Laboratorio No.3 Diseño y verificación del procesador monociclo

Jose Manuel Londoño Castaño

Juan José Gómez Castaño

Universidad de Antioquia

Arquitectura de computadores y laboratorio

Fredy Alexander Rivera Vélez

El Carmen de Viboral Antioquia

8/07/2025

Resumen:

Este informe presenta el diseño e implementación de un procesador monociclo basado en la arquitectura MIPS32, utilizando la herramienta Logisim Evolution. El procesador fue construido para ejecutar un subconjunto de 14 instrucciones MIPS, incluyendo operaciones aritméticas, lógicas, de control de flujo y acceso a memoria. Para verificar su funcionamiento, se desarrolló un programa en lenguaje ensamblador MIPS que calcula los valores máximos y mínimos por fila y columna en una matriz de 4x4 números enteros. La simulación y validación se realizaron empleando MARS para el código ensamblador y Logisim Evolution para el diseño del hardware.

Palabras clave: MIPS32, procesador monociclo, Logisim Evolution, ensamblador MIPS, unidad de control, ruta de datos.

Abstract:

This report presents the design and implementation of a single-cycle processor based on the MIPS32 architecture, using the Logisim Evolution tool. The processor was built to execute a subset of 14 MIPS instructions, including arithmetic, logic, flow control, and memory access operations. To verify its operation, a MIPS assembly language program was developed that calculates the maximum and minimum values by row and column in a 4x4 integer matrix. Simulation and validation used MARS for the assembly code and Logisim Evolution for the hardware design.

Keywords: MIPS32, single-cycle processor, Logisim Evolution, MIPS assembler, control unit, data path.

Tabla de contenido

1. Introducción

El objetivo de este informe de laboratorio es presentar el proceso de construcción de un procesador monociclo que soporte una versión reducida de la arquitectura MIPS32 en la herramienta Logisim Evolution y que esté en capacidad de ejecutar un programa de un problema de ingeniería cotidiano (El ordenamiento de un listado de números) resuelto en lenguaje MIPS.

En este proyecto, se utilizaron técnicas vistas en el curso de la Universidad de Antioquia, Arquitectura de Computadores y Laboratorio, técnicas relacionadas con los temas de diseño de circuitos digitales, programación en bajo nivel, diseño del procesador y unificación de las mismas, herramientas de simulación del lenguaje ensamblador MIPS (MARS) y el simulador de diseño de circuitos Logisim Evolution para cumplir con el objetivo.

En este informe, se describe en detalle el diseño y la implementación del sistema, incluyendo la construcción del Hardware simulado y el Software, en la que se encontraron varios puntos de decisión, donde se buscó priorizar el correcto funcionamiento del procesador monociclo, incluyendo las instrucciones adicionales y el problema planteado utilizando solo el conjunto de instrucciones que se determinó en el procesador.

2. Objetivos

- Comprender los requerimientos de una versión reducida de la arquitectura MIPS de 32 bits para realizar su implementación en forma de procesador monociclo (ruta de datos y unidad de control).
- Codificar, ensamblar y simular un programa de prueba para verificar el comportamiento correcto del procesador implementado.
- Emplear herramientas de software para el diseño y la simulación de computadores digitales.

3. Marco teórico

3.1 Arquitectura MIPS

La arquitectura MIPS (Microprocessor without Interlocked Pipeline Stages) es una arquitectura RISC (Reduced Instruction Set Computer) diseñada para simplificar las instrucciones de máquina y facilitar la implementación eficiente de procesadores. Su estructura se basa en instrucciones de longitud fija (32 bits), un número limitado de formatos de instrucción y un conjunto reducido de operaciones aritméticas, lógicas, de control y de acceso a memoria. Esta arquitectura es ampliamente utilizada en contextos educativos y de investigación debido a su simplicidad y claridad conceptual.

3.2 Procesador Monociclo

Un procesador monociclo es aquel en el que cada instrucción se ejecuta completamente en un solo ciclo de reloj. Esto implica que las operaciones de búsqueda de instrucción, decodificación, ejecución, acceso a memoria y escritura del resultado deben completarse en un único ciclo, lo cual impone restricciones de diseño y eficiencia. Aunque este enfoque no es el más eficiente en términos de rendimiento, es ideal para fines educativos y para comprender el flujo básico de datos y control dentro de una CPU.

3.3 Ruta de Datos y Unidad de Control

La ruta de datos de un procesador monociclo MIPS incluye componentes como el contador de programa (PC), la memoria de instrucciones, el banco de registros, la ALU (unidad aritmético-lógica), la memoria de datos y los multiplexores que dirigen el flujo de información. La unidad de control genera las señales necesarias para coordinar estos componentes de acuerdo con la instrucción actual, interpretando los campos opcode y funct para activar las señales correspondientes.

La correcta interacción entre la ruta de datos y la unidad de control es esencial para que el procesador ejecute las instrucciones correctamente. En este diseño, la unidad de control se divide en dos partes: la unidad de control principal y la unidad de control de la ALU, que juntas determinan el comportamiento del sistema frente a instrucciones como ADD, LW, BEQ, J, entre otras.

3.4 Instrucciones soportadas

El procesador implementado está diseñado para ejecutar un subconjunto reducido de instrucciones de la arquitectura MIPS32, suficiente para abordar problemas de lógica y

procesamiento básico de datos. Este subconjunto contempla operaciones aritméticas, lógicas, de control de flujo y transferencia de datos. Las instrucciones incluidas en el diseño son las siguientes:

- Transferencia de datos**:**
  lw (load word), sw (store word)
- Aritmético-lógicas:
  add, sub, and, or, nor, slt (set-on-less-than)
- Control de flujo:
  beq (branch on equal), j (jump)
- Instrucciones adicionales asignadas:
  addi (add immediate), lh (load halfword)
- Instrucciones de salto a subrutinas:
  jal (jump and link), jr (jump register)

Este conjunto de 14 instrucciones fue seleccionado siguiendo los lineamientos del proyecto, priorizando la capacidad del procesador para ejecutar un programa de prueba completo, como el ordenamiento de una lista de números. Cada instrucción fue implementada considerando su tipo (R, I o J), su codificación binaria y las señales de control requeridas para su ejecución dentro de la arquitectura monociclo diseñada en Logisim Evolution.

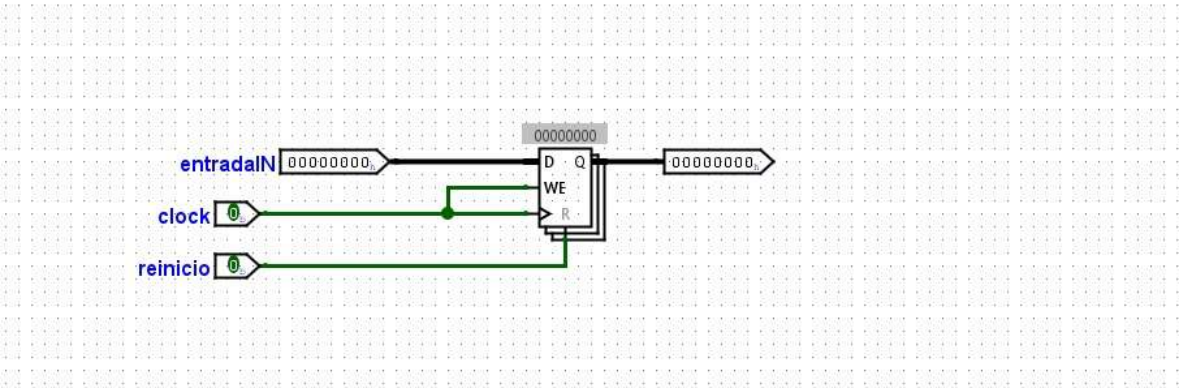
3.5 Herramientas utilizadas: Logisim Evolution y MARS

Para el desarrollo del procesador, se utilizó Logisim Evolution, una herramienta gráfica de simulación de circuitos digitales, ideal para construir la arquitectura del procesador paso a paso. Esta herramienta permite visualizar el funcionamiento interno de la CPU y realizar pruebas controladas con diferentes entradas.

Por otro lado, MARS (MIPS Assembler and Runtime Simulator) se utilizó para escribir y verificar el código ensamblador. Una vez verificado el programa, sus instrucciones fueron codificadas manualmente en binario y cargadas en la memoria de instrucciones del procesador implementado en Logisim.

4. Componentes del sistema

4.1 PC



El PC fue implementado mediante un registro de 32 bits que mantiene la dirección de la siguiente instrucción a ejecutar. Este componente se actualiza en cada ciclo de reloj, dependiendo del flujo del programa (ejecución secuencial o salto).

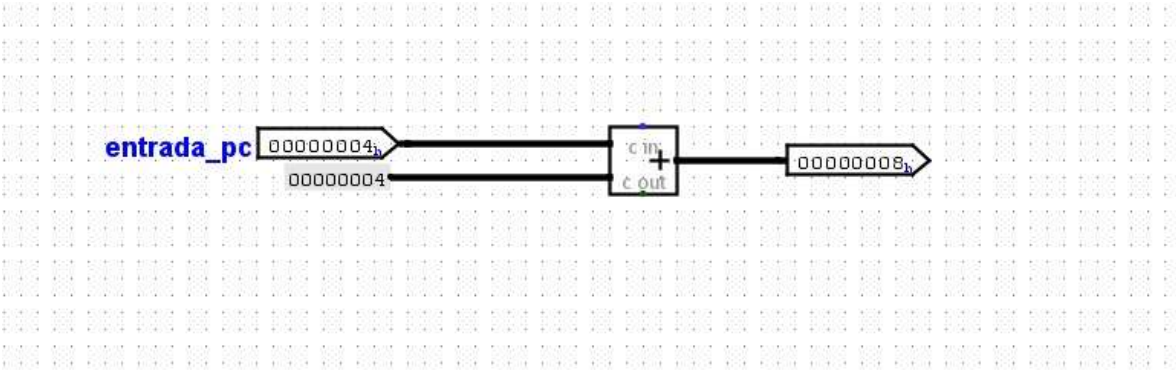
En el diseño realizado en Logisim Evolution, el PC cuenta con las siguientes señales:

- entradaIN**:** entrada de datos de 32 bits que representa la nueva dirección a cargar. Esta puede ser PC + 4, PC + offset, o una dirección directa para instrucciones j.
- clock**:** señal de reloj que sincroniza la escritura del nuevo valor del PC.
- reinicio (R)**:** al activarse, reinicia el valor del PC a cero. Esta señal permite inicializar el sistema al cargar un programa.

El registro se actualiza solo si la señal de WE (Write Enable) está activa, permitiendo así un control preciso sobre cuándo debe cambiar el valor del contador, y su salida se conecta directamente a la memoria de instrucciones, sirviendo como dirección base para la búsqueda de la próxima instrucción.

Esta implementación modular y controlada del PC permite una ejecución secuencial fluida, así como el manejo correcto de instrucciones de salto o cambio de flujo dentro del programa.

4.2 Add (sumador de PC + 4)



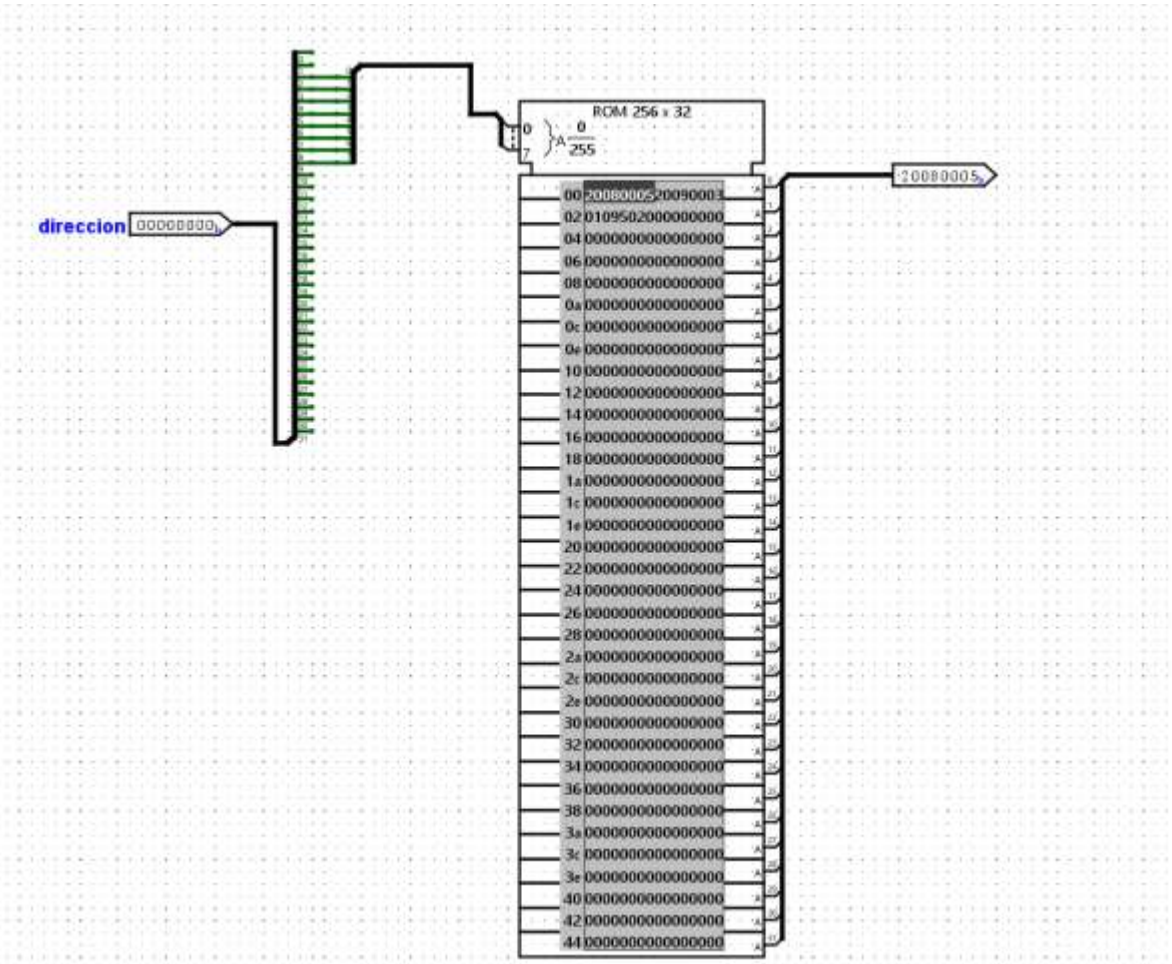
El sumador de PC + 4 es un componente encargado de calcular la dirección de la siguiente instrucción en un flujo de ejecución secuencial. Dado que cada instrucción MIPS ocupa 4 bytes, avanzar una posición en memoria implica incrementar el valor del contador de programa (PC) en cuatro unidades.

En el diseño implementado en Logisim Evolution, este sumador recibe dos entradas:

- La salida actual del PC, es decir, la dirección de la instrucción en curso.
- Una constante de valor 4, que representa el tamaño fijo de cada instrucción.

El resultado del sumador se utiliza como una de las posibles entradas del siguiente valor del PC, en caso de que no se ejecute una instrucción de salto o ramificación. Esta dirección (PC + 4) también se utiliza como base en el cálculo de saltos condicionales (beq) y concatenaciones en saltos absolutos (j).

4.3 instructionmemory



La memoria de instrucciones es el componente responsable de almacenar las instrucciones codificadas en lenguaje máquina que el procesador debe ejecutar. En la arquitectura MIPS32, cada instrucción tiene una longitud fija de 32 bits (4 bytes), por lo que la memoria se organiza en múltiplos de 4.

En el diseño realizado en Logisim Evolution, se utilizó un componente ROM de 256 x 32 bits, donde:

- Cada celda representa una instrucción de 32 bits.
- La entrada de dirección proviene directamente del PC.

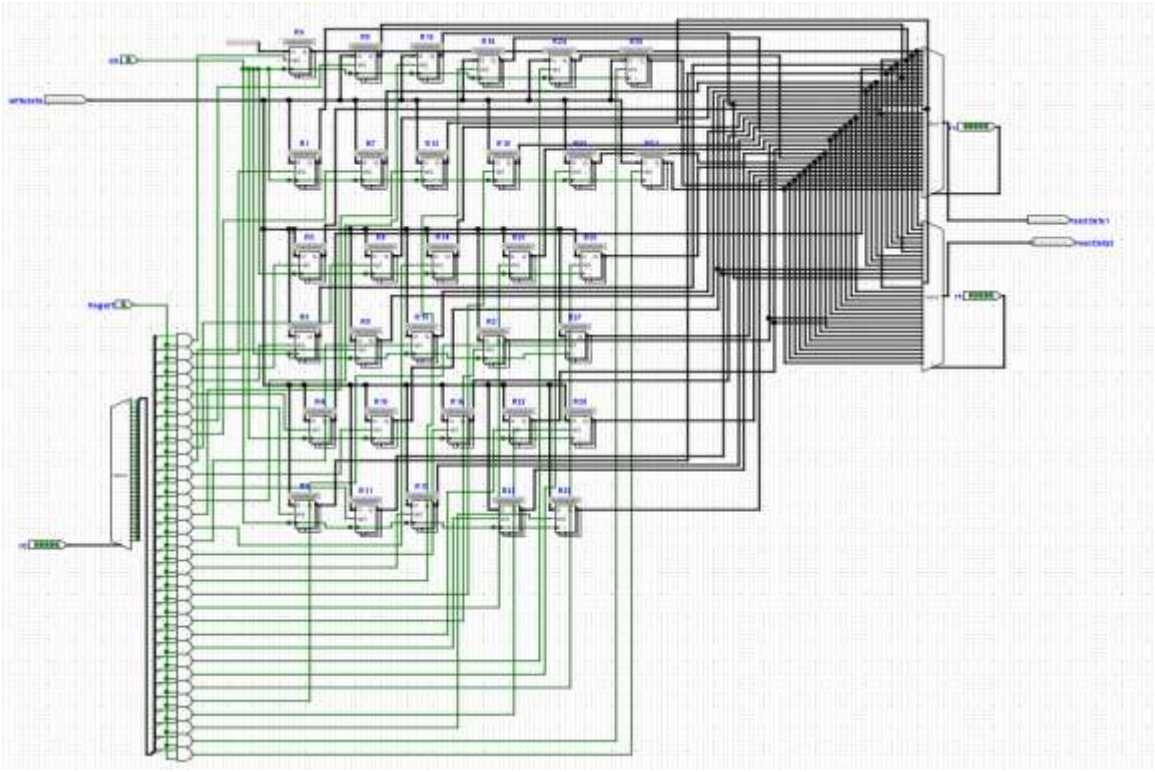
- La salida es la instrucción correspondiente a esa dirección, que será enviada al resto del sistema para ser decodificada y ejecutada.

Para que el direccionamiento funcione correctamente, se realizó una conexión adecuada desde el PC, seleccionando los bits relevantes (normalmente bits 2 a 9) para acceder al índice de instrucción. Esto se debe a que, al ser instrucciones de 4 bytes, las direcciones se alinean naturalmente y los dos bits menos significativos se ignoran (ya que siempre son 00).

El contenido de la ROM se carga de forma manual desde el editor de Logisim, ingresando directamente los códigos binarios o hexadecimales correspondientes a las instrucciones ensambladas previamente en MIPS. Esta memoria no requiere señal de escritura, ya que permanece constante durante la ejecución.

Gracias a esta configuración, la memoria de instrucciones garantiza que el procesador reciba, en cada ciclo de reloj, la instrucción correcta que debe ejecutar según el valor del PC.

4.4 bancoRegistros



El banco de registros es el componente encargado de almacenar valores temporales que las instrucciones del procesador requieren para ejecutar operaciones aritméticas, lógicas o de control de flujo. En la arquitectura MIPS32, este banco está compuesto por 32 registros de 32 bits, identificados como $0 a $31.

En el diseño implementado en Logisim Evolution, el banco de registros se construyó manualmente utilizando una estructura de registros individuales interconectados por medio de multiplexores y decodificadores. Este banco tiene las siguientes señales y características funcionales:

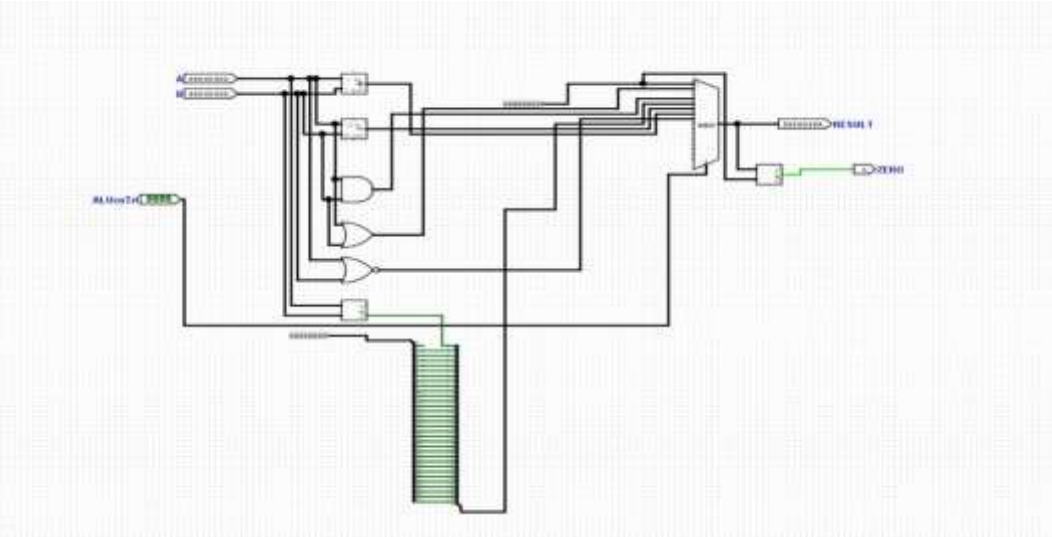
- Entradas:
    - rs y rt: direcciones de los registros fuente que serán leídos.
    - rd: dirección del registro destino donde se escribirá el resultado.
    - RegWrt: señal de control que habilita o bloquea la escritura.
    - WriteData: valor que será almacenado en rd si la escritura está habilitada.
    - clk: señal de reloj para sincronizar la escritura.
- Salidas:
    - ReadData1 y ReadData2: valores leídos desde los registros rs y rt, respectivamente.

Este componente permite leer dos registros simultáneamente y escribir en un tercero durante el mismo ciclo, replicando el comportamiento esperado por las instrucciones tipo R (add, sub, etc.) y algún tipo I (addi, lw, sw).

Además, se respetó la característica del registro $zero (registro 0), el cual siempre contiene el valor cero, sin importar el valor de entrada o la activación de la señal de escritura.

La implementación manual del banco de registros permitió visualizar y controlar de manera precisa el flujo de datos, facilitando el entendimiento del comportamiento interno del procesador y su integración con otros componentes como la ALU y la memoria.

4.5 Alu1



La Unidad Aritmético-Lógica (ALU) es un componente fundamental del procesador, encargada de realizar operaciones matemáticas y lógicas entre dos operandos provenientes del banco de registros. Entre las operaciones que se deben soportar en el conjunto de instrucciones MIPS32 reducido están: suma, resta, AND, OR, NOR y set-on-less-than (SLT).

En el diseño desarrollado en Logisim Evolution, la ALU fue construida utilizando compuertas lógicas básicas, sumadores, comparadores y multiplexores, organizados según la señal de control de operación alu_op. Esta señal de 4 bits indica qué operación debe realizar la ALU según la instrucción ejecutada.

Componentes destacados del diseño:

Entradas:

- A y B, que contienen los operandos leídos desde el banco de registros.
- ALUcnTrl, que selecciona la operación a ejecutar.

Salidas:

- RESULT, que es el valor calculado por la ALU.

- ZERO, una señal que se activa si el resultado es igual a cero (usada principalmente por instrucciones como beq).

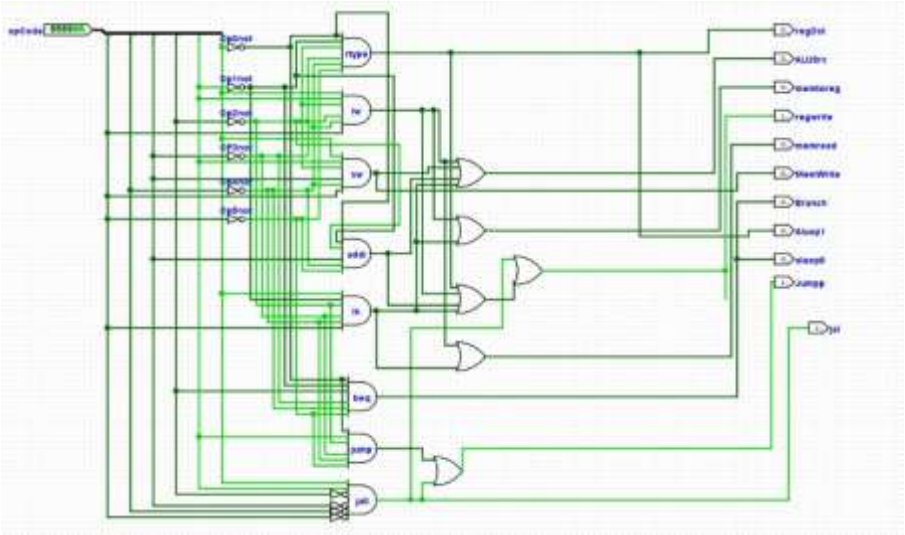
Operaciones implementadas:

- Suma (add)
- Resta (sub)
- AND lógico (and)
- OR lógico (or)
- NOR lógico (nor)
- Comparación (slt)

Estas operaciones están codificadas dentro de un multiplexor de salida, que selecciona el resultado final de acuerdo con la señal alu_op. Además, el comparador de igualdad (zero) facilita la evaluación de condiciones para instrucciones de salto condicional.

Este diseño modular de la ALU permite su integración directa con el resto de la ruta de datos del procesador monociclo, y cubre completamente los requisitos funcionales para ejecutar las 14 instrucciones definidas en el proyecto.

4.6 Unidad de control



La unidad de control es el componente encargado de interpretar el campo opcode de cada instrucción y generar las señales necesarias para coordinar el comportamiento de los

distintos bloques del procesador, tales como la ALU, el banco de registros, la memoria de datos y el flujo del PC. Su función es clave para garantizar que cada instrucción active correctamente los recursos requeridos y se ejecute según su tipo (R, I o J).

En el diseño realizado en Logisim Evolution, la unidad de control fue implementada de forma completamente combinacional, a partir de compuertas lógicas que decodifican los 6 bits del opcode. A partir de esta decodificación, se activan las señales de control correspondientes, según lo especificado en la tabla de verdad del sistema.

Señales de control generadas:

RegDst: selecciona si el registro destino viene del campo rd (tipo R) o del campo rt (tipo I).

ALUSrc: determina si el segundo operando de la ALU proviene del registro rt o de una constante inmediata.

MemtoReg: define si el dato que se escribe en un registro proviene de la ALU o de la memoria de datos.

RegWrite: activa la escritura en el banco de registros.

MemRead: habilita la lectura desde la memoria de datos.

MemWrite: habilita la escritura en la memoria de datos.

Branch: activa la lógica de salto condicional en instrucciones como beq.

Jump: permite realizar saltos incondicionales (j).

ALUOp: conjunto de dos señales (ALUOp1 y ALUOp0) que codifican la operación que debe realizar la ALU.

Jal: activa la escritura del valor PC + 4 en el registro 31, utilizado como dirección de retorno para volver de subrutinas.

Cada línea de control fue diseñada de manera que múltiples instrucciones puedan compartir comportamientos comunes. Por ejemplo, las instrucciones lw y lh comparten la mayoría de sus señales de control, ya que ambas implican una operación de carga desde

memoria; su diferencia radica en cómo se interpreta el dato cargado, lo cual se resuelve fuera de la unidad de control principal.

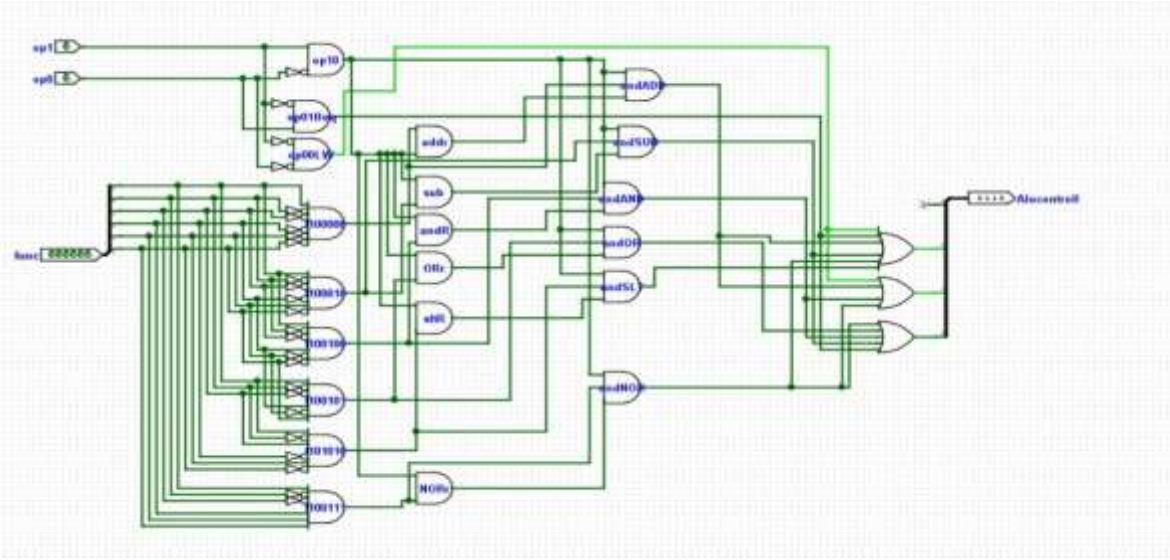
Asimismo, la instrucción addi fue incorporada de forma específica por el equipo 3, como parte de las instrucciones asignadas para su implementación. Esta instrucción requiere una suma inmediata y escritura en un registro tipo I, por lo cual activa las señales ALUSrc, RegWrite, y coloca ALUOp en 00 (operación de suma).

La siguiente tabla resume los valores de las señales de control para cada instrucción implementada, incluyendo las instrucciones adicionales asignadas al equipo (addi y lh):

Instrucción	˝op˝	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp	Jump	Jal
R-type	000000	1	0	0	1	0	0	0	10	0	0
lw	100011	0	1	1	1	1	0	0	00	0	0
sw	101011	X	1	X	0	0	1	0	00	0	0
beq	000100	X	0	X	0	0	0	1	01	0	0
j	000010	X	X	X	0	0	0	0	XX	1	0
jal	000011	X	X	X	1	0	0	0	XX	1	1

Cabe destacar que la señal jr, correspondiente a la instrucción jump register, no es generada directamente en esta unidad de control principal, ya que su activación depende del campo opcode y del campo funct, que fue implementada mediante un subcircuito adicional de detección combinacional, descrito en la sección 4.9, el cual activa la redirección del PC cuando se detecta una instrucción jr.

4.7 ALU control



La unidad de control de la ALU es un módulo complementario a la unidad de control principal, encargado de traducir la señal ALUOp junto con el campo funct (en instrucciones tipo R) en una señal específica de control para la ALU, denominada ALUControl. Esta señal define qué operación exacta debe ejecutar la ALU, como suma, resta, operaciones lógicas o comparación.

En el diseño implementado en Logisim Evolution, esta unidad recibe dos tipos de entradas:

ALUOp: señal de 2 bits generada por la unidad de control principal.

funct: campo de 6 bits proveniente del campo final de la instrucción tipo R.

Comportamiento según tipo de instrucción:

ALUOp igual a 00: se ejecuta una suma, usada por instrucciones como lw, sw, addi o lh.

ALUOp igual a 01: se realiza una resta, usada por beq.

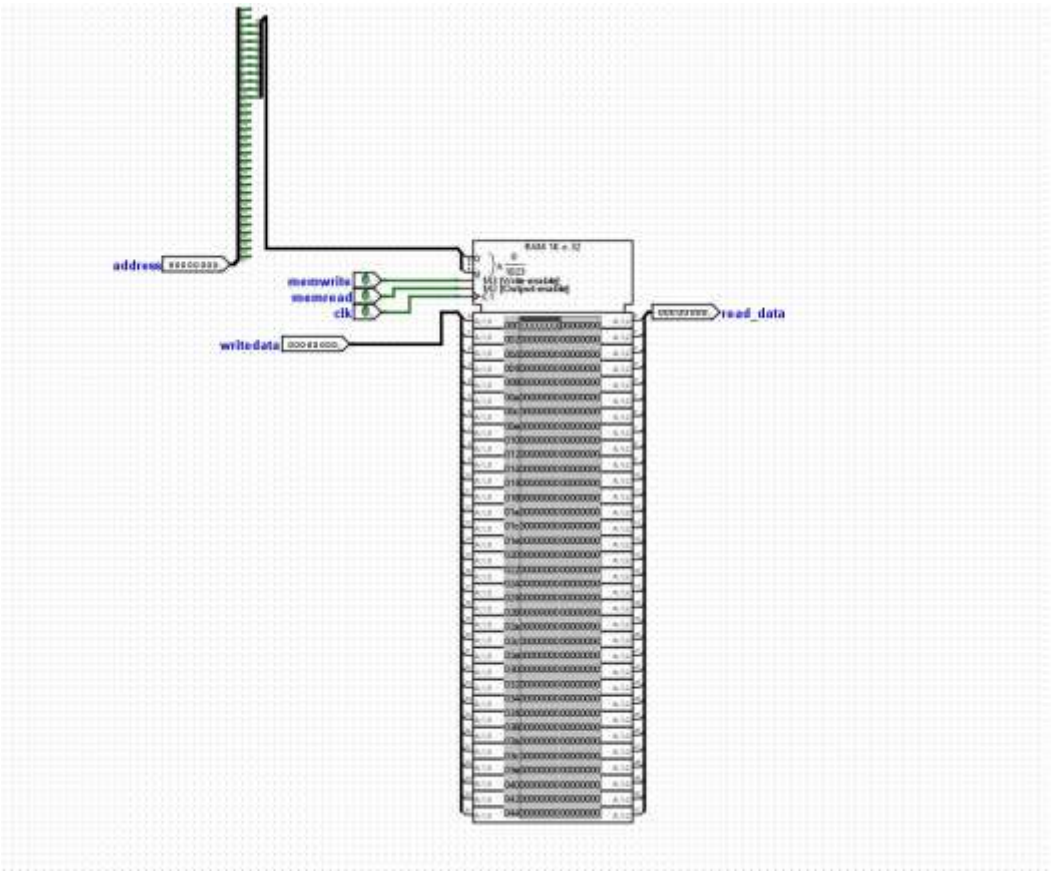
ALUOp igual a 10: se interpreta el campo funct para determinar la operación exacta (solo en instrucciones tipo R).

ALUOp	Funct	Operación	ALU control(asignación #3)
OO	XXXXXX	LW/SW	11O(ADD)
O1	XXXXXX	BEQ	1O1(SUB)
1O	1OOOOO	ADD	11O(ADD)
1O	1OOO1O	SUB	1O1(SUB)
1O	1OO1OO	AND	011(AND)
1O	1OO1O1	OR	OO1(OR)
1O	1OO111	NOR	111(NOR)
1O	1O1O1O	SLT	1OO(SLT)

Estas combinaciones son implementadas con compuertas AND, OR y comparadores de igualdad, que activan una única salida binaria de 4 bits (ALUControl) para dirigir el funcionamiento interno de la ALU.

La señal ALUControl generada se conecta directamente a la ALU para seleccionar la operación deseada, mientras que ALUOp es controlada por la unidad de control principal, que la configura en función del tipo de instrucción.

4.8 DataMemory



La memoria de datos es el componente encargado de almacenar y recuperar valores durante la ejecución de instrucciones de acceso a memoria, como lw, sw, y las instrucciones adicionales lh y otras variantes de carga. Su función es clave para la manipulación dinámica de datos dentro del procesador.

En el diseño implementado en Logisim Evolution, se utilizó un módulo de RAM de 1K x 32 bits, es decir, una memoria con 1024 posiciones de 32 bits cada una. Esta memoria puede ser leída o escrita dependiendo de las señales de control generadas por la unidad de control.

Señales y funcionamiento:

Entradas:

- address: dirección de memoria donde se desea leer o escribir (proviene de la ALU).
- writedata: dato que se desea almacenar en memoria (proveniente del banco de registros).
- memwrite: señal de control que habilita la escritura en memoria (activa M1).
- memread: señal de control que habilita la lectura desde memoria (activa M2).
- clk: señal de reloj que sincroniza las operaciones de escritura.

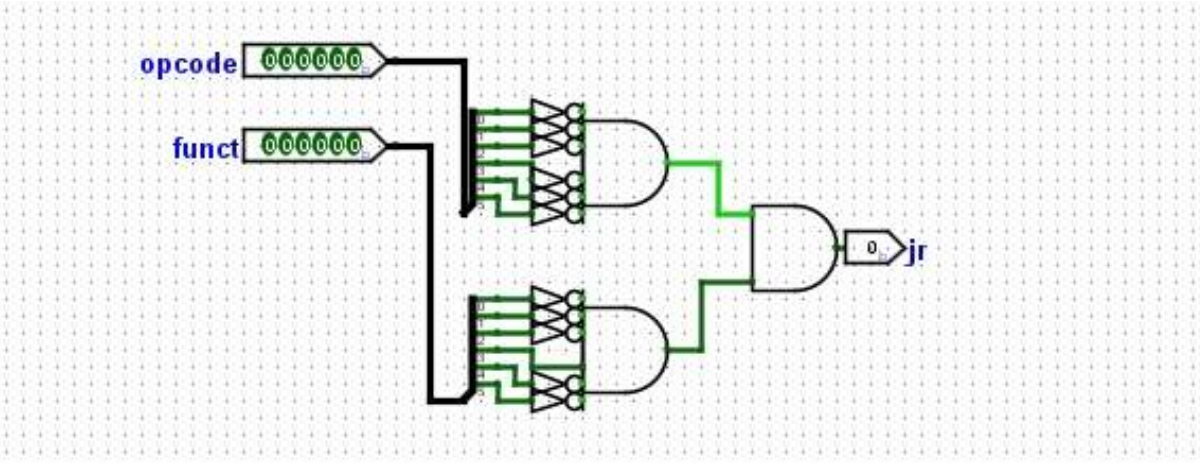
Salida:

- read_data: valor leído desde la dirección indicada (solo si memread está activa).

Comportamiento:

- Cuando memread está activa y memwrite inactiva, la memoria entrega el contenido almacenado en la dirección indicada por address, sin necesidad de esperar al flanco de reloj.
- Cuando memwrite está activa, la memoria almacena el valor presente en writedata en la dirección indicada por address en el siguiente flanco activo del reloj (clk).
- Si ambas señales (memread y memwrite) están desactivadas, la memoria permanece inactiva.

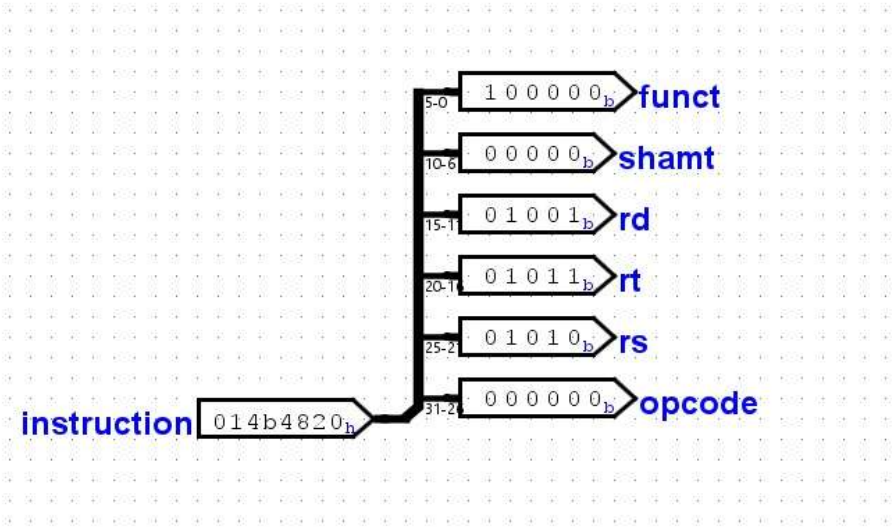
4.9 JR



Este subcomponente fue implementado para reconocer específicamente la instrucción jr (jump register), que forma parte del conjunto de instrucciones tipo R. El circuito compara el campo opcode, que debe ser 000000, y el campo funct, que debe corresponder a 001000. Si ambas condiciones se cumplen, se activa la señal de control jr.

Esta señal permite modificar el valor del PC tomando directamente el contenido de un registro, facilitando así saltos a direcciones almacenadas en tiempo de ejecución (retornos de subrutina o saltos dinámicos). Es un bloque esencial para implementar cambios de flujo controlados mediante registros, como los utilizados en estructuras de funciones.

4.10 ibreakdown



Este componente se encarga de descomponer la instrucción MIPS de 32 bits en sus respectivos campos, necesarios para la decodificación y ejecución correcta en el procesador monociclo. La entrada es una instrucción en formato binario, y sus salidas son los campos extraídos conforme al formato R, I o J.

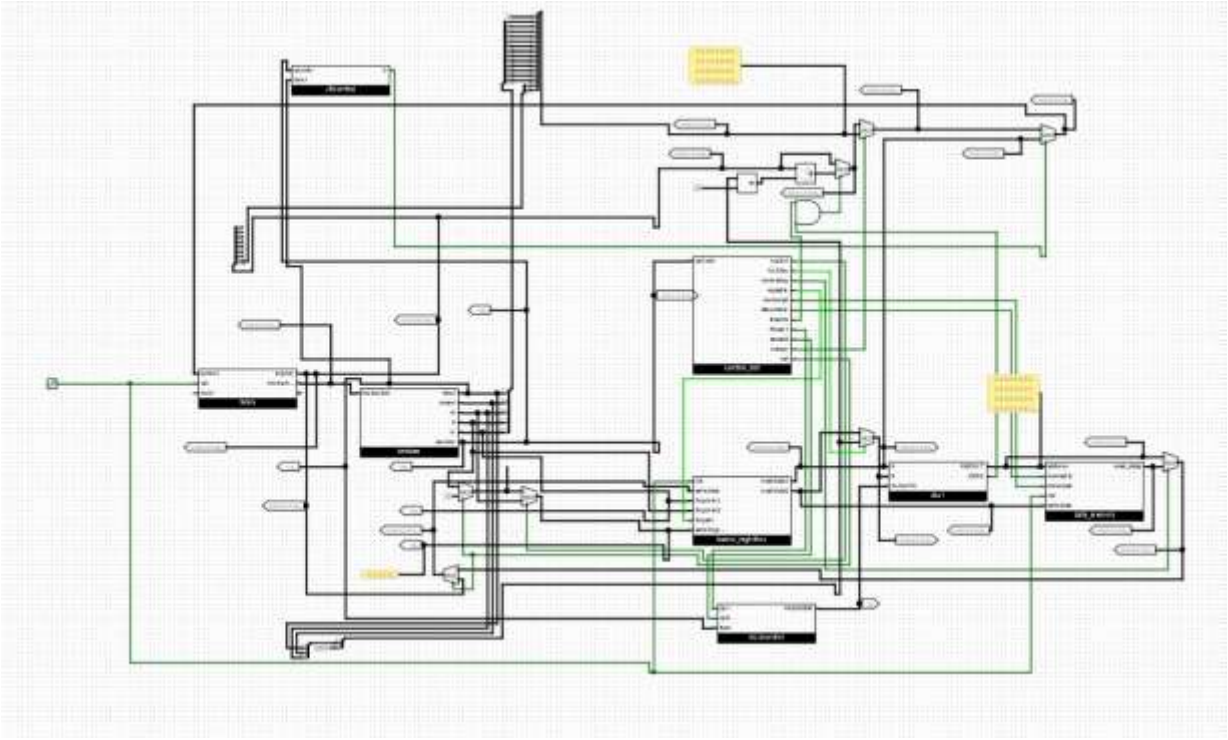
Para el caso mostrado (formato R), la instrucción se divide de la siguiente manera:

- opcode (bits 31–26): Indica el tipo general de instrucción.
- rs (25–21): Registro fuente 1.
- rt (20–16): Registro fuente 2.
- rd (15–11): Registro destino.
- shamt (10–6): Cantidad de desplazamiento (para operaciones shift).

- funct (5–0): Especifica la operación exacta a realizar (cuando opcode es 000000).

Esta descomposición es esencial para que la unidad de control y el datapath puedan determinar cómo actuar sobre la instrucción. El Instruction Breakdown actúa como el primer paso del proceso de ejecución, permitiendo que las señales de control se generen con base en los campos individuales de la instrucción.

4.11 Main



El circuito Main representa la integración total de todos los módulos que componen el procesador monociclo MIPS. En este circuito está la ruta de datos, las unidades de control y los bloques auxiliares, permitiendo la ejecución correcta de instrucciones en un único ciclo de reloj.

Este módulo central incluye los siguientes componentes clave:

- Fetch: contiene el PC, el sumador PC+4 y la memoria de instrucciones. Es responsable de suministrar la siguiente instrucción a ejecutar.

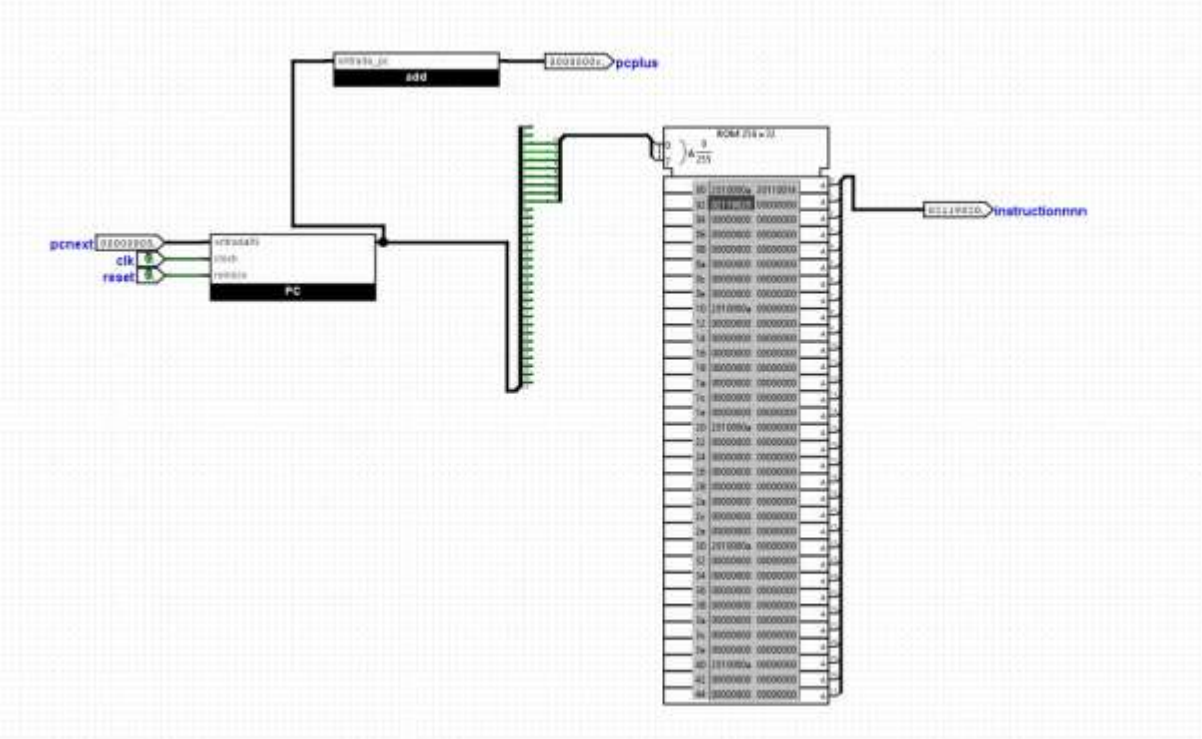
- Instruction Breakdown: descompone la instrucción de 32 bits en sus campos individuales (opcode, rs, rt, rd, shamt, funct), esenciales para la decodificación.
- Unidad de control principal (control_unit): genera las señales de control globales a partir del campo opcode, que determinan el comportamiento del procesador para cada instrucción.
- ALU Control: interpreta las señales ALUOp y funct para generar la operación específica que debe ejecutar la ALU.
- Banco de registros: permite leer dos registros fuente (rs y rt) y escribir en un registro destino (rd o rt), según el tipo de instrucción.
- ALU: realiza operaciones aritmético-lógicas (como suma, resta, AND, OR, SLT, NOR) con base en los datos del banco de registros y la señal ALUControl.
- Data Memory: permite la lectura o escritura de datos en memoria para instrucciones como lw, sw y lh, coordinada por las señales MemRead y MemWrite.
- Multiplexores y lógica de flujo: dirigen los datos y controlan el siguiente valor del PC en función de las instrucciones de salto (j, jal, jr) o condicional (beq).
- JRControl: módulo dedicado a detectar la instrucción jr y redirigir el flujo del PC al valor almacenado en un registro.
- Conexión lógica general: todas las unidades están conectadas mediante buses de 32 bits y señales de control de 1 bit, lo que asegura un flujo de datos preciso y sin interferencias.

5. Decisiones de diseño

5.1 Simplificación del manejo de datos mediante la ROM de instrucciones

Se optó por cargar la matriz con sus datos en el mismo instruction memory para no tenerlos que añadir en el data, esto se logró mediante la instrucción addi, así la instrucción hex para la ROM se puede cargar directamente con todos sus componentes y activar el procesador para ver sus respectivas salidas en el banco de registros, la idea de todo esto fue principalmente para manejar todos los datos en un solo componente y asi simplificar sus ejecuciones

5.2 FETCH



Se optó por encapsular la etapa de búsqueda de instrucciones en un subcircuito independiente llamado Fetch, el cual incluye PC, el sumador (PC + 4) y la memoria de instrucciones. Esta decisión mejora la organización del diseño, facilita la depuración y permite integrar de forma más clara las instrucciones de salto como j, jal y jr.

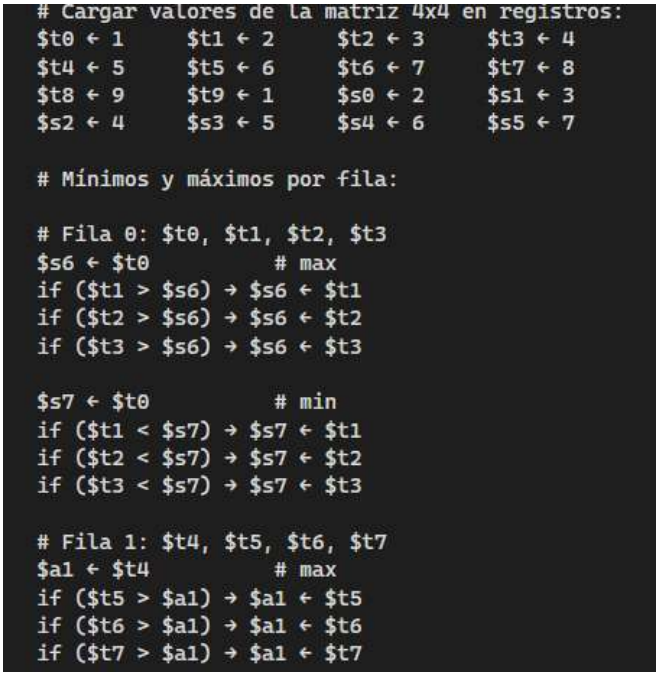
6. Descripción del programa en MIPS

> El programa debe tomar una matriz de 4×4 números enteros con valores entre -100 y 100, y determinar cuáles son el mayor y el menor número almacenado en cada fila y en cada columna de la matriz.

Este problema plantea trabajar sobre una matriz de 4x4 números enteros, con el objetivo de identificar el valor máximo y mínimo en cada fila y en cada columna. Requiere

organizar cuidadosamente el recorrido por la matriz, comparar valores de forma controlada y prestar atención tanto al diseño del procesador como a la lógica del programa en MIPS.

6.1 Descripción en alto nivel



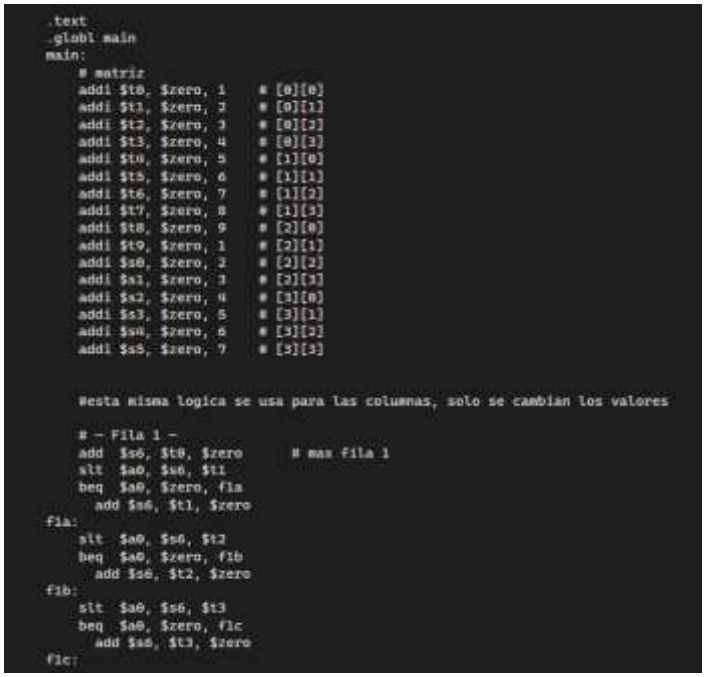
```
# Cargar valores de la matriz 4x4 en registros:
$t0 ← 1     $t1 ← 2     $t2 ← 3     $t3 ← 4
$t4 ← 5     $t5 ← 6     $t6 ← 7     $t7 ← 8
$t8 ← 9     $t9 ← 1     $s0 ← 2     $s1 ← 3
$s2 ← 4     $s3 ← 5     $s4 ← 6     $s5 ← 7

# Mínimos y máximos por fila:

# Fila 0: $t0, $t1, $t2, $t3
$s6 ← $t0           # max
if ($t1 > $s6) → $s6 ← $t1
if ($t2 > $s6) → $s6 ← $t2
if ($t3 > $s6) → $s6 ← $t3

$s7 ← $t0           # min
if ($t1 < $s7) → $s7 ← $t1
if ($t2 < $s7) → $s7 ← $t2
if ($t3 < $s7) → $s7 ← $t3

# Fila 1: $t4, $t5, $t6, $t7
$a1 ← $t4           # max
if ($t5 > $a1) → $a1 ← $t5
if ($t6 > $a1) → $a1 ← $t6
if ($t7 > $a1) → $a1 ← $t7
```

Antes de desarrollar el programa en MIPS, se organizó la lógica general del algoritmo utilizando un pseudocódigo que simula las operaciones paso a paso. En lugar de usar estructuras como matrices tradicionales, se asignaron manualmente los valores de la matriz a registros individuales, permitiendo un control directo sobre cada celda.

El pseudocódigo refleja cómo se comparan los valores dentro de cada fila y luego dentro de cada columna para identificar el mayor y el menor. La estructura se basa en asignar un valor inicial como candidato a máximo y mínimo, y luego actualizarlo si se encuentra un número mayor o menor en el recorrido.

Link del pseudocódigo completo: https://github.com/Joselito17821/Diseno-y-verificacion-del-procesador-monociclo/blob/main/pseudocodigo.txt

6.2 Código MIPS



```
.text
.globl main
main:
    # matriz
    addi $t0, $zero, 1    # [0][0]
    addi $t1, $zero, 2    # [0][1]
    addi $t2, $zero, 3    # [0][2]
    addi $t3, $zero, 4    # [0][3]
    addi $t4, $zero, 5    # [1][0]
    addi $t5, $zero, 6    # [1][1]
    addi $t6, $zero, 7    # [1][2]
    addi $t7, $zero, 8    # [1][3]
    addi $t8, $zero, 9    # [2][0]
    addi $t9, $zero, 1    # [2][1]
    addi $s0, $zero, 2    # [2][2]
    addi $s1, $zero, 3    # [2][3]
    addi $s2, $zero, 4    # [3][0]
    addi $s3, $zero, 5    # [3][1]
    addi $s4, $zero, 6    # [3][2]
    addi $s5, $zero, 7    # [3][3]


    #esta misma logica se usa para las columnas, solo se cambian los valores

    # - Fila 1 -
    add  $s6, $t0, $zero      # max fila 1
    slt  $a0, $s6, $t1
    beq  $a0, $zero, f1a
      add $s6, $t1, $zero
f1a:
    slt  $a0, $s6, $t2
    beq  $a0, $zero, f1b
      add $s6, $t2, $zero
f1b:
    slt  $a0, $s6, $t3
    beq  $a0, $zero, f1c
      add $s6, $t3, $zero
f1c:
```

El programa fue desarrollado en lenguaje ensamblador MIPS utilizando el entorno MARS. Se optó por cargar directamente los datos de la matriz en registros usando instrucciones addi, lo que permite mantener todos los valores iniciales en la memoria de instrucciones (ROM) sin necesidad de utilizar la memoria de datos (RAM). Esta decisión simplifica la ejecución del procesador en Logisim Evolution y facilita el seguimiento de resultados directamente en el banco de registros.

La lógica del código recorre fila por fila y luego columna por columna, utilizando instrucciones como slt y beq para comparar valores y determinar máximos y mínimos de forma controlada. Aunque el enfoque es secuencial, se estructura en bloques claramente separados por filas (y columnas), lo que mejora la legibilidad y el análisis.

Link del código completo: https://github.com/Joselito17821/Diseno-y-verificacion-del-procesador-monociclo/blob/main/codigoMIPS.asm

6.3 Código de maquina

Una vez verificado el funcionamiento del programa en MIPS, se procedió a convertir cada instrucción ensamblador a su representación en código máquina en formato hexadecimal. Esta codificación fue ingresada manualmente en la memoria de instrucciones (ROM) del procesador implementado en Logisim Evolution, lo que permite que el procesador ejecute directamente las instrucciones sin necesidad de un ensamblador externo.

Link del código de maquina completo: https://github.com/Joselito17821/Diseno-y-verificacion-del-procesador-monociclo/blob/main/codigoMaquina.txt

7. Simulación

A continuación, se presenta un video que muestra la simulación completa del funcionamiento del procesador monociclo y del programa ensamblador que calcula máximos y mínimos en una matriz. En el video se puede observar el recorrido de la señal, el cambio en los registros y la lógica interna ejecutándose paso a paso: https://www.youtube.com/watch?v=-MYcStL1nbE

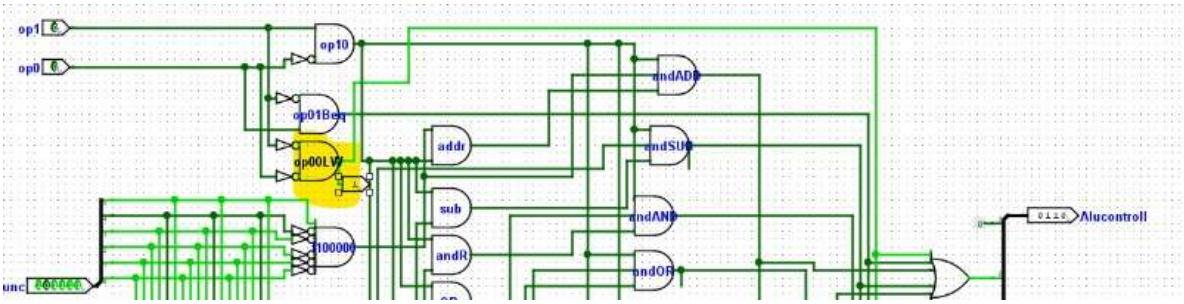
Si quieres probar tu propia simulación del sistema o tener en tus manos este procesador monociclo descarga el siguiente archivo (recuerda ejecutar el archivo .circ en el software de logisim-evolution y el .asm en MARS): https://github.com/Joselito17821/Diseno-y-verificacion-del-procesador-monociclo

8. Pruebas de cada instrucción

Transferencia de datos**:**

- SW (store word): Guarda el contenido de un registro en una dirección específica de la memoria.

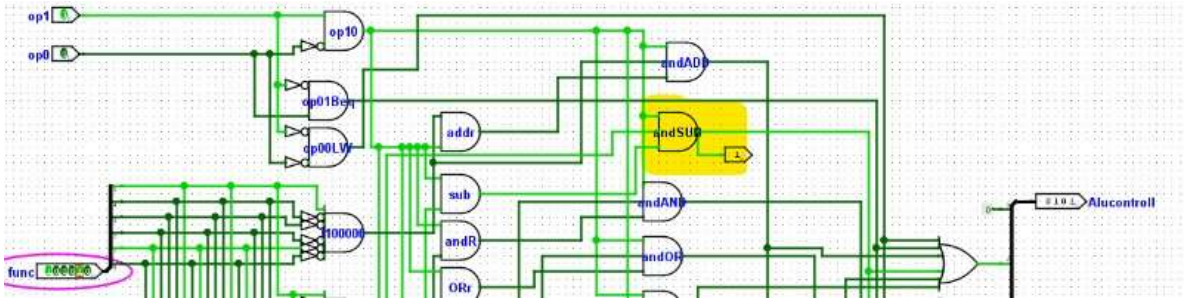
- LW (load word): Carga un dato de 32 bits desde una dirección de memoria hacia un registro.



Las instrucciones LW/SW al activarse en dan una salida en el alucontrol 0110 es decir un ADD, se pueden activar cuando op es 00 y la funct xxxxxx.

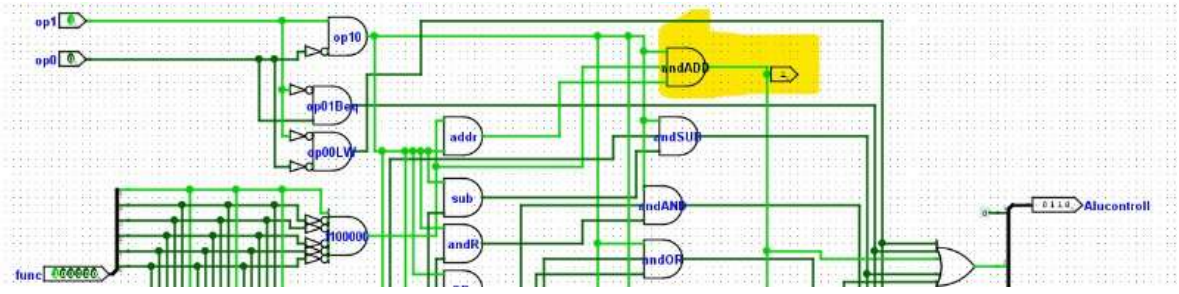
<u>Aritmético-lógicas:</u>

- SUB: Resta dos registros y almacena el resultado en un tercer registro.



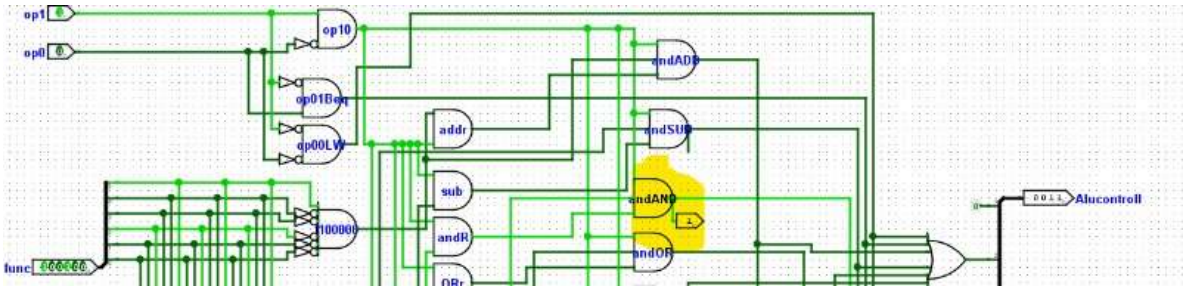
La instrucción SUB se puede activar cuando op está en 10 y la funct en 100010 o cuando op es 01 y la funct xxxxxx para dar la señal respectiva de la operación 0101.

- ADD: Suma dos registros y almacena el resultado en un tercer registro.



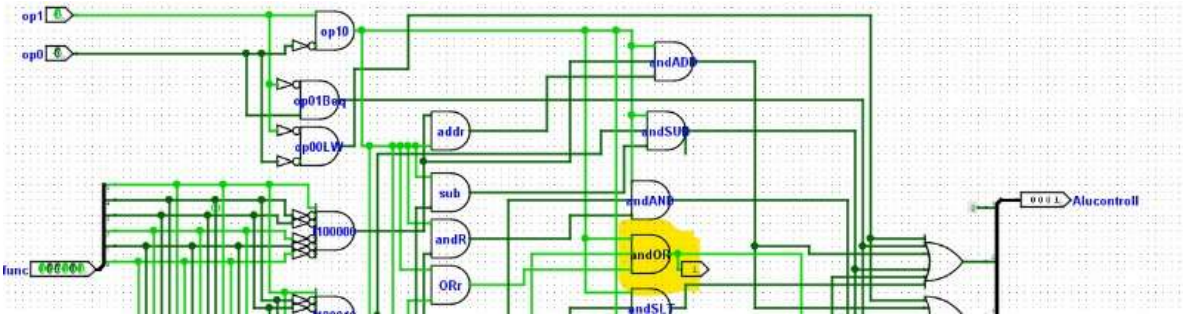
La instrucción ADD se activa en el alucontrol cuando el op está en 10 y la funct en 100000 esto daría la señal de la operación que seria 0110.

- AND: Realiza una operación AND bit a bit entre dos registros.



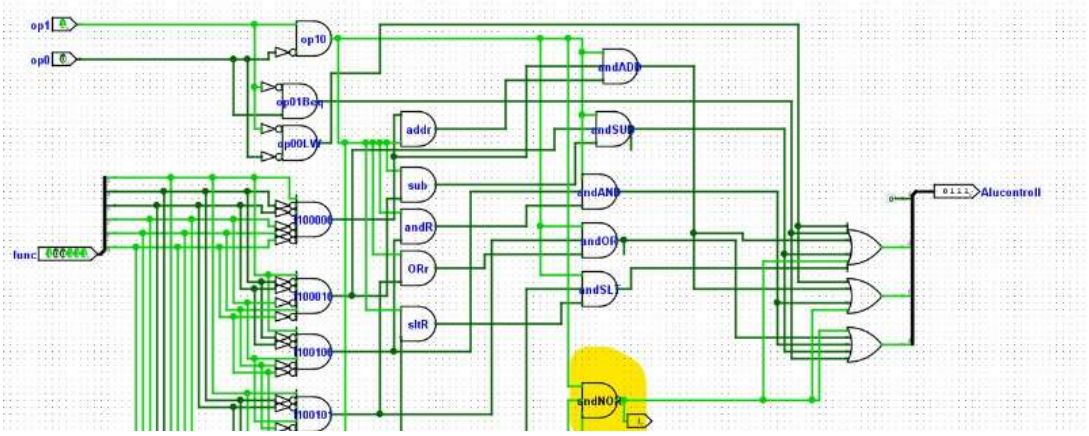
La instrucción AND se activa en el alucontrol cuando el op está 10 y la funct es 100100 y arroja la salida 0011.

- OR: Realiza una operación OR bit a bit entre dos registros.



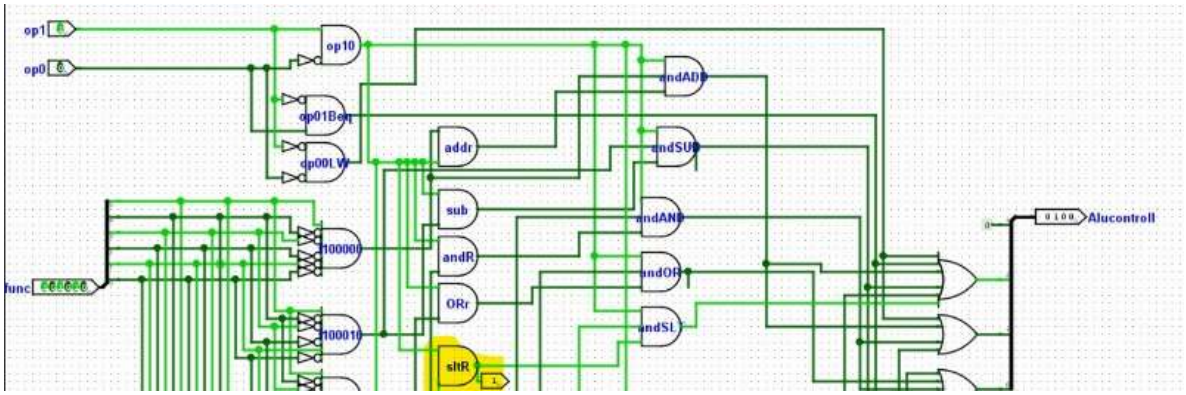
La instrucción OR se activa en el alucontrol cuando el op está 10 y la funct 100101 para dar una salida 0001.

- NOR: Realiza una operación NOR bit a bit entre dos registros.



La instrucción NOR se activa en el alucontrol cuando el op está 10 y funct 100111 para una salida 0111.

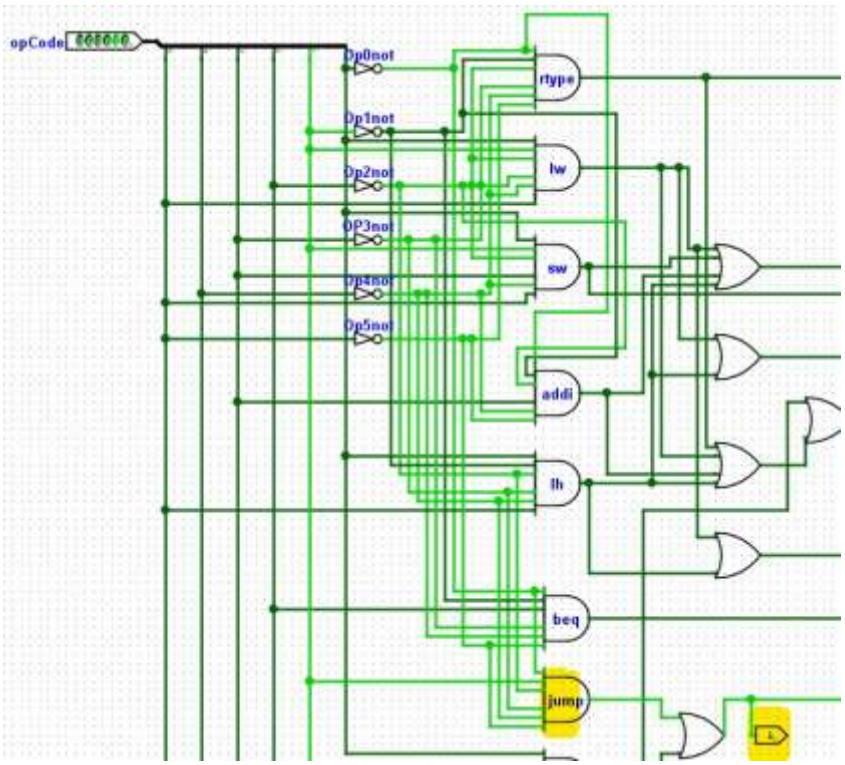
- SLT (set-on-less-than): Coloca 1 si el primer operando es menor que el segundo, de lo contrario coloca 0.



La instrucción SLT se activa en el alucontrol cuando el op está 10 y funct es 101010 para una salida 0100.

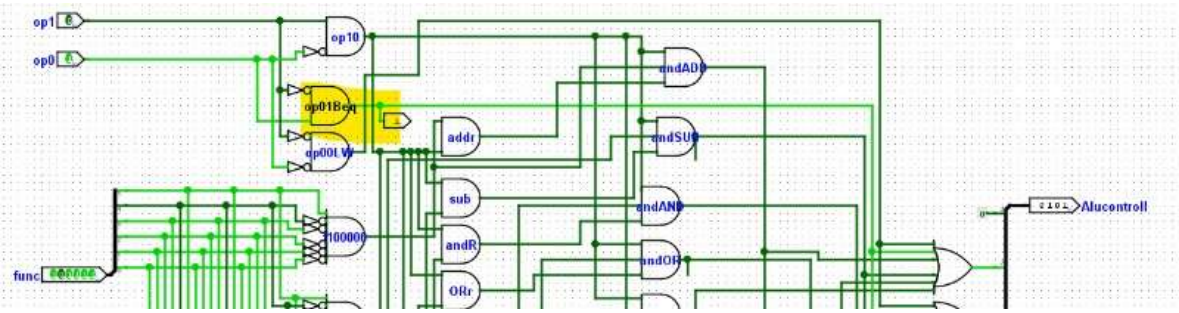
<u>Control de flujo:</u>

- J (jump): Salta incondicionalmente a una dirección específica de la memoria de instrucciones.



Una instrucción J o JUMP se activa cuando el opCode es 000010.

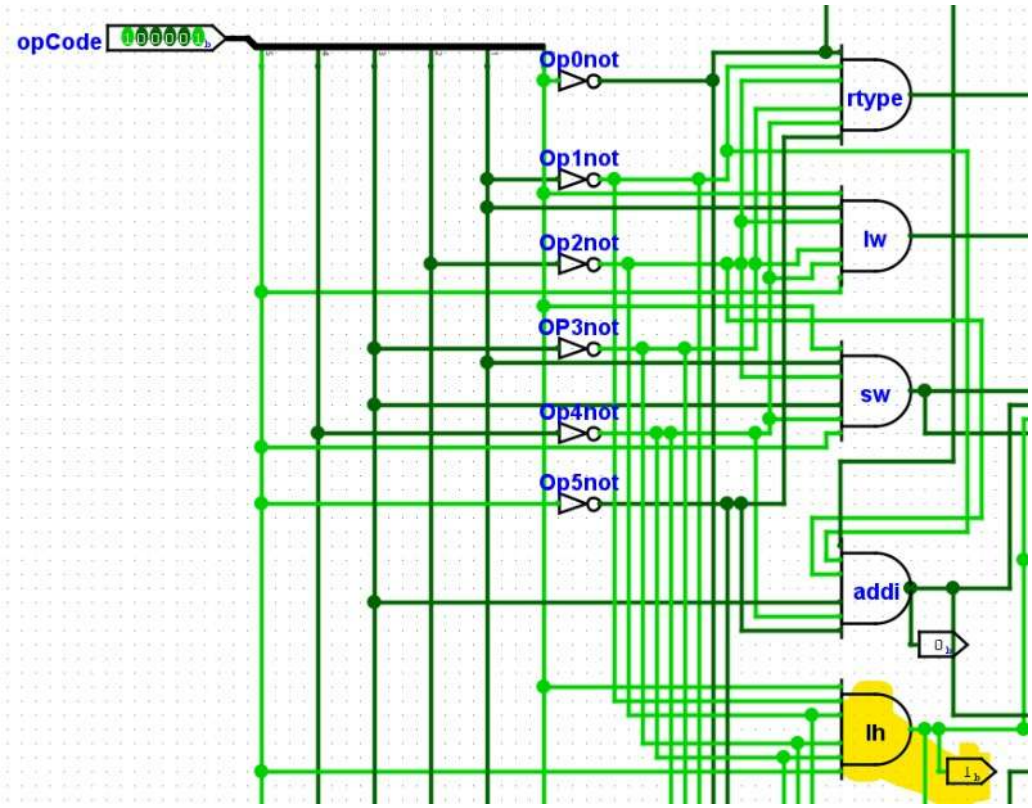
- BEQ (branch on equal): Salta a una dirección relativa si los dos registros comparados son iguales.



La instrucción BEQ se activa cuando op es 01 y la Funct es xxxxxx activando una señal sub con salida 0101.

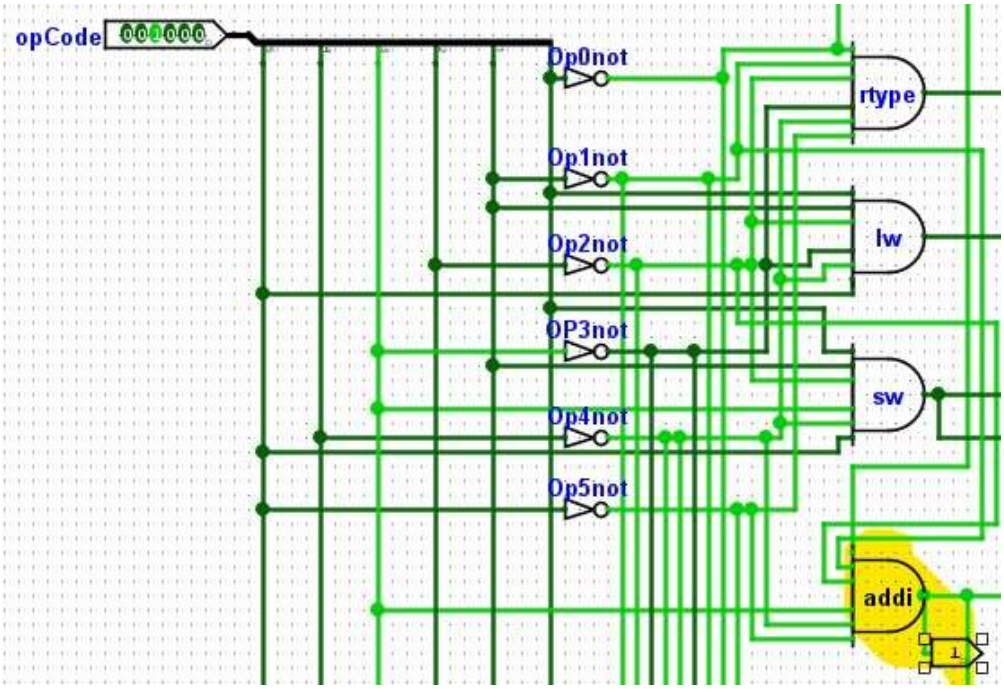
<u>Instrucciones adicionales asignadas:</u>

- LH (load halfword): Carga 16 bits desde memoria (la mitad de una palabra) hacia un registro.



La instrucción LH se activa cuando el opCOde es 100001.

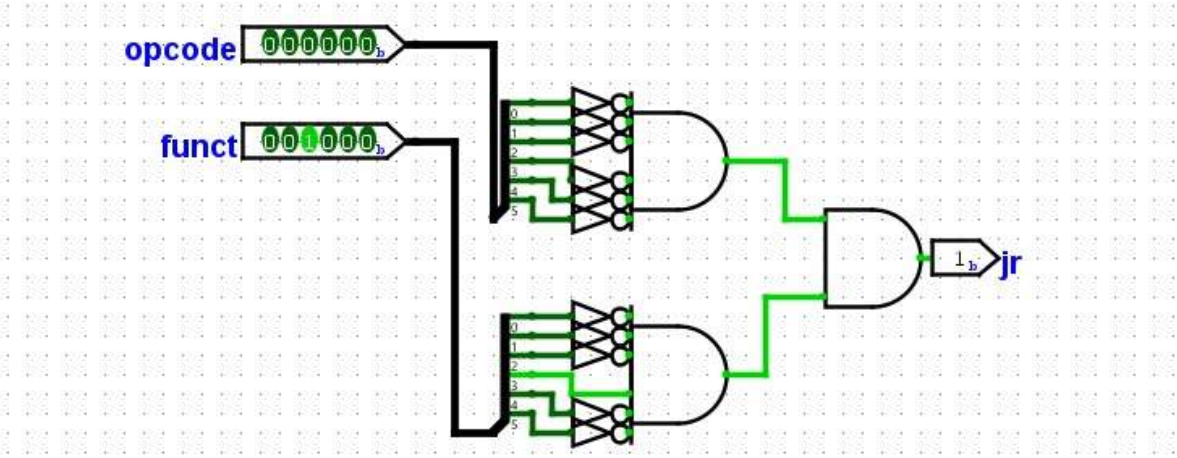
- ADDI (add immediate): Suma un valor inmediato (constante) a un registro y almacena el resultado.



La instrucción ADDi se activa cuando el opCode es 001000.

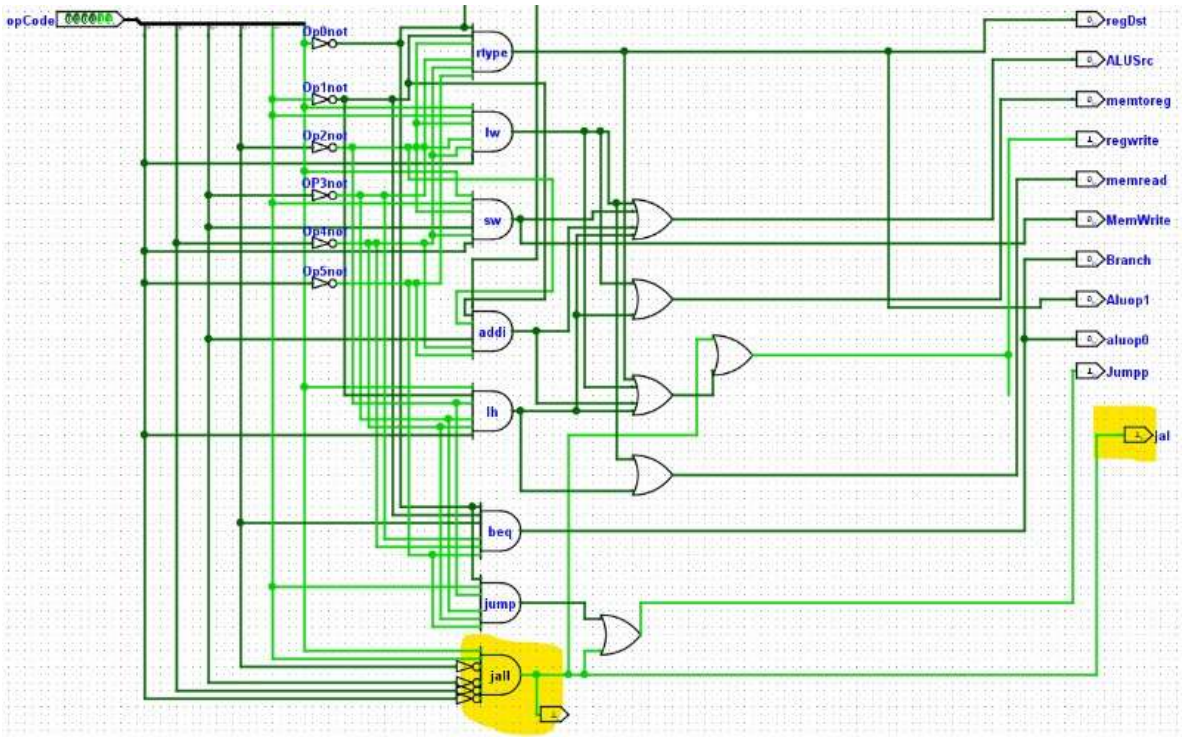
<u>Instrucciones de salto a subrutinas:</u>

- JR (jump register): Salta a la dirección almacenada en un registro (retorno de subrutinas).



La instrucción JR se activa cuando opcode es 000000 y funct 001000.

- JAL (jump and link): Salta a una dirección y guarda la dirección de retorno en el registro 31.



Una instrucción JAL se activa cuando el opCode es 000011.

9. Resultados de ejecución

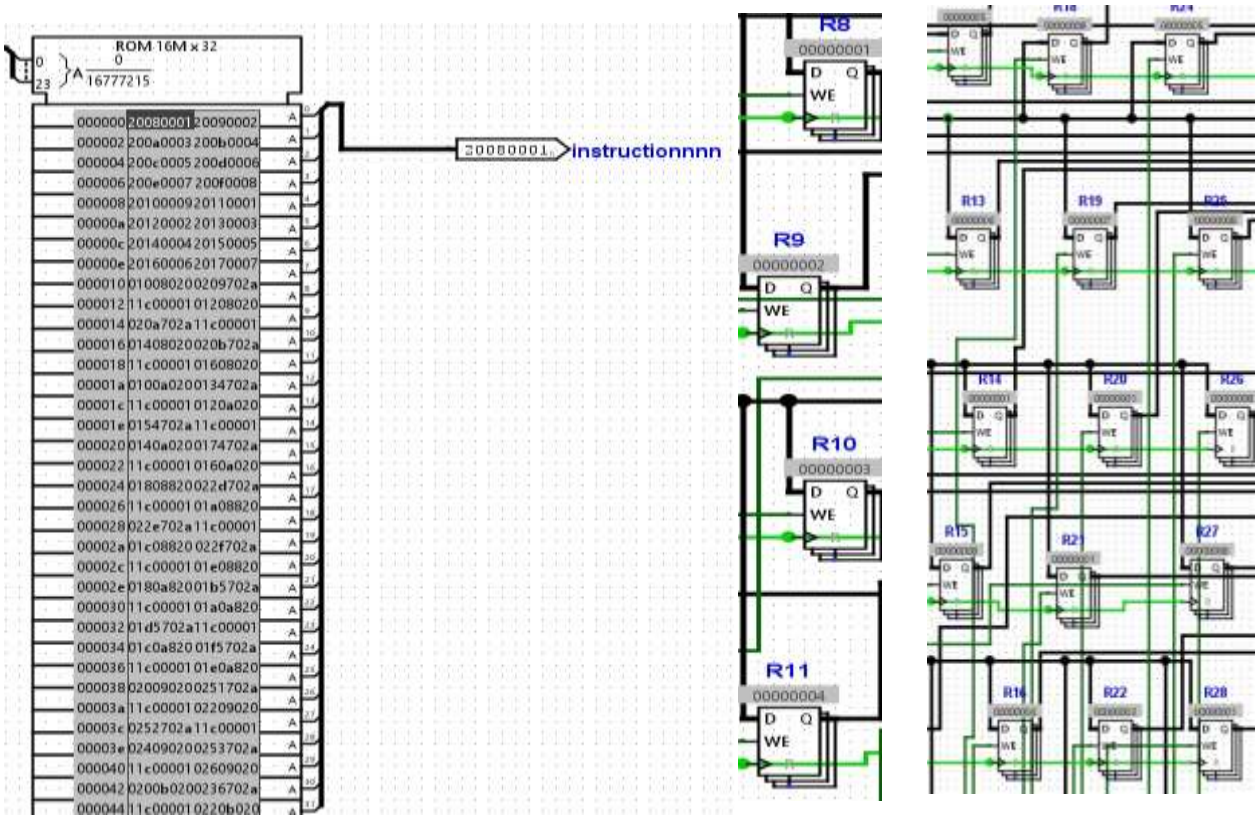
Estos son los resultados de la ejecución final del código diseñado para resolver el problema asignado a nuestro equipo, que consistía en identificar los valores máximos y mínimos por fila y por columna en una matriz de 4x4 números enteros.

Los resultados fueron obtenidos tras ejecutar correctamente el programa en MARS, utilizando los registros para almacenar tanto los datos originales de la matriz como los resultados de las comparaciones realizadas.

El programa fue validado paso a paso mediante simulación, observando cómo cada instrucción era ejecutada en el procesador monociclo y cómo los registros se actualizaban con los valores esperados.

Esto permitió comprobar que tanto el código ensamblador como la arquitectura diseñada en

Logisim Evolution funcionaban de forma coherente y precisa para cumplir el objetivo propuesto.



10. Observaciones

- Una de las cosas más curiosas fue darnos cuenta de lo fácil que es perderse entre tantas conexiones en el circuito. Al principio parecía todo entendible, pero cuando empezamos a unir módulos, si algo fallaba, encontrar el error no era tan obvio como pensábamos.
- No esperábamos fue ver cómo decisiones pequeñas, como agrupar componentes en subcircuitos o usar nombres claros, terminaban haciendo una gran diferencia cuando ya el diseño estaba más avanzado. Fue un recordatorio de que la organización no solo es estética, también es práctica.
- Mientras realizamos el proyecto, nos dimos cuenta de que un pequeño error podía desajustar todo, incluso si no parecía importante al principio. Eso nos hizo tomarle

más respeto al diseño, porque entendimos que no se trata solo de “hacer que funcione”, sino de pensar con cuidado cada conexión.

11. Conclusiones

- A lo largo del proyecto se logró una comprensión sólida del diseño de cada componente del procesador y de cómo interactúan entre sí. Aunque al integrarlos en el circuito principal (Main) surgieron errores puntuales, se logró mantener una estructura limpia y ordenada que permitió verificar el funcionamiento correcto de cada módulo. Esta experiencia no solo fortaleció nuestros conocimientos técnicos, sino que también nos enseñó la importancia de la organización, la paciencia y el análisis detallado en el desarrollo de sistemas digitales.
- A través de la implementación y prueba de 14 instrucciones MIPS, se comprobó que es posible construir un procesador funcional desde cero utilizando herramientas como Logisim y MARS, reforzando conceptos vistos en clase.
- El desarrollo de este laboratorio fue una oportunidad para aplicar conocimientos técnicos en un proyecto completo, desarrollar habilidades de trabajo autónomo y enfrentar de forma realista los errores y desafíos propios de la ingeniería.

12. Video explicativo

A continuación, se presenta un video para dar una explicación general del funcionamiento del código y todos sus componentes: https://youtu.be/-EgdVVQQncs

13. Referencias

- Patterson, D. A., & Hennessy, J. L. (2014). Computer organization and design: The hardware/software interface (5th ed.). Morgan Kaufmann.

- Universidad de Antioquia – Facultad de Ingeniería. (2025). Material del curso Arquitectura de Computadores y Laboratorio. Medellín, Colombia**.**