LABORATORIO 3 - SIMULACIÓN DE PROCESADOR MONOCICLO

ANDRÉS FELIPE GIRALDO YUSTI[1] **y JUAN CAMILO VELEZ**[2]

[1] *Universidad de Antioquia, Facultad de Ingenierí , andres.giraldoy@udea.edu.co*
[2] *Universidad de Antioquia, Facultad de Ingeniería , camilo.velez4@udea.edu.co*

JOHN BYRON BUITRAGO PANIAGUA

1. Introducción

En esta práctica se documentan las decisiones de diseño, herramientas empleadas y el proceso de desarrollo de un procesador monociclo MIPS de 32 bits, implementado utilizando la herramienta Logisim Evolution. El diseño del procesador se realizó en base a los conocimientos adquiridos en el curso de Arquitectura de Computadores, aplicando conceptos clave como el manejo de instrucciones tipo R, I y J, y la integración de componentes fundamentales como la ALU, el banco de registros, el contador de programa y la memoria.

Una de las condiciones del proyecto fue la modificación de ciertas instrucciones del conjunto MIPS, específicamente las instrucciones tipo R add y and. Según las indicaciones del ejercicio, debíamos reemplazar el campo funct de estas instrucciones por los dos últimos dígitos de nuestras cédulas. En nuestro caso, los valores correspondientes eran 20 y 36. Como parte del diseño, se decidió asignar el número 20 a la instrucción add, modificando así su identificador funct, mientras que la instrucción and conservó su valor original, ya que su funct predeterminado coincide con el número 36. Este cambio implicó una adaptación de la ALU de control, asegurando que el procesador funcione con el nuevo formato.

Adicionalmente, se desarrolló un algoritmo en pseudocódigo (usando Python) y en lenguaje ensamblador MIPS (utilizando el simulador MARS), cuya función consiste en determinar cuántos elementos de un arreglo son iguales al primer elemento. Este algoritmo se diseñó respetando las restricciones impuestas por el subconjunto de instrucciones disponibles en el procesador.

Finalmente, el algoritmo ensamblador fue probado en el procesador simulado en Logisim, verificando el funcionamiento correcto de la arquitectura, el cumplimiento de las instrucciones personalizadas y la correcta ejecución del algoritmo asignado. Este informe presenta cada etapa del desarrollo, desde el diseño del procesador hasta la verificación de que este funcione mediante pruebas prácticas.

2. Panteamento del problema y objetivos

2.1. Problema

En el diseño e implementación de arquitecturas de procesadores, uno de los principales retos está en comprender la estructura interna y el funcionamiento de cada uno de sus componentes, así como en adaptar o personalizar el conjunto de instrucciones para ajustarse a condiciones específicas del sistema. En este proyecto, se propuso el desarrollo de un procesador monociclo MIPS de 32 bits utilizando Logisim Evolution, herramienta que permite simular circuitos digitales. A diferencia de una implementación tradicional, se introdujo una condición especial: modificar las instrucciones tipo R add y and, cambiando sus códigos de operación funct por los dos últimos dígitos de las cédulas de los integrantes del equipo.

2.2. Objetivos

Desarrollar e implementar un procesador monociclo MIPS de 32 bits en Logisim Evolution, incorporando modificaciones en las instrucciones add y and, y validando su funcionamiento mediante la ejecución de un algoritmo diseñado en lenguaje Python y ensamblador.

3. Fundamentos teóricos

1. Arquitectura MIPS:
MIPS (Microprocessor without Interlocked Pipeline Stages) es una arquitectura basada en RISC (Reduced Instruction Set Computer). Está diseñada para ejecutar instrucciones en un número reducido de ciclos de reloj, utilizando un conjunto de instrucciones. En el caso del procesador monociclo, todas las instrucciones se ejecutan en un único ciclo de reloj, lo que simplifica el diseño, aunque puede afectar el rendimiento si se compara con arquitecturas segmentadas.

MIPS utiliza instrucciones de 32 bits divididas principalmente en tres formatos: tipo R (registro), tipo I (inmediato) y tipo J (salto). Las instrucciones tipo R incluyen operaciones aritméticas y lógicas que requieren tres registros y especifican una función adicional mediante el campo funct, lo cual es importante en este proyecto, ya que este campo fue modificado para personalizar instrucciones como add y and.

2. Procesador monociclo:
Un procesador monociclo es un tipo de procesador en el cual cada instrucción se ejecuta completamente en un solo ciclo de reloj. Esto significa que la duración del ciclo debe ser lo suficientemente larga como para cubrir el tiempo de ejecución de la instrucción más compleja. Su diseño es adecuado para fines didácticos, ya que permite comprender de forma clara el flujo de datos y el funcionamiento de cada componente en el procesamiento de instrucciones.

Entre los componentes fundamentales de un procesador monociclo se encuentran:

- Unidad de control: genera las señales de control necesarias para la ejecución de instrucciones.
- Banco de registros: contiene los registros utilizados por el programa.
- ALU (Unidad Aritmético-Lógica): realiza operaciones matemáticas y lógicas.
- Memoria de instrucciones y datos: almacena las instrucciones del programa y los datos correspondientes.
- Contador de programa (PC): indica la dirección de la siguiente instrucción a ejecutar.

3. Logisim Evolution
Es una herramienta de simulación de circuitos digitales que permite diseñar, implementar y probar sistemas digitales, incluyendo procesadores. Su interfaz gráfica facilita la construcción modular de circuitos, lo cual es útil para implementar y probar por separado componentes como multiplexores, registros, ALU, memoria, etc. En este proyecto, Logisim fue utilizado para modelar la microarquitectura del procesador MIPS monociclo y simular su funcionamiento con instrucciones personalizadas.

4. Instrucciones tipo R y el campo funct
Las instrucciones tipo R en MIPS se caracterizan por tener seis campos: opcode, rs, rt, rd, shamt y funct. El campo opcode para

instrucciones tipo R es comúnmente 000000, por lo que la distinción entre instrucciones se realiza mediante el campo funct, que indica la operación específica que debe ejecutar la ALU. En este proyecto, se modificaron los valores del campo funct para las instrucciones add y and, asignando valores personalizados (20 y 36) según los últimos dígitos de las cédulas de los integrantes del grupo. Esto requirió adaptar la unidad de control y la ALU para reconocer y ejecutar correctamente estas instrucciones.

5. Lenguaje ensamblador y simulador MARS

El lenguaje ensamblador MIPS permite la programación directa a nivel de instrucciones de máquina. Utiliza una sintaxis sencilla para manipular registros y realizar operaciones de bajo nivel. El simulador MARS (MIPS Assembler and Runtime Simulator) es una herramienta que permite escribir, ejecutar y depurar programas en ensamblador MIPS. En este proyecto, se empleó MARS para desarrollar y verificar un algoritmo que determina cuántos elementos de un arreglo son iguales al primero, asegurando que solo se usaran instrucciones válidas dentro del conjunto implementado en el procesador personalizado.

4. Metodología

La implementación del procesador monociclo MIPS de 32 bits se llevó a cabo utilizando Logisim Evolution, una herramienta de simulación digital que permite construir circuitos lógicos de manera modular y visual. El diseño se abordó por partes, comenzando por los componentes fundamentales de la arquitectura, que luego fueron integrados en un circuito principal. A continuación, se describe detalladamente la construcción de cada uno de estos módulos, iniciando con el archivo de registros.

5. Diseño del sistema

5.1. RegFile

Archivo de registros (RegFile) El módulo denominado RegFile tiene como propósito principal almacenar y manejar los valores utilizados en las operaciones aritméticas y lógicas dentro del procesador. Este componente está compuesto por 32 registros de 32 bits, los cuales permiten guardar datos intermedios y resultados de las instrucciones ejecutadas. Cada uno de estos registros está identificado por una dirección única que permite su lectura o escritura.

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Figura 1. Registros

El RegFile se encarga de gestionar el acceso a los registros: determina cuál valor debe escribirse, cuál registro será modificado y de cuáles registros se deben leer datos para realizar una operación. Para diseñar este módulo, es necesario comprender la estructura de las instrucciones tipo R, ya que estas definen los registros involucrados en cada operación.

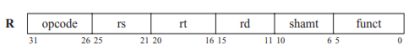

Figura 2. Registros tipo R

En una instrucción tipo R, los registros rs, rt y rd representan respectivamente el primer operando, el segundo operando y el registro donde se almacenará el resultado. Estos campos se conectan en el RegFile como ReadReg1, ReadReg2 y WriteReg. Es decir, ReadReg1 y ReadReg2 se encargan de leer datos desde los registros rs y rt, mientras que WriteReg define en cuál registro (rd) se almacenará el resultado de la operación.

Cada una de estas entradas debe ser de 5 bits, ya que se deben poder direccionar 32 registros distintos. Una vez definidas estas entradas, se procede a diseñar el entorno donde se ubican los 32 registros de 32 bits, cada uno con su respectiva entrada de datos, señal de habilitación (enable) y salida.

Para controlar la escritura en un registro específico, se utiliza un decodificador de 5 bits, el cual activa solo la línea correspondiente al registro que debe ser escrito. Este decodificador recibe dos entradas: una señal de control de 1 bit (RegWrite) que indica si está permitida la escritura, y una señal de 5 bits (WriteReg) que selecciona el registro destino.

La lectura de registros se realiza mediante dos multiplexores de 32 entradas y 5 bits de selección, encargados de seleccionar el valor de salida correspondiente a ReadReg1 y ReadReg2. Estas salidas, denominadas ReadData1 y ReadData2, son los datos que serán utilizados por la ALU u otras partes del procesador durante la ejecución de instrucciones.

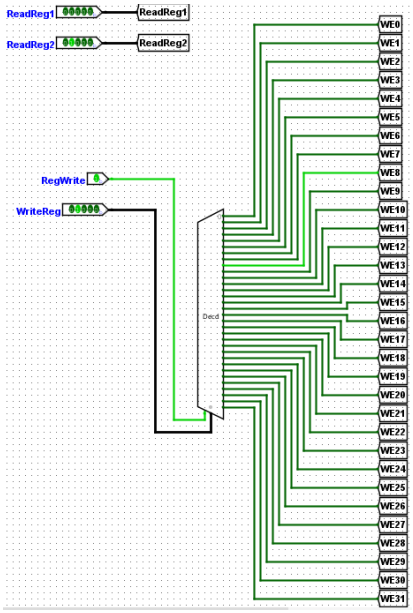

Figura 3. Decodificador

La señal RegWrite, cuando está activa, permite que el decodificador interprete la dirección de WriteReg y habilite la escritura en el registro correspondiente. El dato que se escribirá se recibe a través de la entrada WriteData (de 32 bits), y al completarse la operación, el nuevo valor será visible en la salida del registro (R).

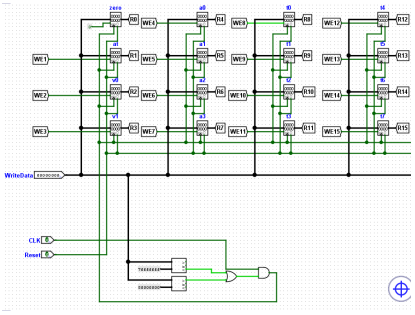

Figura 4. Registros

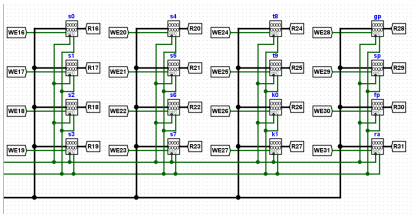

Figura 5. Registros

Por último, la salida de los registros seleccionados por ReadReg1 y ReadReg2 se realiza a través de los multiplexores mencionados anteriormente. Esto asegura que los datos leídos correspondan exactamente a los registros especificados en la instrucción actual. A continuación, se muestra la implementación visual de este módulo en Logisim Evolution.

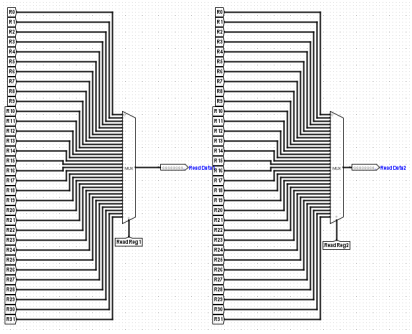

Figura 6. Salida

5.2. Control o Unidad de Control

El circuito descrito anteriormente, el RegFile, contiene una señal de entrada denominada RegWrite, la cual es proporcionada por un componente fundamental del procesador: la unidad de control. Esta unidad tiene la responsabilidad de generar múltiples señales que activan o desactivan operaciones específicas dentro del procesador. Dichas señales son importantes para determinar cómo se ejecuta cada instrucción y hacia qué parte del sistema deben dirigirse los datos.

Las salidas principales generadas por la unidad de control son: RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc y RegWrite. Todas estas señales son de 1 bit, excepto ALUOp, que tiene una longitud de 2 bits, ya que puede representar varias combinaciones necesarias para seleccionar operaciones específicas en la ALU.

Estas señales de control dependen de una entrada de 6 bits llamada Opcode, que representa los seis bits más significativos de la instrucción. Este campo está presente en las instrucciones tipo R, I y J, y es fundamental para que el procesador reconozca el tipo de instrucción que está procesando. Con base en esta entrada, la unidad de control determina qué señales deben activarse para dirigir adecuadamente el flujo de datos y el comportamiento de los componentes internos.

Antes de construir este circuito, es importante definir la función específica de cada señal de control:

- **RegDst:** Determina cuál campo se utilizará como dirección del registro de destino. Si su valor es 1, se selecciona el campo rd (instrucciones tipo R); si es 0, se utiliza el campo rt (instrucciones tipo I).
- **Jump:** Se activa cuando la instrucción corresponde a un salto incondicional (j). Si su valor es 1, el procesador realizará un salto a la dirección especificada.
- **Branch:** Permite la ejecución de instrucciones condicionales como beq o bne. Si esta señal está activa, el procesador evalúa la condición y, si se cumple, realiza un salto relativo.
- **MemRead:** Habilita la lectura de datos desde la memoria. Se activa, por ejemplo, con la instrucción lw, para cargar datos desde una dirección específica.
- **MemtoReg:** Indica el origen del dato que será escrito en un registro. Si su valor es 1, el dato proviene de la memoria; si es 0, el dato proviene de la ALU.
- **ALUOp:** Es una señal de 2 bits que se utiliza para indicar la operación que la ALU debe ejecutar, según la instrucción procesada. Su valor será interpretado por una unidad adicional (ALU Control).
- **MemWrite:** Se activa cuando se desea escribir un dato en la memoria (por ejemplo, en la instrucción sw). Un valor de 1 habilita la escritura en la dirección especificada.
- **ALUSrc:** Define el segundo operando de la ALU. Si su valor es 1, se utilizará un valor inmediato (campo imm de la instrucción tipo I); si es 0, se tomará un valor de un segundo registro (rt).
- **RegWrite:** Habilita la escritura del resultado de una operación (de la ALU o la memoria) en un registro. Un valor de 1 permite el almacenamiento del resultado en el registro de destino.

Estas señales permiten que el procesador actúe de manera diferente según el tipo de instrucción que se esté ejecutando. A continuación, se presenta una tabla que relaciona cada código Opcode con las señales de control correspondientes, de acuerdo con las instrucciones implementadas en esta arquitectura.

Instrucción	Opcode	RegDst	Jump	Branch	MemRead	MentoReg	ALUOp	MemWrite	ALUsrc	RegWrite
Tipo-R	000000	1	0	0	0	0	10	0	0	1
Lw	100011	0	0	0	1	1	00	0	1	1
Sw	101011	X	0	0	0	X	00	1	1	0
Beq	000100	X	0	1	0	X	01	0	0	0
J	000010	X	1	0	0	X	XX	0	X	0
Jal	000011	X	1	0	0	X	XX	0	X	1
Andi	001100	0	0	0	0	0	11	0	1	1

Cabe aclarar que, dentro del conjunto de instrucciones tipo R implementadas en esta arquitectura, se encuentran: add, sub, and, or, nor, slt y jr.

Una vez construida la tabla que relaciona los códigos Opcode con las correspondientes señales de control considerando cada tipo de instrucción, procedemos a su implementación en el simulador Logisim Evolution. Para ello, se traduce dicha tabla en una tabla de verdad que define el comportamiento esperado de la unidad de control ante cada valor posible del campo Opcode.

Las combinaciones de entrada que no corresponden a ninguna instrucción específica se dejarán sin definir, permitiendo que el simulador las complete automáticamente al generar el circuito.

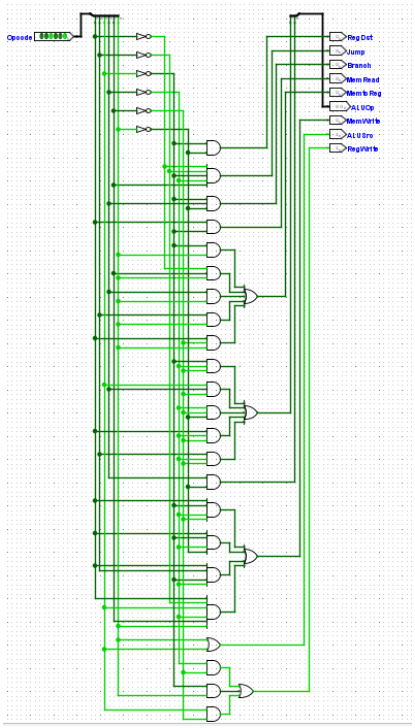

Figura 7. Componente de control

5.3. ALUControl

Una vez se ha identificado el tipo de instrucción y se han generado sus señales de control, es necesario determinar si esta requiere una operación aritmético-lógica. Para ello, se asignaron códigos de 3 bits a seis operaciones ALU: sub (000), add (001), slt (010), and (100), nor (101) y or (110).

Este módulo, conocido como ALU Control, tiene como objetivo principal definir qué operación específica ejecutará la ALU, según los valores de entrada. Además, genera una señal adicional (jr) que se activa únicamente cuando se detecta la instrucción jr.

Las dos entradas que recibe este bloque son: el campo ALUOp, proveniente de la unidad de control, y el campo funct de la instrucción (en caso de ser tipo R). Combinando estos dos valores, el módulo puede determinar si se trata de una instrucción tipo R o tipo I, y asignar el código binario correspondiente a la operación que deberá realizar la ALU.

En este proyecto, se introdujo una modificación específica al campo funct de las instrucciones add y and, reemplazando sus valores originales por los últimos dos dígitos de las cédulas de los integrantes del grupo. Esta modificación se detalla en la siguiente tabla:

intrucción	ALUOp	funct	functionALU	ALU OP
lw	00	xxxxxx	Add	001
sw	00	xxxxxx	Add	001
Beq	01	xxxxxx	Sub	000
Add	10	010100	Add	001
Sub	10	100010	Sub	000
And	10	100100	And	100
Or	10	100101	Or	110
Slt	10	101010	Slt	010
Nor	10	100111	Nor	101
Andi	11	xxxxxx	And	100
Jr	10	001000	NA	NA
Jal	xx	xxxxxx	NA	NA
J	xx	xxxxxx	NA	NA

La tabla de verdad previamente construida resume el comportamiento esperado del módulo ALU Control, basada tanto en los códigos definidos durante el diseño como en la documentación oficial de la arquitectura MIPS de 32 bits. Con esta información, procedimos a implementarla.

Para construir el circuito, fue necesario obtener las expresiones booleanas correspondientes a cada una de las salidas del módulo. Estas ecuaciones se derivaron a partir de los valores de la tabla de verdad, utilizando herramientas en línea que permiten simplificar funciones lógicas mediante mapas de Karnaugh. Gracias a este proceso, se logró optimizar el diseño del circuito, reduciendo la cantidad de compuertas necesarias y facilitando su implementación en Logisim.

Ecuaciones:

$$Op_0 = A_1'A_0' + A_0'F_5'F_4F_3'F_2F_1'F_0' + A_0'F_5F_4'F_3'F_2F_1F_0$$

$$Op_1 = A_1A_0'F_5F_4'F_3'F_2F_1'F_0 + A_1A_0'F_5F_4'F_3F_2'F_1F_0'$$

$$Op_2 = A_1F_5F_4'F_3'F_2F_1' + A_1F_5F_4'F_3'F_2F_0 + A_1A_0$$

A continuación, se muestra el circuito realizado en Logisim.

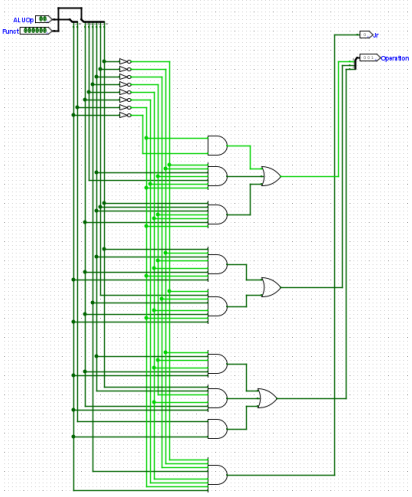

Figura 8. ALU control

5.4. ALU

El circuito anterior nos proporciona el código correspondiente a la operación que debe ejecutar la ALU. A partir de esto, es necesario construir el módulo encargado de realizar las operaciones aritmético-lógicas, utilizando dicho código.

Para ello, se requieren dos datos adicionales: los valores contenidos en los dos registros que participarán en la operación. Cada uno de estos datos es de 32 bits, ya que provienen directamente del archivo de registros (ReadData1 y ReadData2).

Como salidas, este bloque generará también dos señales: una correspondiente al resultado de la operación, y otra llamada Zero, que indica si el resultado obtenido es igual a cero. Esta última es particularmente útil para instrucciones condicionales como las de tipo Branch, donde se evalúa si dos registros contienen valores iguales o diferentes.

Figura 9. ALU

El diseño de este bloque se basa en la librería de componentes de Logisim Evolution. Allí se implementan las seis operaciones definidas previamente (add, sub, and, or, nor, slt), y mediante un multiplexor controlado por la señal ALUOperation (proveniente del módulo ALU Control) se selecciona cuál de estas operaciones se ejecutará. El resultado final se direcciona hacia el resto del procesador para continuar con la ejecución de la instrucción correspondiente.

5.5. BreakDown

Tambien es necesario preparar las entradas que le daran los datos a los bloques previamente diseñados. Para ello, se debe descomponer la instrucción actual en sus distintos campos, según su formato. Esta descomposición nos permite extraer valores como: Opcode, rs, rt, rd, shamt (no utilizado en esta arquitectura), funct, immediate y address.

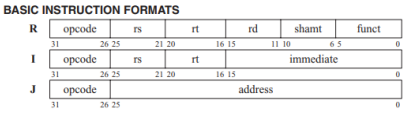

Figura 10. Formato de las instrucciones basicas

En la imagen se señalan claramente los rangos de bits correspondientes a cada campo. Para lograr esta separación en Logisim Evolution, se hace uso exclusivo de separadores (splitters), que permiten extraer subconjuntos de bits desde la instrucción completa (de 32 bits).

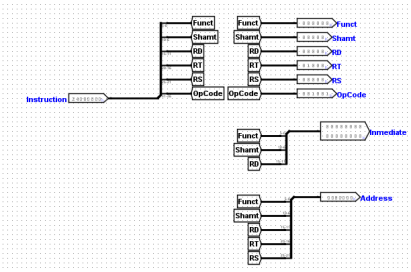

Figura 11. Estructura BreakDown

Una vez obtenidos estos campos, podrán conectarse a los distintos módulos del procesador, como la unidad de control, el RegFile y la ALU. Sin embargo, la conexión final de estas señales se realizará más adelante, ya que el flujo de datos pasa primero por ciertos componentes lógicos como multiplexores, comparadores, desplazadores, entre otros, que influirán en las decisiones de control. Estos elementos se abordarán con mayor detalle en las siguientes secciones.

5.6. IFetch (Instruction Fetch)

Entramos ahora a uno de los bloques esenciales del procesador: el encargado de obtener y gestionar las instrucciones. En esta etapa, no solo se recupera la instrucción actual desde memoria, sino que también se calcula la dirección de la siguiente instrucción, a través del registro PC y su incremento por 4 (PC + 4), como es característico en la arquitectura MIPS.

Para almacenar las instrucciones del programa, se requiere una memoria adecuada. En este caso, la opción más eficiente es utilizar una memoria ROM, ya que esta permite únicamente la lectura, y en nuestro diseño, las instrucciones no serán modificadas durante la ejecución. La ROM se ajusta perfectamente a esta necesidad, ya que contiene el conjunto de instrucciones que el procesador debe ejecutar, sin permitir alteraciones en tiempo de ejecución.

Respecto a las señales PC y PC-Next (la dirección de la próxima instrucción), se tuvo en cuenta una restricción del entorno de simulación: por ello, se usaron 24 bits del valor de PC-Next (bits 2 a 25) para conectarlos a la entrada de la ROM. De este modo, se garantiza un direccionamiento funcional sin interferir con los límites de la herramienta.

Para mantener la sincronización sin interrumpir la instrucción que se está ejecutando, se utilizó un registro intermedio, que almacena la dirección de la próxima instrucción. Además, se implementó un sumador que incrementa en 4 el valor de PC, lo que asegura la lectura secuencial del programa. Finalmente, se conectó una salida denominada Instruction, la cual extrae el valor desde la ROM y lo entrega al resto del procesador para su decodificación y ejecución.

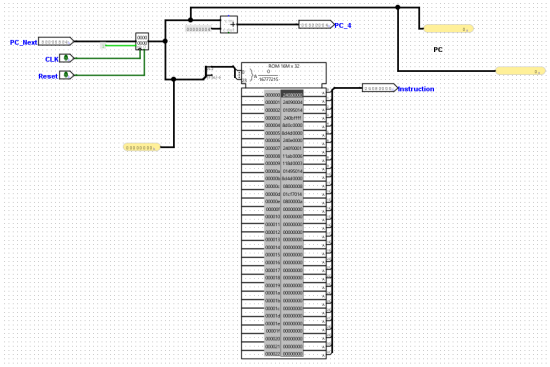

Figura 12. Estructura del IFetch

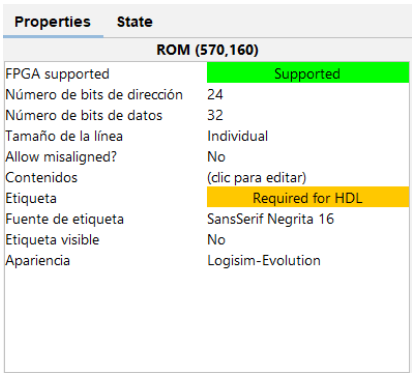

Figura 13. Propiedades de la ROM

Al configurar la memoria ROM en Logisim Evolution para el módulo IFetch, se deben establecer correctamente sus propiedades para que se adapte al tamaño y tipo de instrucciones utilizadas en la arquitectura MIPS de 32 bits. Las propiedades configuradas fueron las siguientes:

Número de bits de dirección: 24

Este valor define cuántas direcciones distintas puede manejar la ROM, es decir, cuántas instrucciones puede almacenar. Con 24 bits, se pueden direccionar hasta $2^{24} = 16,777,216$ ubicaciones posibles. Aunque no se utilicen todas, este tamaño fue elegido principalmente para coincidir con el valor de PC-Next, del cual se utilizan los bits 2 al 25.

Número de bits de datos: 32
Cada dirección de memoria almacena una palabra de 32 bits, lo cual corresponde exactamente al tamaño de cada instrucción en la arquitectura MIPS.

5.7. DataMemory

Después de haber configurado la memoria de instrucciones, es necesario implementar el bloque encargado de almacenar y gestionar los datos utilizados por el programa. Para ello, se utiliza una memoria RAM, ya que, a diferencia de la ROM, esta permite tanto lectura como escritura, lo cual es fundamental para manejar el segmento data en la ejecución de instrucciones como lw (load word) y sw (store word).

Los datos de entrada necesarios para esta memoria son similares a los de la ROM:

- Una dirección que indica la posición de memoria a acceder.
- El dato a escribir (en caso de escritura).
- El dato para leer (en caso de lectura).
- También es necesario conectar el reloj (clock) del sistema para sincronizar las operaciones y una señal de reset que permite reiniciar el estado de la memoria si es necesario.

La salida principal de esta memoria será el valor de 32 bits correspondiente al contenido de la dirección apuntada, representado como ReadData.

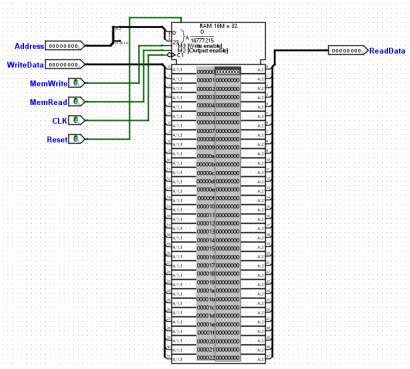

Figura 14. Estructura de DataMemory

Properties	State
RAM (370,120)	
FPGA supported	Not supported
Número de bits de dirección	24
Número de bits de datos	32
Habilita:	El uso del byte permite
Tipo de carnero	volátil
Usar el pasador transparente	Sí
Flanco	Flanco de bajada
Lectura asincrónica:	Sí
Control de lectura y escritura	Lectura/escritura de palabras c...
Implementación del bus de da...	Buses de datos separados par...
Etiqueta	Required for HDL
Fuente de etiqueta	SansSerif Negrita 16
Etiqueta visible	No
Apariencia	Logisim-Evolution

Figura 15. Propiedades de la RAM

Para el correcto funcionamiento de la memoria de datos dentro del procesador monociclo, se realizaron los siguientes ajustes en las propiedades de la RAM en Logisim Evolution:

Número de bits de dirección: 24
Este valor permite direccionar hasta 2^{24} posiciones de memoria, garantizando un espacio suficiente para pruebas en el simulador.

Número de bits de datos: 32
Se configura en 32 bits para que coincida con el tamaño de las instrucciones MIPS y el ancho de palabra del procesador..

Usar pasador transparente: Sí
Esta opción permite que el contenido de la celda de memoria seleccionada se refleje inmediatamente en la salida cuando la lectura está habilitada. Esto simplifica la conexión con otros componentes y evita la necesidad de controladores adicionales.

Flanco: Flanco de bajada
Se configura la escritura para que ocurra en el flanco descendente del reloj, asegurando una sincronización precisa con el resto del sistema. Esto reduce la posibilidad de conflictos durante la escritura en memoria y mejora la estabilidad del circuito.

Lectura asíncrona: Sí
Activar la lectura asincrónica permite que, al cambiar la dirección de acceso, el dato correspondiente se refleje instantáneamente en la salida sin necesidad de esperar al siguiente pulso de reloj. Esto es útil para operaciones como lw, donde se requiere respuesta inmediata de la memoria.

6. Integración de los componentes del procesador monociclo

Al conectar los distintos módulos del procesador monociclo, algunas uniones requieren una lógica más complejas, mientras que otras son más directas. A continuación, se describen las principales conexiones necesarias para garantizar el funcionamiento correcto y coherente del procesador.

6.1. Registro de escritura (WriteRead) entre Rd y Rt

En las instrucciones tipo R e I, el registro donde se escribe el resultado puede variar. Por eso, no se conecta directamente rd o rt al bloque RegFile, sino que se utiliza un multiplexor que decide cuál de los dos se usará como destino (WriteReg).

Este multiplexor es controlado por las señales:

RegDst: Si está activa (valor 1), se selecciona rd (para instrucciones tipo R).

Jump (JAL): Si está activa, se fuerza la selección del registro 31 (0x1F, en binario 11111) como destino, para guardar la dirección de retorno.

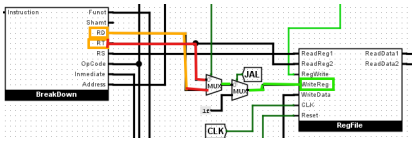

Figura 16. Multiplexor de WriteReg

6.2. Cálculo de PC-Next y control de salto

Debido a ciertas limitaciones en Logisim Evolution, para calcular la dirección siguiente (PC-Next) en caso de que se ejecute una instrucción de tipo Branch, es necesario realizar una suma entre PC + 4 y el valor Inmediato (de la instrucción), el cual debe ser previamente desplazado dos bits a la izquierda. Esta operación se realiza a través de un sumador adicional, ya que no puede hacerse directamente en la unidad IFetch.

En caso de que no se esté ejecutando un Branch, simplemente se tomará el valor PC + 4 calculado previamente en la etapa IFetch como la próxima instrucción a ejecutar.

Cuando se ejecuta una instrucción de tipo Jump, se utiliza otro camino alternativo. En este caso, se debe calcular la dirección absoluta de salto usando el campo address de la instrucción. Para esto:

- Se desplazan dos bits a la izquierda los 26 bits del campo address.
- Se extienden a 28 bits.
- Finalmente, se concatenan con los 4 bits más significativos de PC + 4 para formar una dirección completa de 32 bits.

Si el bloque de Breakdown (separador de campos) ha sido configurado correctamente, esta lógica asegurará que se apunte a la dirección de salto deseada.

Una vez calculadas estas posibles direcciones (Branch, Jump o PC + 4), se utilizarán multiplexores para seleccionar cuál será efectivamente el valor de PC-Next. Finalmente, si se está ejecutando la instrucción jr, el último multiplexor seleccionará el contenido de ra (ReadData1) como nueva dirección de salto. En caso contrario, se tomará la dirección seleccionada previamente.

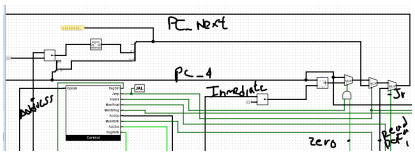

Figura 17. Cálculo PC

7. Código a realizar para la simulación

El problema asignado consistía en desarrollar un algoritmo que contara cuántos elementos de una lista son iguales al primer valor de dicha lista. En otras palabras, el objetivo era recorrer una lista de datos y determinar cuántas veces se repite el primer elemento dentro del resto de la lista.

7.1. Codigo en Python

Aunque la práctica solicitaba la implementación en pseudo-código, decidimos inicialmente desarrollarlo en Python, ya que esto nos permitió probar más fácilmente los resultados del algoritmo realizado en MIPS.

Cabe aclarar que, durante el conteo, no consideramos el primer valor como parte del conteo, es decir, solo empezamos a contar las apariciones del mismo valor a partir del segundo elemento en adelante. De este modo, el resultado refleja cuántas veces se repite el primer dato, sin contarse a sí mismo.

```
    vector = [3,1,3,4,3,3,3,3,-1]

    t0 = 0              # Posicion inicial del vector
    t1 = 1              # Para subir las posciones del
     ↪ vector
    t2 = t0 + t1        # Segunda posicion del vector
    t3 = -1             # Valor de parada
    t4 = vector[t0]     # Primer valor del vector (Valor
     ↪  para comparar)
    t5 = vector[t2]     # Valor a comprar
    t6 = 0              # Contador
    t7 = 1              # Incrementar contador

    while t5 != t3:
        if t4 == t5:
            t6 += 1
        t2 += t1
        t5 = vector[t2]

    print(f"El numero de veces que se repitio el primer
     ↪ valor es de: {t6}")
```

```

     ↪ ------------------------------------------------------
     ↪
           El numero de veces que se repitio el primer
     ↪ valor es de: 5

     ↪ ------------------------------------------------------
     ↪

```

7.2. Codigo en MIPS

El algoritmo propuesto fue implementado inicialmente en ensamblador MIPS utilizando la herramienta MARS, con el objetivo de contar cuántos elementos de un arreglo son iguales al primer valor, excluyendo el mismo del conteo. La estructura del código hace uso de instrucciones básicas como lw, sw, beq, bne, add y j, todas compatibles con el conjunto mínimo de instrucciones requerido por un procesador monociclo MIPS.

Limitaciones del procesador implementado

Nuestro procesador, diseñado en Logisim Evolution, no cuenta con todas las funciones estándar del conjunto MIPS, por lo que fue necesario adaptar el código original. Por ejemplo:

No se implementaron pseudoinstrucciones como la (load address), ni funciones de sistema como syscall.

La memoria no inicia en una etiqueta simbólica (vector:), sino en una posición fija de memoria, como la dirección 0.

Vector = [3,1,3,3,2,3,3,-1]

```
        .data
    vector: .word 3, 1, 3, 3, 2, 3, 3, -1   # El -1
     ↪ indica fin del vector
    resultado: .word 0

    .text
    main:
        la $t0, vector        # $t0 = puntero al vector
        lw $t1, 0($t0)        # $t1 = primer elemento
     ↪ del vector (comparador)
        addi $t2, $zero, 0    # $t2 = contador de
     ↪ elementos iguales

    recorrer:
        lw $t3, 0($t0)        # $t3 = vector[i]

        addi $t4, $zero, -1   # Valor centinela
        beq $t3, $t4, fin     # Si vector[i] == -1,
     ↪ termina

        bne $t3, $t1, siguiente
        addi $t2, $t2, 1      # Si son iguales,
     ↪ incrementar contador

    siguiente:
        addi $t0, $t0, 4      # Mover al siguiente
     ↪ elemento
        j recorrer

```

```
24      fin:
25          la $t5, resultado
26          sw $t2, 0($t5)        # Guardar el resultado en
        ↪ memoria
27
```

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	268501020
$t1	9	3
$t2	10	5
$t3	11	-1
$t4	12	-1
$t5	13	268501024
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194360
hi		0
lo		0

argument 2

Figura 18. Resultados

En el registro t2 se guardo el contador.

7.3. Versión adaptada para el procesador en Logisim

En la versión adaptada, cada salto o carga se hace utilizando valores numéricos en lugar de etiquetas simbólicas. Por ejemplo, en lugar de hacer un salto a fin, se usaría la dirección exacta en la que se encuentra esa instrucción dentro del flujo de ejecución. Esto se debe a que Logisim no interpreta etiquetas ni realiza ensamblado automático, por lo cual toda la lógica debe manejarse directamente con direcciones de memoria.

Este enfoque nos permitió adaptar el programa con éxito y verificar su correcto funcionamiento dentro del entorno simulado, cumpliendo así con los objetivos de la práctica.

```
1       .text
2   main:
3         la $t0, 0          # Posicion inicial del
    ↪ vector ($t0 = 0)
4         la $t1, 4   # Para subir las posciones del
    ↪ vector ($t1  = 4)
5         add $t2, $t0, $t1     # Segunda posicion del
    ↪ vector ($t2 = $t0 + $t1)
6         la $t3, -1
7         lw $t4, 0($t0)
8         lw $t5, 0($t2)
9         la $t6, 0
10        la $t7, 1
11
12  loop:
13        beq $t5, $t3, fin
14        beq $t4, $t5, contador
15
16  incrementar:
17    add $t2, $t2, $t1
18        lw $t5, 0($t2)
19        j loop
20
21  contador:
22    add $t6, $t6, $t7
23    j incrementar
24
25  fin:
26
27
```

8. Simulación

Para llevar a cabo la simulación del algoritmo en el procesador monociclo implementado en Logisim Evolution, se utilizaron dos archivos externos que contienen los datos necesarios para su ejecución:

Datos-memoria: contiene los valores del arreglo y demás datos, representados en formato hexadecimal, y es cargado directamente en la RAM del procesador.

Instrucciones: contiene el programa ensamblador traducido a código máquina hexadecimal, adaptado específicamente a la arquitectura del procesador construido, incluyendo los cambios realizados a las instrucciones add y and, cuyos códigos funct fueron modificados de acuerdo con las condiciones de la práctica.

Una vez cargados ambos archivos en sus respectivas memorias (ROM para instrucciones y RAM para datos), la simulación se ejecuta permitiendo que el reloj (clk) avance libremente. Esto activa el flujo de datos entre los bloques del procesador, permitiendo la ejecución secuencial de las instrucciones.

Durante este proceso, el valor de los registros se va actualizando según el comportamiento del programa, permitiendo verificar que cada instrucción se ejecuta correctamente conforme al diseño del procesador.

Esta metodología asegura una simulación fiel al comportamiento esperado de un procesador monociclo MIPS, respetando las adaptaciones realizadas en la arquitectura y las limitaciones impuestas por el entorno de diseño.

9. Ejercución y resultados

Una vez cargados correctamente los archivos Datos-memoria e Instrucciones en la RAM y ROM del procesador monociclo, respectivamente, se procedió a la ejecución del programa haciendo avanzar el reloj (clk). Esto permitió activar la secuencia de instrucciones y la interacción entre los distintos módulos del procesador: unidad de control, registros, ALU, memoria, y demás componentes diseñados.

Durante la ejecución, fue posible verificar el correcto funcionamiento del algoritmo diseñado. La secuencia de instrucciones recorrió adecuadamente el arreglo cargado en memoria, comparando cada valor con el primero y realizando el conteo según la lógica establecida. La ALU operó según los códigos modificados para las instrucciones add y and, sin generar conflictos ni errores de interpretación.

Los resultados del conteo se reflejaron directamente en los registros del procesador, específicamente en el registro utilizado como acumulador del contador, el cual fue monitoreado para validar la ejecución completa y correcta del algoritmo.

Gracias a la correcta configuración del circuito, el flujo de instrucciones y datos se desarrolló de forma continua, demostrando que tanto el diseño del hardware como la adaptación del código ensamblador

fueron adecuados para cumplir con el objetivo de la práctica.

En este ejemplo usamos una lista de datos diferentes.

Lista = [00000003 00000001 00000003 00000004 00000003 00000003 00000003 ffffff000 00000003 00000003 00000003 00000003 00000003 00000004 00000007 00000008 00000000 00000008 00000007 00000004 00000000 00000003 00000004 00000057 00000056 00000003 ffffffff]

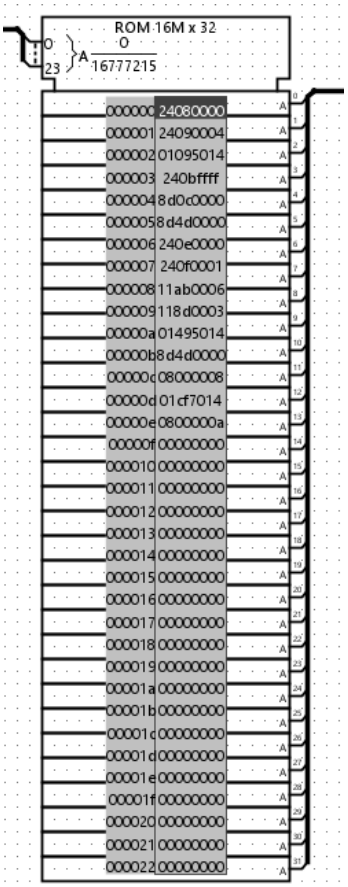

Figura 20. Memoria de instrucciones cargada

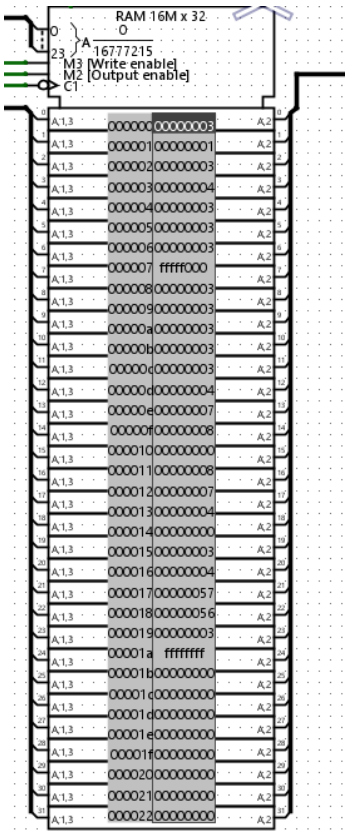

Figura 19. Memoria de datos cargada

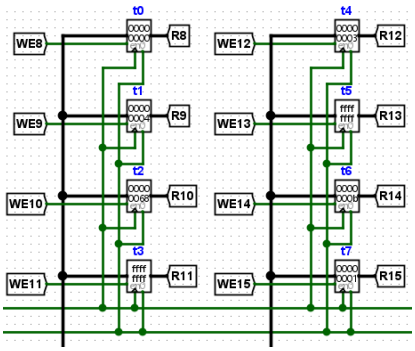

Figura 21. Resultados

Según lo evidenciado en la imagen de resultados obtenida al finalizar la simulación, se comprueba que el procesador:

- Inicializó correctamente los registros implicados.
- Realizó adecuadamente el recorrido del arreglo en memoria.
- Ejecutó correctamente el conteo de elementos iguales al primero.
- Almacenó el resultado final de manera correcta en el registro asignado.

Este comportamiento validó tanto la funcionalidad del procesador como la adaptación del código ensamblador al conjunto de instrucciones personalizadas, cumpliendo así con los objetivos técnicos de la práctica.

Link del repositorio: Repositorio de la practica

10. Bibliografias

32, M. (s.f.). MIPS Reference Data

Buitrago, J. B. (2025). Documentos compartidos en el curso Arquitectura de Computadores y Laboratorio. Medellin.