro.
2. El valor de entrada a la memoria siempre tiene una longitud de 32 bits.
3. Al circuito deben entrar los 32 bits almacenados en el registro y los 32 bits almacenados en la dirección de memoria a la que se apunta.

Para comenzar con la implementación se recomienda completar la Tabla 5.4 con los valores del registro y de la memoria mostrados en la Figura 5.7 de forma que el valor resultante de 4 bytes sea el valor entrante a la memoria según los valores que tomen los bits de control. Los bytes de control de la tabla y la salida a completar del circuito son:

- D1: segundo bit menos significativo de la dirección de memoria señalada.
- D0: bit menos significativo de la dirección de memoria señalada.
- F31: segundo bit menos significativo del campo *funct3* del formato de la instrucción.
- F30: bit menos significativo del campo *funct3* del formato de la instrucción.
- Sal x: valor que toma cada unos de los cuatro bits de la salida del circuito *store* y se va a almacenar en la memoria

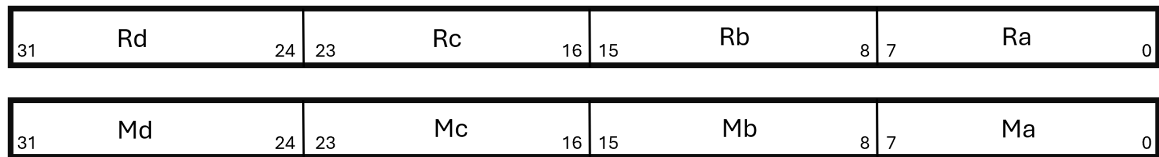


Figura 5.7: Esquema de los bytes que constituyen los datos almacenados en el registro (Ra-Rd) y en la dirección de memoria apuntada (Ma-Md). La combinación de ambos resulta en el nuevo valor almacenado en la memoria.

D1	D0	F31	F30	Sal a	Sal b	Sal c	Sal d
0	0	0	0				
0	0	0	1				
0	0	1	0				
⋮				⋮	⋮	⋮	⋮

Tabla 5.4: Tabla a completar durante la realización del laboratorio.

- **Ejercicio práctico de comprobación**

Para comprobar el correcto funcionamiento de ambos circuitos se realiza un ejercicio práctico de simulación. Para su ejecución es necesario conectar un pin de entrada de 32 bits que simule el valor extraído del registro a la entrada del circuito *store*. También se necesita colocar otro, también de 32 bits, a la entrada de la memoria que indica la dirección a señalar. A la hora de colocar este último pin hay que tener en cuenta que la memoria está alineada en direcciones múltiplos de cuatro y, los dos bits menos significativos de la dirección indicada en el pin, se utilizan también en ambos circuitos como señal de control.

Por último, también es necesario colocar un pin de salida de 32 bits que sea regulado por un búfer colocado inmediatamente antes del pin. Este búfer es controlado por una señal (Leer) que se activa de forma manual y que siempre toma un valor lógico de 1 cuando la instrucción a ejecutar es *load*.

Antes de activar la señal de reloj, se introducen los valores indicados en la Tabla 5.5 a los pines correspondientes. Después, se ha de completar el resto de la tabla con los valores obtenidos de la simulación.

Para terminar la práctica, se ha de conectar la memoria y los circuitos realizados con el circuito final del laboratorio anterior. Estas conexiones generan un pequeño conflicto, el cual se ha de resolver, a la entrada del banco de registros donde se une la salida de la ALU y la salida de la memoria de datos. La práctica se ha de entregar con todos los circuitos conectados de forma correcta.

address (hex)	**dataIn** (dec)	**F33** (sign)	**F31**	**F30**	**Escr**	**Leer**	**OutStore**	**OutLoad**
1FFC	84148994	0	1	0	1	0		
1FE8	84148994	0	0	0	1	0		
1FE9	84148994	0	0	0	1	0		
1FEA	84148994	0	0	0	1	0		
1FEB	84148994	0	0	0	1	0		
1FE8	0	0	1	0	1	0		
1FE8	84148994	0	0	1	1	0		
1FE9	84148994	0	0	1	1	0		
1FEA	84148994	0	0	1	1	0		
1FFC	x	0	1	0	0	1		
1FFC	x	0	0	0	0	1		
1FFD	x	0	0	0	0	1		
1FFE	x	0	0	0	0	1		
1FFF	x	0	0	0	0	1		
1FFC	x	0	0	1	0	1		
1FFD	x	0	0	1	0	1		
1FFE	x	0	0	1	0	1		
1FFC	x	1	0	0	0	1		
1FFD	x	1	0	0	0	1		
1FFE	x	1	0	0	0	1		
1FFF	x	1	0	0	0	1		
1FFC	x	1	0	1	0	1		
1FFD	x	1	0	1	0	1		
1FFE	x	1	0	1	0	1		

Tabla 5.5: Tabla a completar durante la realización del laboratorio.

5.3.3. Preguntas a reflexionar sobre la práctica

Tras el desarrollo de los circuitos propuestos en el enunciado, se plantea a continuación una serie de situaciones y preguntas al estudiante con el objetivo de que reflexione sobre los conocimientos aplicados en la práctica.

1. Si la memoria no requiriese que los datos estuvieran naturalmente alineados, ¿que otras posibles combinaciones de datos sería capaz de almacenar en memoria? Realiza una tabla, igual que la tabla guía del circuito correspondiente a las instrucciones de tipo *store*, y completa con los valores que tomarían los bits de control para esas otras salidas no valoradas para el circuito realizado en la práctica.

2. ¿Como afectaría la situación planteada en la pregunta anterior a las instrucciones de tipo *load*?

5.4. Laboratorio 4

En esta sección, se propone una continuación de los laboratorios anteriores, en la que el circuito resultante de la sesión 3 es modificado con el fin de incorporar el conjunto de instrucciones *branch*, la instrucción **jal** de tipo J y la instrucción **jalr** de tipo I.

5.4.1. Introducción al laboratorio

Para la realización del laboratorio se debe partir del resultado del circuito implementado en el laboratorio anterior. A partir de este se deben realizar modificaciones en el bloque de la ALU para poder realizar todas las comparaciones necesarias para la instrucciones *branch*. El resultado obtenido de estas comparaciones deberá extraerse a través de una nueva salida de un solo bit de la ALU. Para finalizar esta primera parte se ha de realizar la parte correspondiente al salto en la memoria de programa y resolver con ayuda de multiplexores los posibles conflictos que surjan.

La segunda parte de la práctica consiste en seguir modificando el circuito para conseguir implementar las instrucciones **jal** y **jalr**. Para ello se ha de modificar el camino que introduce datos al banco de registros y el camino que indica la dirección de la siguiente instrucción a ejecutar. En ambos casos hay conflictos con otros buses de datos que deben resolverse, también, con ayuda de multiplexores.

El objetivo de esta práctica es que el estudiante, a partir de los conocimientos adquiridos sobre la arquitectura, reflexione sobre los cambios necesarios para modificar el circuito sin alterar de forma errónea las instrucciones ya implementadas. Este proceso implicará identificar y resolver conflictos en el flujo de datos utilizando multiplexores, asegurando así que todas las instrucciones funcionen de manera coherente y eficiente en el circuito final. A través de esta práctica, los estudiantes reforzarán su comprensión de la arquitectura RISC-V y mejorarán sus habilidades en el diseño y optimización de circuitos complejos.

La extensión del laboratorio está diseñada para ser realizada en dos sesiones de una hora y media cada una, distribuidas en dos semanas consecutivas. Los estudiantes tendrán un plazo límite de entrega de dos semanas desde la sesión inicial para completar y presentar sus circuitos.

5.4.2. Enunciado del laboratorio

El objetivo de este laboratorio es modificar el circuito final implementado en la práctica anterior para añadir soporte a instrucciones de tipo *branch*, **jal** (*jump and link*) y **jalr** (*jump and link register*). Este laboratorio se divide en dos partes:

- **Implementación de Instrucciones *Branch***

En esta primera parte, se trabaja sobre el circuito resultante del laboratorio anterior. Las operaciones de tipo *branch* necesitan realizar una serie de comparaciones que se han de implementar en la ALU. La salida de esas comparaciones es fundamental a la hora de decidir la dirección de la siguiente instrucción a ejecutar, por lo que se debe

extraer del circuito ALU a través de una salida de un único bit.

Para resolver la parte correspondiente al salto en la memoria de programa se ha de emplear el bit resultante de la comparación junto con otra señal, cuyo valor siempre es un 1 lógico para las instrucciones de tipo *branch*, una puerta lógica y un multiplexor.

Para simplificar la parte de las comparaciones, se recomienda emplear unos bloques de comparación nuevos y no los ya empleados para las instrucciones de comparación de tipo I y de tipo R.

- **Implementación de Instrucciones Jal y Jalr**

En esta segunda parte, se continúa modificando el circuito para implementar las instrucciones **jal** y **jalr**. Para ello, se ha de modificar el camino del dato a introducir al banco de registros y la lógica que actualiza al contador de programa con la dirección de la siguiente instrucción a ejecutar. Para resolver los conflictos con otros buses de datos han de emplearse multiplexores.

Para la instrucción **jalr**, el dato de dirección de memoria a saltar debe tener el último bit a cero para que la instrucción sea par. ¿Qué puerta lógica hay que pueda enmascarar a cero el último bit?

- **Ejercicio práctico de comprobación**

Para finalizar la práctica se ha de ejecutar un pequeño programa proporcionado en binario en un *.txt* que se ha de introducir en la memoria de programa. A mayores, se ha de conectar un pin de salida a la entrada del contador de programa para poder visualizar la dirección de la siguiente instrucción a ejecutar. El programa a ejecutar proporcionado es el siguiente:

```
1  addi t0, t0, 2
2  addi t1, t1, 0
3  addi t3, zero, 0x00000008
4  bge t1, t0, loop2
5  addi t1, t1, 1
6  jalr s0, t3, 4
7  loop2:
8  bltu t0, t1, end
9  addi t0, t0, -1
10 jal s1, -8
11 end:
12     beq zero, zero, end
```

Código 5.1: Código ensamblador del programa a ejecutar.

Se ha de completar la Tabla 5.6 con la dirección indicada en el pin de salida colocado a la entrada del contador de programa a cada ciclo de reloj hasta completar la ejecución de todas las instrucciones.

Nº de ciclo de reloj	Dirección del PC en binario (sin los dos bits más bajos)
1	
2	
3	
4	
⋮	⋮

Tabla 5.6: Tabla a completar durante la realización del laboratorio.

Por último se ha de explicar brevemente el programa y contestar a la siguiente cuestión:

- En el segundo bucle del programa, ¿por qué motivo se termina cumpliendo que el valor en t1 es menor que en t0 si únicamente se va decrementando el valor de t0 en cada vuelta?

5.4.3. Preguntas a reflexionar sobre la práctica

Tras el desarrollo de la práctica propuesta, se plantea a continuación una pregunta al estudiante. Esta tiene como objetivo que reflexione y repase algunos de los conocimientos adquiridos en la implementación de circuitos anteriores y entienda mejor la conexión que hay entre ellos.

- Para el caso de la instrucción **jal**, ¿por qué razón el último bit del valor con el que se actualiza el contador de programa no se enmascara con una puerta lógica?

5.5. Laboratorio 5

En esta sección, se propone la realización de dos circuitos combinacionales esenciales para la arquitectura RISC-V: un circuito generador de señales de control para la ALU y un circuito de control que maneje el flujo de datos a través de la arquitectura. Estos componentes son fundamentales para el correcto funcionamiento y coordinación de las operaciones dentro del procesador. Este laboratorio es útil para aplicar lo aprendido sobre circuitos combinacionales y comprender mejor su diseño.

5.5.1. Introducción al laboratorio

El primer circuito a desarrollar en esta práctica es el generador de señales de control para la ALU. Este circuito se encargará de determinar qué operación debe realizar la ALU en función de las instrucciones recibidas. Utilizando los bits almacenados en el campo *opcode* y otros campos relevantes de la instrucción, el generador de señales de control producirá las señales necesarias para seleccionar la operación adecuada en la ALU. Este proceso de decodificación y generación de señales es crucial para la ejecución correcta de las instrucciones en el procesador.

El segundo circuito es el circuito de control que maneja el flujo de datos a través de la arquitectura RISC-V. Este circuito coordina el movimiento de datos entre los diferentes componentes del procesador, como la memoria, el banco de registros, y la

ALU. Utilizando señales de control específicas, este circuito asegura que los datos se muevan de manera eficiente y correcta a través de los componentes que forman la arquitectura, permitiendo la ejecución secuencial y ordenada de las instrucciones.

El objetivo de esta práctica es que el estudiante logre conectar todos los circuitos desarrollados previamente para formar un circuito final que funcione como un componente integral del procesador. A través de la generación de cada una de las señales de control, el estudiante profundizará en la coordinación entre los distintos circuitos. Esto permitirá visualizar y analizar cómo las instrucciones se decodifican y ejecutan de manera secuencial y ordenada, consolidando su comprensión de la interacción y sincronización necesarias para el correcto funcionamiento de un procesador basado en la arquitectura RISC-V

La extensión del laboratorio está diseñada para ser realizada en dos sesiones de dos horas cada una, distribuidas en dos semanas consecutivas. Los estudiantes tendrán un plazo límite de entrega de tres semanas desde la sesión inicial para completar y presentar sus circuitos.

5.5.2. Enunciado del laboratorio

En esta práctica de laboratorio se desarrollan dos circuitos combinacionales. Los circuitos combinacionales son aquellos cuya salida depende únicamente del estado actual de sus entradas, sin depender de estados anteriores. Estos circuitos se utilizan ampliamente en la electrónica digital para controlar el flujo de datos y señales en sistemas más complejos. La correcta implementación de estos circuitos requiere un análisis detallado de las funciones lógicas que deben realizar y la construcción de tablas de verdad que definan su comportamiento.

- **Control de Operaciones en la ALU**

La primera implementación consiste en diseñar un circuito combinacional que seleccione el resultado de la operación realizada en la unidad aritmético-lógica (ALU) implementada en el primer laboratorio. Este circuito debe ser capaz de manejar todas las operaciones definidas por el conjunto de instrucciones de la arquitectura RV32I, no solo las de tipo R.

A continuación, se plantea una serie de pasos que el estudiante podrá seguir que servirán como guía en la construcción de la tabla de verdad del circuito y la consecución de laboratorio.

Enumeración de los pasos recomendados:

1. Realizar una lista con todos los tipos de instrucciones que hacen uso de la ALU.
2. Desglosar las instrucciones de cada tipo e indicar el tipo de operación que precisa cada una para su ejecución.
3. Identificar los bits del formato de las instrucciones que permite diferenciarlas entre sí.

4. Realizar una tabla de verdad con los bits diferenciativos entre las instrucciones y el valor que debe de tomar la señal de control para seleccionar la entrada correspondiente a la operación del multiplexor del subcircuito ALU.

Se recomienda también, seguir el orden indicado en la tabla guía del diseño de la ALU del primer laboratorio (Tabla 5.1).

- **Circuito Unidad de control**

El segundo proyecto se centrará en la implementación de un circuito de control que gestione el flujo de datos en la arquitectura del procesador a partir de señales de control. Este circuito debe ser capaz de dirigir los datos a través de los diferentes componentes del procesador según el tipo de instrucción en ejecución.

La solución a este circuito de control no es única, pues depende de las señales de control elegidas. Esto implica que el diseño de la arquitectura también puede variar, pero no su funcionalidad. Para facilitar su resolución, a continuación se presenta una serie de pasos que el estudiante podrá seguir y que servirán de guía en la construcción de la tabla de verdad del circuito.

Enumeración de los pasos recomendados:

1. Unificar en un solo circuito todos los circuitos implementados en los laboratorios anteriores y conectarlos entre sí.
2. Colocar multiplexores auxiliares donde se genere algún conflicto para controlar el flujo de datos.
3. Comprobar si los multiplexores tienen señal de selección.
4. Repasar todos los elementos que forman la arquitectura RISC-V y ver si necesitan alguna señal de control que dirija el flujo de datos que circula por ellos (lectura, escritura, etc).
5. Realizar una tabla e ir comprobando instrucción por instrucción su recorrido y las señales necesarias para su correcta ejecución.

- **Comprobación final**

Para finalizar con la práctica se va a simular un pequeño programa. El propósito del programa es mover 32 valores de 32 bits almacenados a partir de la dirección de memoria 0x000000FC a posiciones consecutivas de la memoria a partir de la dirección 0x00000400. Para comprobar el correcto funcionamiento del circuito primero se ha de introducir en la memoria de datos un fichero *.txt* proporcionado con los valores a mover. También ha de introducirse en la memoria de programa en formato little-endian otro fichero *.txt* proporcionado con las instrucciones codificadas en binario.

Una vez introducidos ambos ficheros en las respectivas memorias se procede a rellenar los campos ‘operación lógica’ y ‘cambios realizados’ de la Tabla 5.7 que contiene las instrucciones del programa a realizar. Por último, tras la finalización del programa se comprueba si la ejecución se ha realizado de forma correcta.

Etiqueta	Lenguaje Ensamblador	Operación lógica	Cambios Realizados
	addi t0, zero, 0x020		
FOR	beq t0, zero, FIN		
	addi t0, t0, 0xFFF		
	lw t1, 0x0FC(t2)		
	sw t1, 0x400(t2)		
	addi t2, t2, 0x004		
	beq zero, zero, FOR		
FIN	beq zero, zero, FIN		

Tabla 5.7: Tabla a completar durante la realización del laboratorio.

5.5.3. Preguntas a reflexionar sobre la práctica

Tras el desarrollo de los circuitos propuestos en el enunciado, se plantea a continuación una pregunta al estudiante con el objetivo de que, basándose en los conocimientos adquiridos a lo largo de las prácticas, encuentre más de una única solución al problema planteado.

- En el circuito correspondiente a la unidad de control de la práctica se comenta que hay mas de una solución para las señales de control, ¿se te ocurre alguna? Explica la solución alternativa.

5.6. Laboratorio 6

En esta sección se propone el último laboratorio que compone las prácticas para la implementación del RISC-V. El laboratorio se divide en dos partes: la primera se centra en la adición de las instrucciones **lui** y **auipc** a un circuito preexistente, y la segunda en la ejecución de un programa que se proporciona al estudiante en formato *.txt* para verificar el correcto funcionamiento del circuito modificado.

5.6.1. Introducción al laboratorio

El desarrollo de la práctica se basa en completar el diseño de un circuito proporcionado basado en la arquitectura RISC-V. La finalidad consiste en ejecutar la totalidad de las instrucciones que componen la arquitectura RV32I. La segunda parte de este laboratorio se encarga de comprobar el correcto funcionamiento de la arquitectura final construida a través de un programa proporcionado.

El objetivo de este laboratorio es que los estudiantes comprendan y apliquen los conceptos de diseño y simulación de arquitecturas de procesadores, específicamente la arquitectura RISC-V RV32I.

La extensión del laboratorio está diseñada para ser realizada en una sesión de dos horas. Los estudiantes tendrán un plazo límite de entrega de dos semanas desde la sesión inicial para completar y presentar su circuito final.

5.6.2. Enunciado del laboratorio

En esta práctica de laboratorio se implementan las instrucciones de tipo U. Para ello se proporciona un circuito compuesto por todos los circuitos implementados en prácticas anteriores y que es capaz de ejecutar de forma correcta la totalidad de las instrucciones de la arquitectura RV32I a excepción de las de tipo U.

Tras completar la implementación del circuito que constituye la arquitectura RV32I se procede a ejecutar un programa cuyo objetivo consiste en verificar el correcto funcionamiento del circuito modificado, asegurando que todas las instrucciones, incluidas las nuevas instrucciones de tipo U, se ejecutan correctamente.

- **Adición de Instrucciones de tipo U (lui y auipc)**

En esta primera parte, se proporciona un circuito que implementa todas las instrucciones de la arquitectura RV32I, excepto las instrucciones de tipo U (lui y auipc). Los estudiantes deben añadir los elementos necesarios al circuito para soportar las instrucciones lui y auipc. Esto incluye la modificación de la tabla de verdad de la unidad de control para generar las señales de control adecuadas para estas instrucciones.

- **Comprobación a través de la ejecución de un Programa**

En la segunda parte del laboratorio, se va a simular un pequeño programa proporcionado en un archivo *.txt* que contiene las instrucciones en binario. Este programa se debe cargar en formato little-endian en la memoria de programa para poder simularlo.

La Tabla 5.8 muestra las instrucciones del programa a ejecutar, junto con dos campos vacíos que se han de completar antes de realizar la simulación. Una vez se han completado, se ha de ejecutar la simulación y comprobar su correcto funcionamiento.

Etiqueta	Lenguaje Ensamblador	Operación	Cambios Realizados
	add x4, zero, zero		
	lui x6, 0x60000		
	srai x5, x6, 15		
	addi x7, zero, 1		
LOOP	beq x4, x5, FIN		
	add x7, x7, x7		
	sw x7, 0(x4)		
	addi x4, x4, 4		
	jal x0, LOOP		
FIN	beq zero, zero, FIN		

Tabla 5.8: Tabla a completar durante la realización del laboratorio.

5.6.3. Preguntas a reflexionar sobre la práctica

Después de completar la implementación de la totalidad del procesador y haber visto su comportamiento en tiempo real durante un programa en ejecución, se plantea a continuación una pregunta al estudiante con el objetivo de que se familiarice con los programas en lenguaje ensamblador.

- Después de ejecutar el programa propuesto explícalo e indica a qué operación equivale la parte correspondiente al bucle del programa. ¿Podrías realizar otro programa en el que se calcule la multiplicación entre los números 5 y 4?

Conclusiones

En este trabajo se ha realizado la implementación de la extensión RV32I de la arquitectura RISC-V utilizando la herramienta de simulación y diseño Logisim-Evolution. El proyecto desarrollado ha sido un proyecto integral que ha permitido desarrollar habilidades avanzadas en el diseño y simulación de circuitos digitales, así como una comprensión profunda del funcionamiento de la arquitectura RISC-V.

En primer lugar, se ha conseguido implementar el procesador dividiendo su desarrollo según los tipos de instrucciones que componen la arquitectura. Este enfoque ha facilitado la comprensión de la funcionalidad de cada instrucción, los componentes de la arquitectura que utilizan y el desglose del formato de cada instrucción en el circuito obtenido. Además, se ha aprendido a emplear la herramienta Logisim-Evolution de manera efectiva para el diseño del circuito.

El circuito final obtenido cumple con el objetivo principal del proyecto: implementar un procesador funcional basado en la arquitectura RISC-V que puede ser utilizado en hardware real, como una FPGA, o en futuros proyectos que busquen ampliar la capacidad de instrucciones del procesador. La verificación exhaustiva realizada ha asegurado que el diseño cumple con las especificaciones y funciona de manera fiable.

En segundo lugar, se ha logrado completar una serie de prácticas de laboratorio que guían al estudiante a implementar un procesador basado en la arquitectura RISC-V de cero. Estos laboratorios incluyen una parte de implementación, una parte práctica para probar el circuito y una parte teórica que cuestiona ciertos aspectos del diseño, Los laboratorios han permitido abordar la arquitectura del procesador de manera progresiva, permitiendo a los estudiantes centrarse en cada componente de manera individual y comprender su funcionamiento y su interrelación con otros componentes.

En conclusión, la combinación del conocimiento adquirido respecto al ámbito técnico durante la implementación del circuito, junto con la aplicación de dicho conocimiento en el ámbito docente, ha resultado en un proyecto integral que no solo cumple con los objetivos planteados, sino que también proporciona una valiosa experiencia educativa. Este trabajo ha demostrado la viabilidad y las ventajas de la arquitectura RISC-V, subrayando la importancia de seguir investigando y enseñando sobre arquitecturas abiertas en el ámbito académico y profesional.

Bibliografía

[1] D. A. Patterson y A. Waterman, Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta, Primera edición. Berkeley, California: Strawberry Canyon LLC. que es, 2018. Disponible en: http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf

[2] A. S. Waterman, Design of the RISC-V instruction set architecture. University of California, Berkeley, 2016. Disponible en: https://escholarship.org/content/qt7zj0b3m7/qt7zj0b3m7.pdf

[3] C. P. Bou, «Barcelona da un salto adelante con el desarrollo de nuevos chips avanzados», www.elperiodico.com. Disponible en: https://www.elperiodico.com/es/sociedad/20231213/barcelona-desarrollo-semiconductores-tecnologia-chips-supercomputacion-95783674

[4] C. Burch et al., «Logisim-evolution». octubre de 2022. Disponible en: https: https://github.com/logisim-evolution/logisim-evolution

[5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, «The RISC-V Instruction Set Manual, Volume I: Base User-level ISA», EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, vol. 116, pp. 1-32, 2011. Disponible en: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf

[6] «History – RISC-V International». Disponible en: https://riscv.org/about/history/

[7] «Members – RISC-V International». Disponible en: https://riscv.org/members/

[8] K. Asanović and D. A. Patterson, «Instruction sets should be free: The case for RISC-V», EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014. Disponible en: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf

[9] «An Introduction to RISC-V—Understanding RISC's Open ISA - Technical Articles». Disponible en: https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/

[10] S. Álvarez, «Proceso de traducción de los lenguajes de programación», DesarrolloWeb.com. Disponible en: https://desarrolloweb.com/articulos/2387.php

[11] C. Chen, G. Novick, & K. Shimano. «RISC Architecture: RISC vs. CISC», Stanford University. (2006, Diciembre 16) Disponible en: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

[12] T. Chen and D. A. Patterson, «RISC-V Genealogy» EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6, 2016. Disponible en: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.pdf

[13] C. Celio, «CS 152 Laboratory Exercise 3» UC Berkeley, Regents of the University of California, 2015. Disponible en: https://inst.eecs.berkeley.edu//~cs152/sp14/handouts/lab3.pdf

[14] «¿Qué es una patente? - Portal OEPM». Disponible en: https://www.oepm.es/es/invenciones/como-proteger-las-invenciones/conceptos-basicos/que-es-una-patente/

[15] D. Mangum, «RISC-V Bytes: Caller and Callee Saved Registers». Disponible en: https://danielmangum.com/posts/risc-v-bytes-caller-callee-registers/

[16] D. A. Patterson and J. L. Hennessy, «Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition», Elsevier/Morgan Kaufmann, 2017, pp.1657 - 1658. Disponible en: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/HandP_RISCV.pdf

[17] M. Kayaalp, «Using Logisim-evolution for Teaching Datapath and Control», 2021 ACMIEEE Workshop Comput. Archit. Educ. WCAE, pp. 1-8, jun. 2021, doi: 10.1109/WCAE53984.2021.9707148. Disponible en: https://www.semanticscholar.org/paper/Using-Logisim-evolution-for-Teaching-Datapath-and-Kayaalp/ccefe922c33537ac76f9ed4fd90c5be2169a2e56

[18] «Computer Organization with Logisim». Disponible en: https://mkayaalp.net/computer-organization-logisim/

[19] S. Sharma, «RISC-V Architecture: A Comprehensive Guide to the Open-Source ISA». Disponible en: https://www.wevolver.com/article/risc-v-architecture

[20] «Common CPU components - The CPU - Eduqas - GCSE Computer Science Revision - Eduqas», BBC Bitesize. Disponible en: https://www.bbc.co.uk/bitesize/guides/zhppfcw/revision/2

[21] «Control unit», Wikipedia. 27 de abril de 2024. Disponible en: https://en.wikipedia.org/w/index.php?title=Control_unit&oldid=1221052119

[22] M. M. Mano and C. R. Kime, «Logic and Computer Design Fundamentals», 3rd ed., p. 453, 2004. Disponible en: https://powerunit-ju.com/wp-content/uploads/2017/01/Book-Logic-and-Computer-design-Fundamenals-4ed.pdf

[23] D. K. Dennis et al., «Single cycle RISC-V micro architecture processor and its FPGA prototype», in 2017 7th International Symposium on Embedded Computing and System Design (ISED), 2017, pp. 1-5. Disponible en: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8303926&casa_token=pja5IzQB2lwAAAAA:G2OFHP98tantkKwKThN_-yio4TgEHeR_VMxkL74oupeGKTAppLTz1zvSHy-c9GjXURWuac6Nqw&tag=1

[24] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, «The RISC-V Instruction Set Manual. Volume 1: User-level ISA, Version 2.0», California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, Tech. Rep., 2014. Disponible en: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf

[25] H. Miyazaki, T. Kanamori, M. A. Islam, and K. Kise, «RVCoreP: An optimized RISC-V soft processor of five-stage pipelining», IEICE TRANSACTIONS on Information and Systems, vol. 103, no. 12, pp. 3-4, 2020. Disponible en: https://arxiv.org/abs/2002.03568

[26] F. Embeddev, «RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA», Five EmbedDev. Disponible en: https://www.five-embeddev.com//riscv-user-isa-manual/Priv-v1.12/rv32.html

[27] W. Green, «RISC-V Assembler: Load Store», Project F. Disponible en: https://projectf.io/posts/riscv-load-store/

[28] «Ripes/docs/README.md at master · mortbopet/Ripes», GitHub. Disponible en: https://github.com/mortbopet/Ripes/blob/master/docs/README.md

[29] «RISC-V: Instrucciones auipc y lui en la implementación monociclo. – Rincón de SystemVerilog». Disponible en: https://dsd.webs.upv.es/?p=734