Laboratorio No. 3

Diseño de Procesador Monociclo

Procesador Monociclo MIPS 32

Autor: Santiago Correa Marulanda

C.C: 1033486496

Arquitectura de Computadores y Laboratorio

Fredy Alexander Rivera Velez

22/09/2024

Contenido

INTRODUCCIÓN 3

OBJETIVO 3

DESCRIPCIÓN 3

- RegFile 3
- Control o Unidad de Control 8
- ALUControl 11
- ALU 13
- BreakDown 14
- IFetch 15
  - Configuración de la ROM 16
- DataMemory 17
  - Configuración de la memoria RAM 18
  - Conexiones del circuito 18
- RD y RT con WriteReg 18
- Cálculo de PC_Next 20
- Código del problema 21

Simulación 26

- Segmento .data 26
- Segmento .text 27
- Memoria RAM 27
- Memoria ROM 28

Ejecución 29

Detalles de simulación 29

Archivos de simulación 30

Conclusión 30

Referencias 31

Bibliografía 31

INTRODUCCIÓN

Este informe presenta las decisiones de diseño y los procesos empleados en el desarrollo de un procesador monociclo MIPS 32. A partir de los conocimientos adquiridos en el diseño de sistemas digitales, se detallan los pasos necesarios para implementar la arquitectura como circuito. Además, se analiza el uso del lenguaje de programación ensamblador MIPS 32 para la ejecución de un programa que utiliza únicamente las instrucciones implementadas en dicha arquitectura, garantizando su funcionamiento exitoso y permitiendo evidenciar la eficiencia del procesador.

OBJETIVO

El objetivo de este laboratorio es aplicar de manera práctica los conocimientos adquiridos a lo largo del curso, abarcando tanto los temas relacionados con los sistemas digitales y la programación en bajo nivel, como el dominio de las matemáticas discretas, la lógica de programación, y el manejo de simuladores y herramientas asociadas. El propósito es integrar todos estos aprendizajes para abordar la solución de un problema real, en el que se busca la correcta implementación de una arquitectura de procesador monociclo MIPS 32, asegurando su funcionamiento adecuado y optimizado. A través de esta experiencia, se pretende consolidar el entendimiento de los procesos de diseño y ejecución en sistemas digitales, al tiempo que se fomente el desarrollo de habilidades prácticas en la resolución de desafíos reales y complejos.

DESCRIPCIÓN

RegFile

El circuito RegFile tiene como función principal almacenar y gestionar los valores utilizados en las operaciones aritméticas y lógicas dentro del procesador.

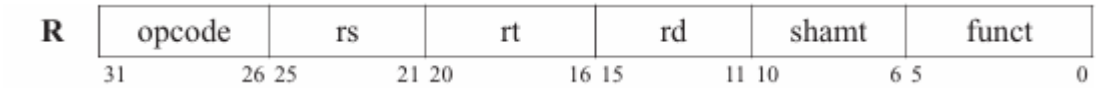
Este archivo de registros se compone de 32 registros de 32 bits, que se emplean para almacenar y operar con los valores que contienen. La enumeración de estos registros seguirá este diseño:

REGISTER NAME, NUMBER,

NAME	NUMBER
$zero	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$k0-$k1	26-27
$gp	28
$sp	29
$fp	30
$ra	31

(32)

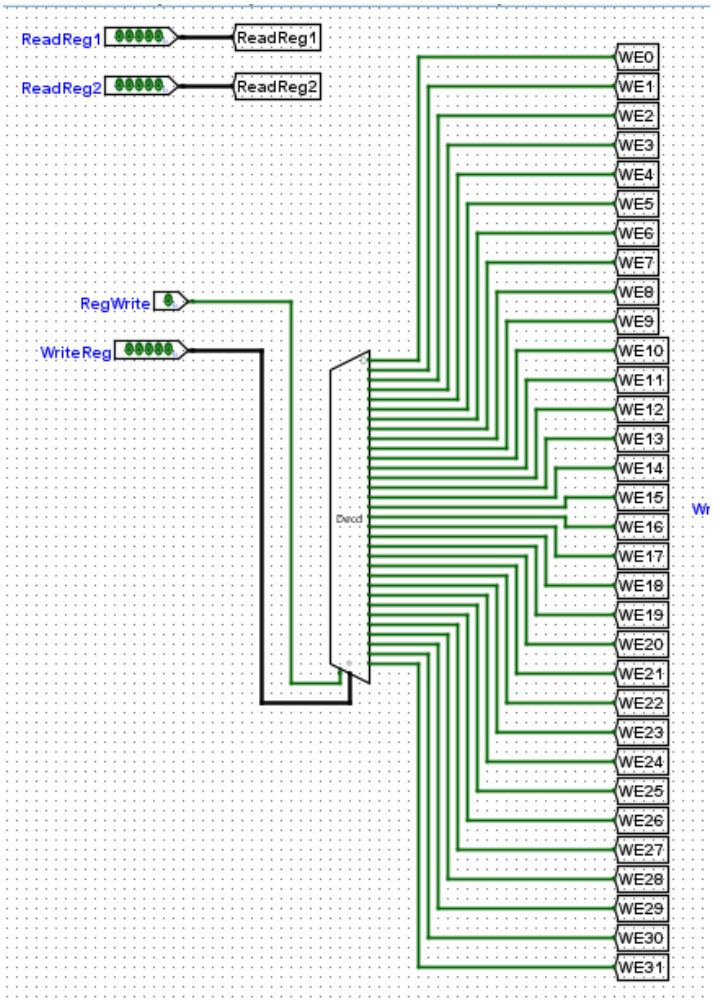
En esencia, el RegFile se encarga de gestionar la información relacionada con los registros: indica qué valor debe escribirse, cuál es el registro o dirección que se debe sobrescribir, y qué registros serán operados durante el proceso. Para su funcionamiento, debemos basarnos en los formatos de instrucciones



En este caso, el RegFile recibe datos de tres registros: rs (registro base), rt (registro fuente) y rd (registro de destino), que se asignan a ReadReg1, ReadReg2 y WriteReg, respectivamente. Esto se debe a que rs y rt son los registros de los cuales se leerán los valores, mientras que rd es el registro en el que se escribirá el resultado de la operación que se realice con los dos primeros registros. Para las instrucciones de tipo I y J, se seguirá una lógica específica para el manejo de estos registros, la cual será detallada más adelante.

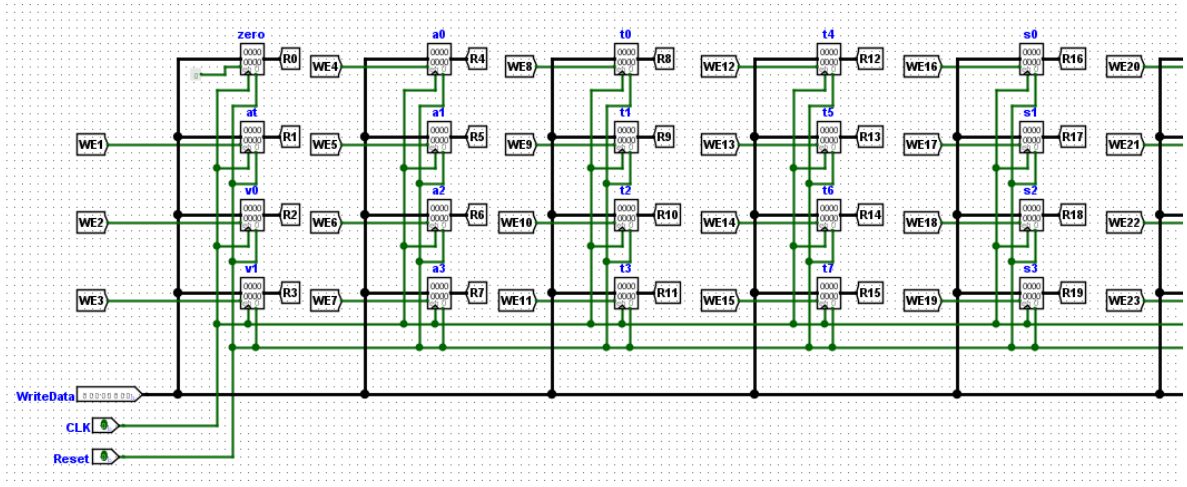
Como se indica en el gráfico, estos 3 deben ser una entrada de 5 bits, Luego de esto, debemos preparar el entorno de los 32 registros de 32 bits, cada registro recibirá un valor que será el que se escribirá en el indicado, además una entrada de activación, así aseguramos que únicamente se modifique la información del registro deseado, además, conectaremos su respectiva salida. Para implementar este bit de activación, debemos implementar un decodificador con 5 bits de selección, donde la entrada de enable será un

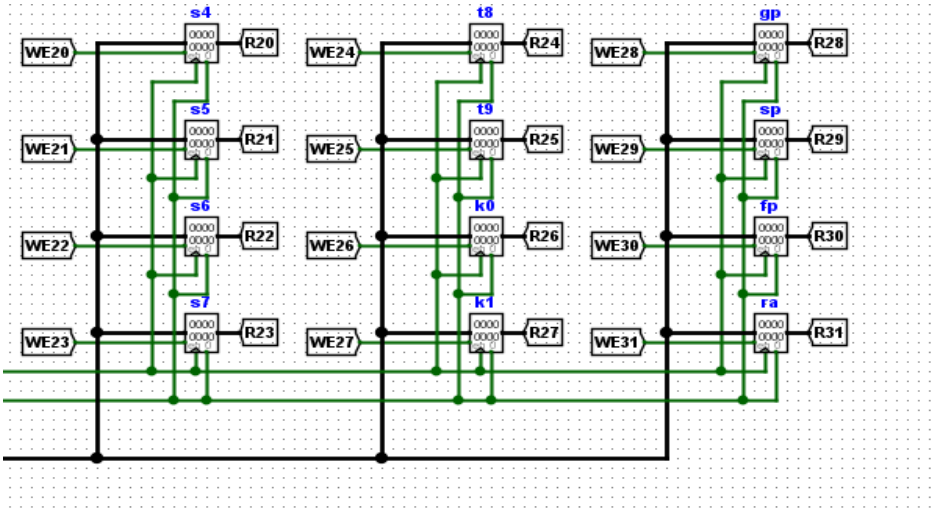
dato de 1 bit que indicará si la instrucción que está siendo ejecutada, permite la escritura sobre un registro o no, en la entrada select, recibirá un dato de 5 bits, que indicará el registro que será modificado, así permitiendo la escritura del respectivo registro. Ahora como se mencionó al inicio, ReadReg1 y ReadReg2 se hará cargo de obtener el valor almacenado en los registros que estos 2 indiquen, con ayuda de 2 multiplexores de 32 bits y 5 bits de selección, donde los 32 registros estarán conectados a su respectiva entrada en el multiplexor, la entrada select tendrá ReadReg1 o 2 para permitir el pasó del registro a leer y a cada multiplexor conectar la salida ReadData1 y ReadData2 que tendrá el valor de cada uno.



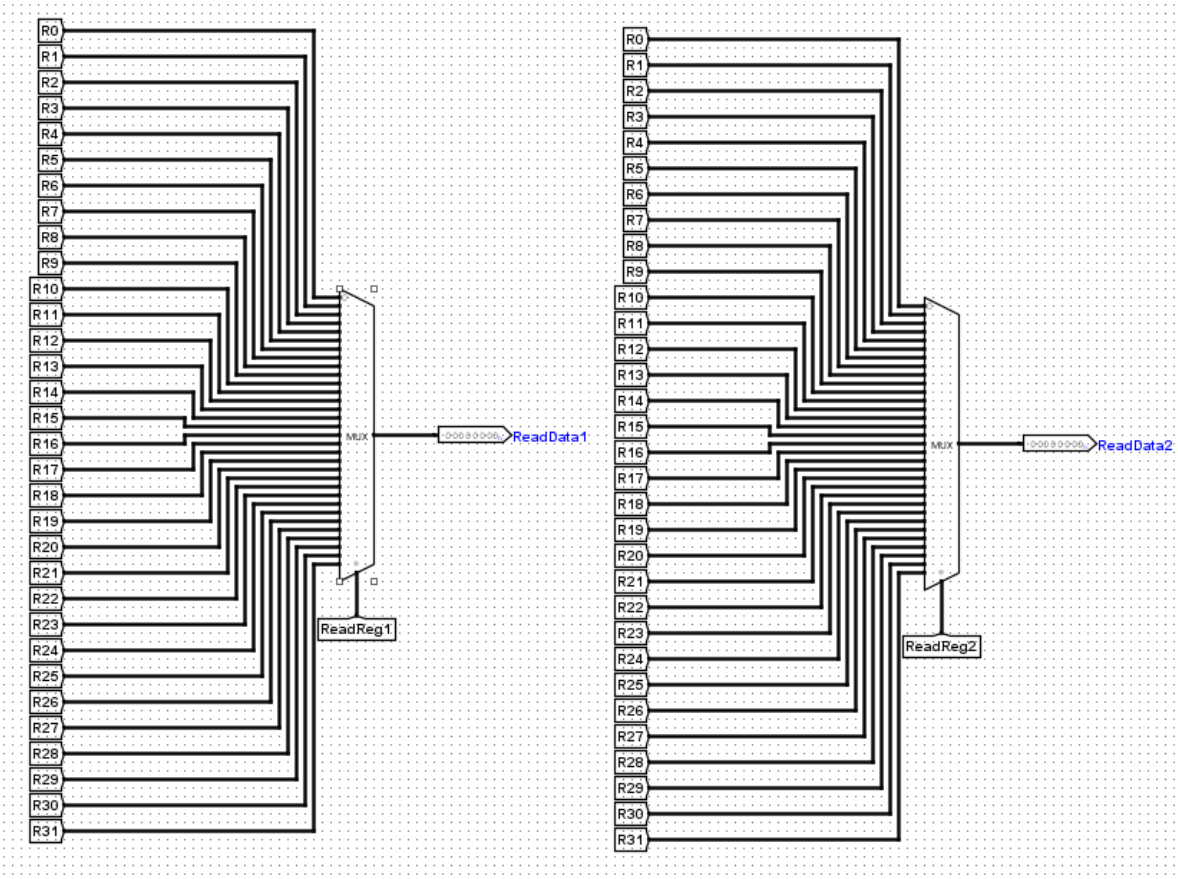
Aquí se observa como dependiendo de la entrada RegWrite (que será interpretada como una señal de activación para sobre escribir un registro o no) permitirá la decodificación de la

señar WriteReg, que indica el registro a sobre escribir, dependiendo del valor que haya aquí, activará la respectiva señal del codificador.

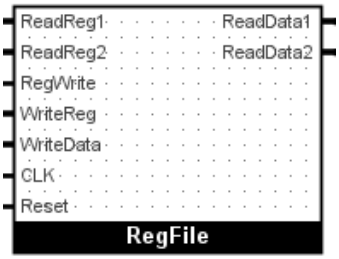




Aquí vemos la conexión entre las salidas del decodificador y los registros, además de entrada WriteData de 32 bits, está entrada es el valor a escribir en el registro, dependiendo de la salida WE del decodificador, activará la escritura del registro indicado y recibirá el valor que hay en WriteData, luego este se vera en la salida R.



Aquí observamos finalmente la salida de los datos ReadData1 y ReadData2, estos multiplexores permitirán la salida indicada en ReadReg1 y ReadReg2 (recordar que estos indican el numero de registro a leer) la salida del Mux será el valor almacenado en el registro indicado. A continuación, se muestra el circuito a la hora de implementarlo en un "main". (Velez, 2024)



Control o Unidad de Control

El circuito anteriormente diseñado, tiene un dato de entrada que hemos llamado RegWrite, este es uno de los datos de salida que tendrá la unidad de control, que se encarga de habilitar diferentes salidas que hacen referencia a señales que podemos interpretar cómo señales de activación, que dependiendo del valor que haya en estas, el procesador realizará las operaciones correspondientes. Estas salidas son: RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, y RegWrite, cada una de estas, a excepción de ALUOp, serán de 1 bit, esta ultima debe ser de 2 bits.

Los valores que los datos de salida reciben dependerán de una entrada de 6 bits que llamaremos Opcode, que hace referencia a los 6 bits mas significativos de cada instrucción, que, si recordamos el formato de instrucción R, I y J, está presente en cada una de estas, en consecuencia a la lógica que se desarrolla en este circuito, podremos determinar (por medio de su Opcode) el formato de instrucción y que señales de activación enviará a otros circuitos.

Para diseñar esta pieza, es necesario definir cada señal de control y su función en el procesador:

RegDst: Controla el registro de destino. Cuando su valor es 1, indica que el registro de destino es rd (usado en instrucciones de tipo R). Cuando su valor es 0, el registro de destino es rt (usado en instrucciones de tipo I).

Jump: Gestiona si la instrucción en ejecución es de tipo J. Si lo es, se activa con un valor de 1; de lo contrario, su valor es 0.

Branch: Se activa cuando la instrucción es de tipo de salto condicional, como bne, permitiendo que el procesador realice un salto si se cumple la condición.

MemRead: Controla si la instrucción requiere leer datos de la memoria (por ejemplo, en lw). Cuando esta señal está activada, el procesador lee un dato de la memoria y lo almacena en el registro de destino.

MemtoReg: Determina la fuente de los datos que se escribirán en el registro de destino. Si su valor es 1, el dato proviene de la memoria, si es 0, el dato proviene de la salida de la ALU.

ALUOp: Define la operación que la ALU debe realizar. Este valor se ajusta según la instrucción en ejecución y permite seleccionar la operación adecuada en la ALU.

MemWrite: Controla la escritura en memoria. Si su valor es 1, la instrucción activa la escritura en memoria; si es 0, no se realiza ninguna operación de escritura.

ALUSrc: Indica si la ALU debe usar un valor inmediato (constante) o un valor de otro registro como segundo operando. Un valor de 1 selecciona un valor inmediato, mientras que un valor de 0 selecciona un valor de registro.

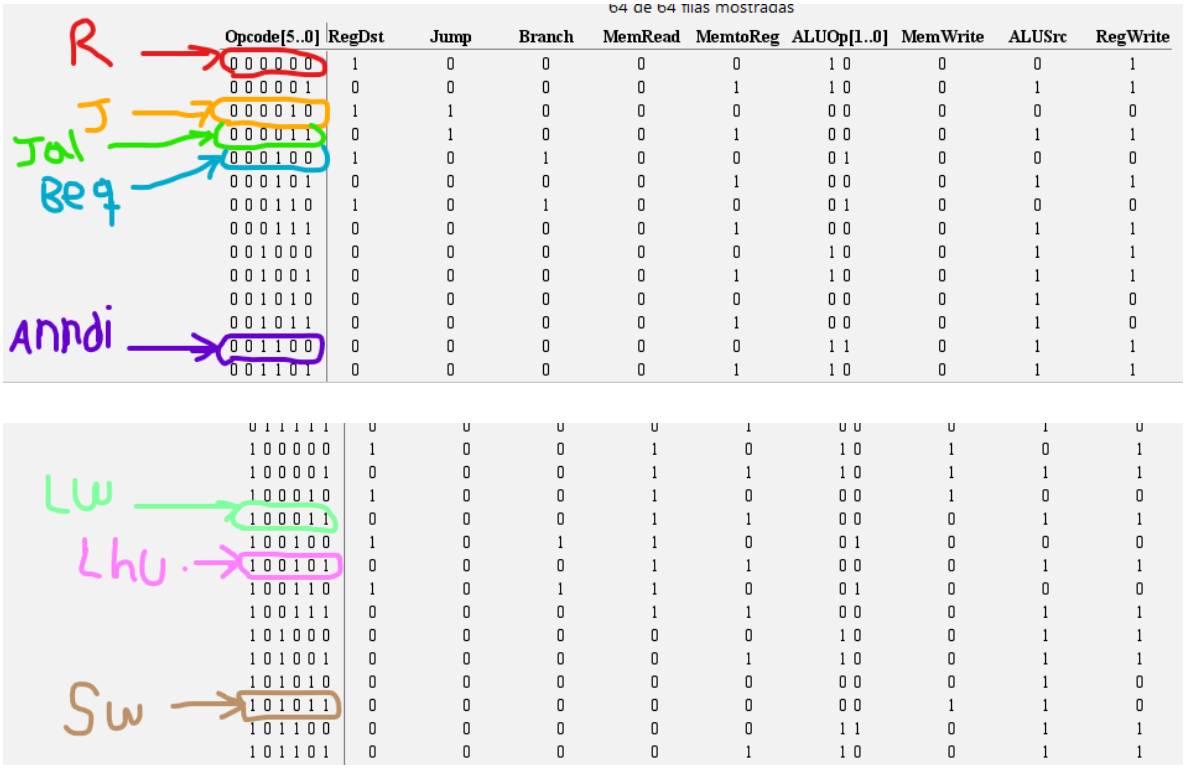
RegWrite: Controla la escritura en el registro de destino. Se activa (con un valor de 1) cuando el resultado de una operación (de la ALU o de la memoria) debe almacenarse en un registro de destino.

Seguido de esto, vamos a plasmar los Opcode de las diferentes instrucciones implementadas en esta arquitectura junto con sus salidas,

Instr	Opcode	RegDst	Jump	Branch	MemRead	MemtoReg	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
Lhu	100101	0	0	0	1	1	00	0	1	1

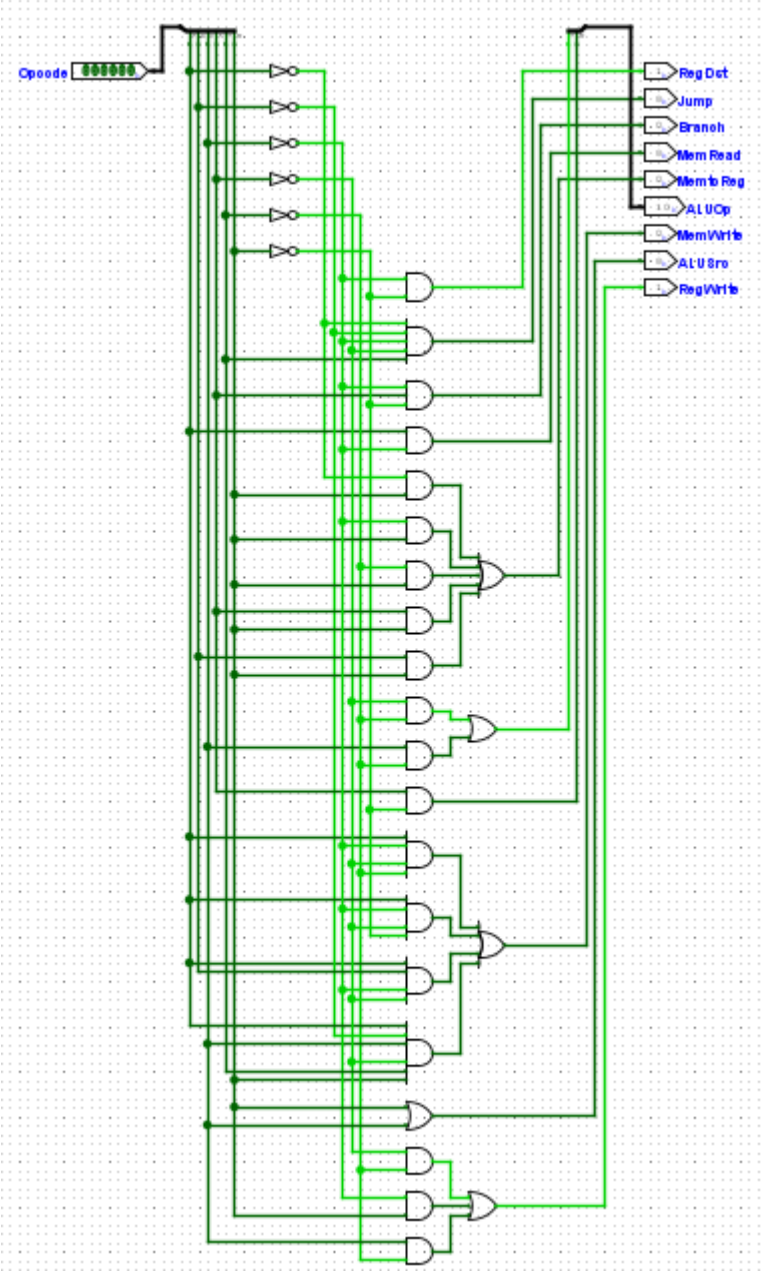
Aclarar que las instrucciones tipo R incluyen las instrucciones add, sub, and, or, nor slt y jr, sumando las otras 7, da un total de 14 instrucciones implementadas en la arquitectura.

Una vez realizada esta tabla de verdad, podemos implementarla en el simulador Logisim Evolution para crear el circuito. (Únicamente llenaremos los datos que hemos recopilado en la tabla anterior, el resto permitiremos que se complete de manera automática con optimización de minterms).



Opcode[5..0]	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp[1..0]	MemWrite	ALUSrc	RegWrite
000000	1	0	0	0	0	10	0	0	1
000001	0	0	0	0	1	10	0	1	1
000010	1	1	0	0	0	00	0	0	0
000011	0	1	0	0	1	00	0	1	1
000100	1	0	1	0	0	01	0	0	0
000101	0	0	0	0	1	00	0	1	1
000110	1	0	1	0	0	01	0	0	0
000111	0	0	0	0	1	00	0	1	1
001000	0	0	0	0	0	10	0	1	1
001001	0	0	0	0	1	10	0	1	1
001010	0	0	0	0	0	00	0	1	0
001011	0	0	0	0	1	00	0	1	0
001100	0	0	0	0	0	11	0	1	1
001101	0	0	0	0	1	10	0	1	1
011111	0	0	0	0	1	00	0	1	0
100000	1	0	0	1	0	10	1	0	1
100001	0	0	0	1	1	10	1	1	1
100010	1	0	0	1	0	00	1	0	0
100011	0	0	0	1	1	00	0	1	1
100100	1	0	1	1	0	01	0	0	0
100101	0	0	0	1	1	00	0	1	1
100110	1	0	1	1	0	01	0	0	0
100111	0	0	0	1	1	00	0	1	1
101000	0	0	0	0	0	10	0	1	1
101001	0	0	0	0	1	10	0	1	1
101010	0	0	0	0	0	00	0	1	0
101011	0	0	0	0	0	00	1	1	0
101100	0	0	0	0	0	11	0	1	1
101101	0	0	0	0	1	10	0	1	1

Las entradas don’t care, dejaremos que se completen de forma automática por el simulador, y al crear el circuito sería implementado de esta manera.



ALUControl

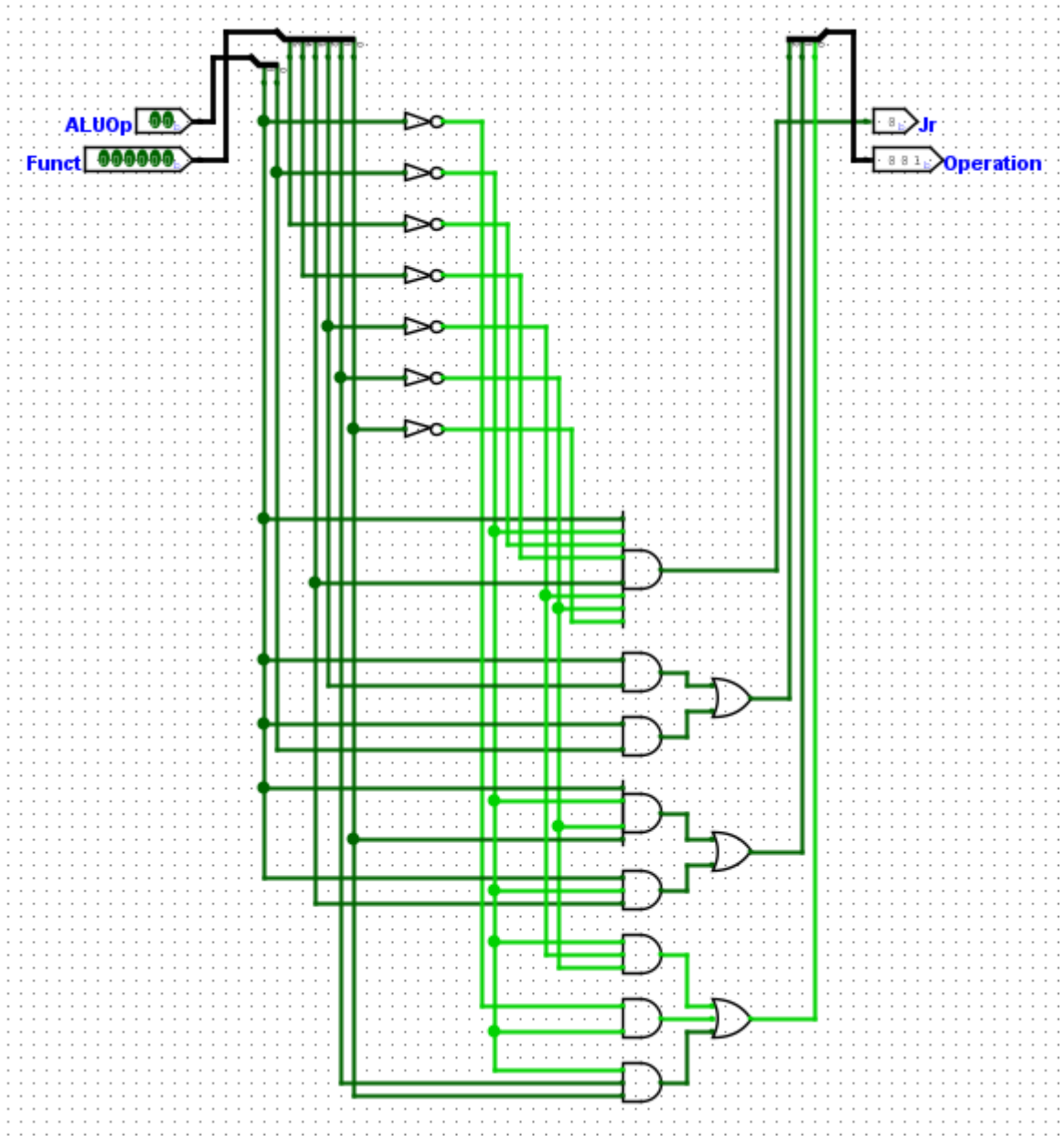
Luego de definir el tipo de instrucción que se esta ejecutando y sus datos de salida o activación, debemos igualmente decidir si se ejecutará alguna operación aritmeticológica sobre esta, para esta parte se han definido códigos de 3 bits para 6 operaciones aritmeticológicas: Sub (000), Add (001), Slt (010), And (100), Nor (101) y Or (110)

Una vez definidas, debemos tener en cuenta que esto bloque tiene como función indicar el código de la operación que realizará el ALU, además de una salida jr que indicará si se está

ejecutando esa instrucción o no. Los datos de entrada serán el Opcode que nos ha porporcionado el Control o Unidad de control y el segundo es funct, así, dependiendo del Opcode que indicará si la instrucción es R o I, y el código de 5 bits de funct, podremos saber la operación que se ejecutará y su código asignado.

Instrucción		ALUOp	funct	function ALU	ALU OP
LW		0	XXXXXX	ADD	#001
SW		0	XXXXXX	ADD	#001
BEQ		#01	XXXXXX	SUB	#000
TIPO R	ADD	10	100000	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
LHU		0	XXXXXX	ADD	#100
JR		10	#001000	NA	NA
JAL		XX	XXXXXX	NA	NA
J		XX	XXXXXX	NA	NA

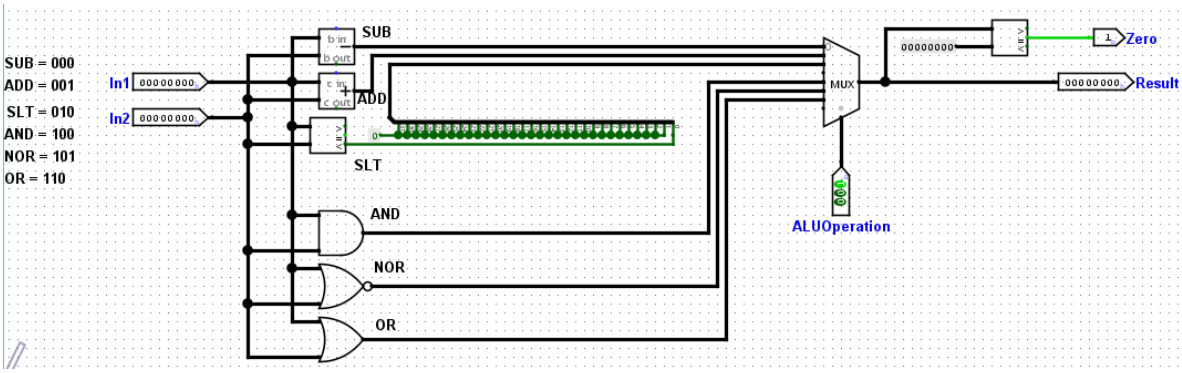
La anterior tabla de verdad, plasma la idea, basándonos en los códigos asignados en el diseño y la documentación de la arquitectura MIPS 32, a continuación, creamos la tabla de verdad en Logisim y creamos el circuito.



ALU

El anterior circuito nos ha proporcionado el código que hemos asignado para las diferentes operaciones a realizar, por lo que ahora debemos construir el bloque que nos permita llevar a cabo esta tarea, por lo que debemos tener los otros 2 datos: el contenido que hay en el primer registro a operar y lo mismo para el segundo registro, es importante tener en cuenta que estamos hablando del contenido del registro, por lo que se trata de un dato de 32 bits cada uno, además para los datos de salida, tendremos igualmente un par, para el resultado y

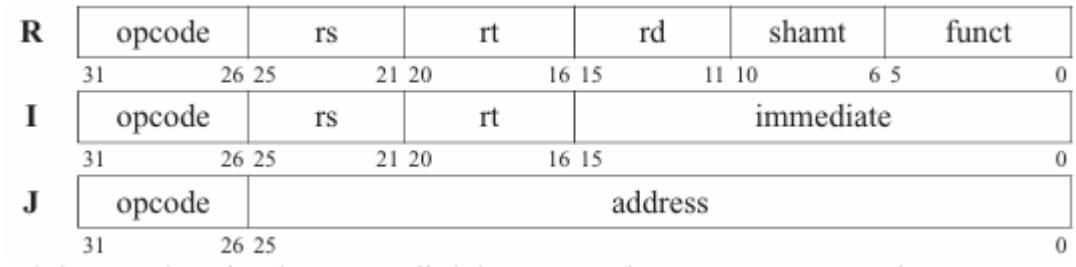
otro que será zero, que indicará si la operación realizada es cero o no, esto no usamos porque será útil para las instrucciones Branch.



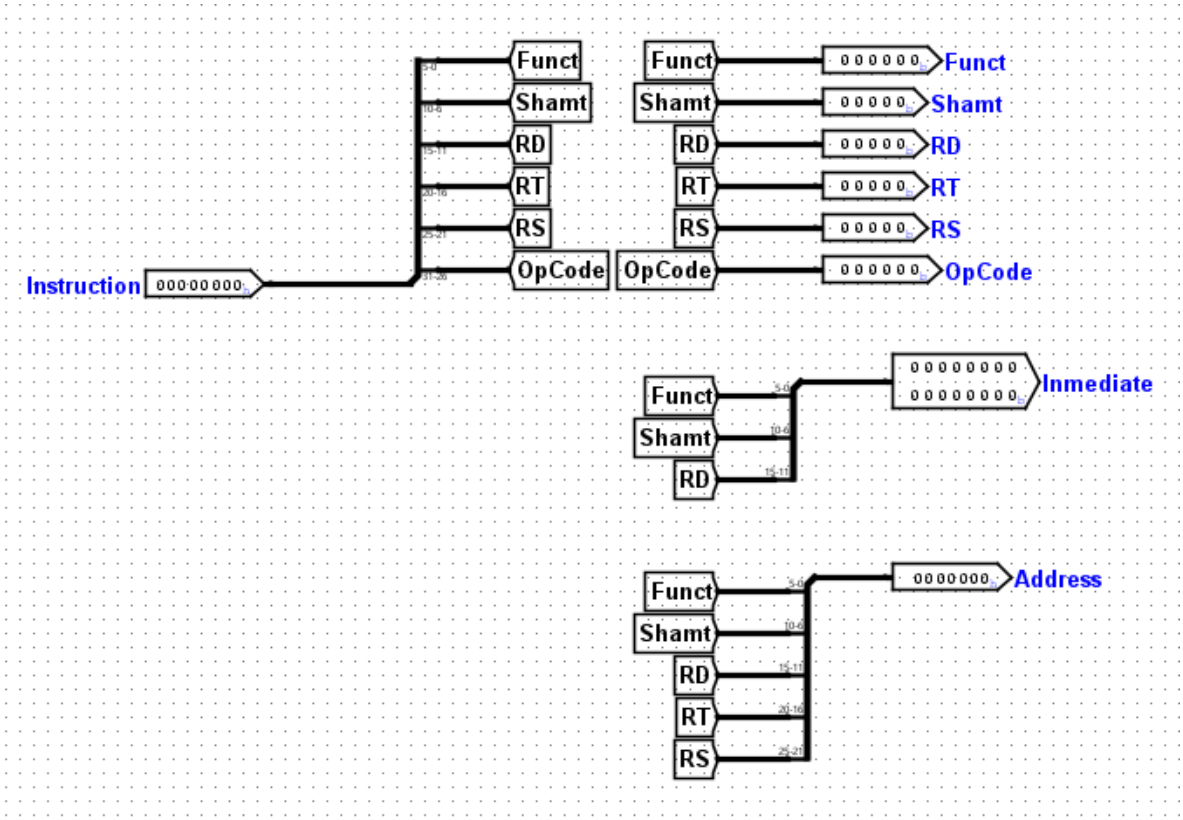
En el bloque podemos ver cómo, usando la librería de Logisim Evolution, realizaremos las 6 diferentes operaciones, luego, con ayuda el multiplexor, permitiremos el paso de que operación se ha ejecutado junto con su valor, por medio de la entrada ALUOperation, recordar que esta última es la salida del circuito anterior.

BreakDown

Retrocedemos un poco en el flujo del circuito, preparando entradas para los bloques que hemos creado anteriormente, este se encarga de desglosar la instrucción ejecutada, en sus datos de formato de instrucción, es decir, Opcode, rs, rt, rd, shamt (No utilizable en esta arquitectura), funct, inmediate y address.



En esta imagen indicamos igualmente los bits que cada dato deberá tener, para llevar a cabo esto, únicamente haremos uso de Spliters o Separadores, para obtener los bits necesarios para cada dato:



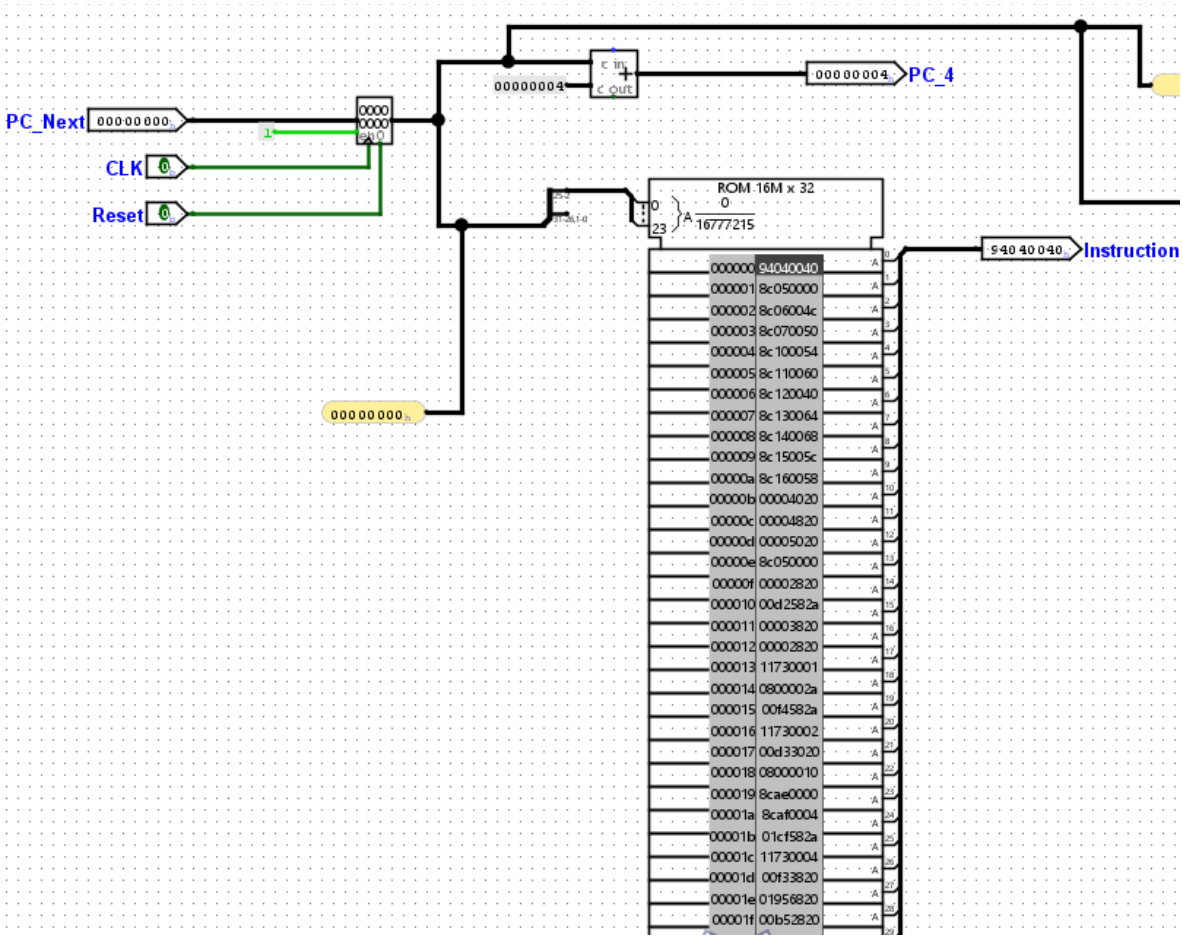
Ahora estas podremos conectarlas a los demás circuitos, pero lo haremos mas adelante, ya que por medio del flujo encontraremos algunas tomas de decisiones por medio de multiplexores, comparadores, desplazadores y demás, por lo que luego se hará énfasis en esta parte.

IFetch

Entramos a un bloque fundamental, Aquí es donde ejecutaremos cada instrucción, además de indicar la siguiente instrucción a ejecutar por medio del pc y pc+4, para poder gestionar las instrucciones que se ejecutarán, debemos tenerlas almacenadas en algún lado, específicamente en alguna memoria, pero, ¿Cuál podríamos usar? Para este caso lo más eficiente sería una memoria ROM, el motivo es principalmente que esta únicamente permite la lectura de los datos que hay allí y debemos recordad que lo que almacenaremos aquí, no será modificado, únicamente será leído.

Una vez aclarado esto, es hora de tratar los datos PC y PC_Next (pc a ejecutar), debido a limitaciones del sistema, usaremos 24 bits de Pc_Next (2-25) conectados a la ROM, y por

medio de un registro podremos sincronizar este dato sin interrumpir la instrucción ejecutada en el momento. Resta usar un sumador para sumar 4 a cada pc y conectar un dato de salida Instruction a la salida de la memoria.



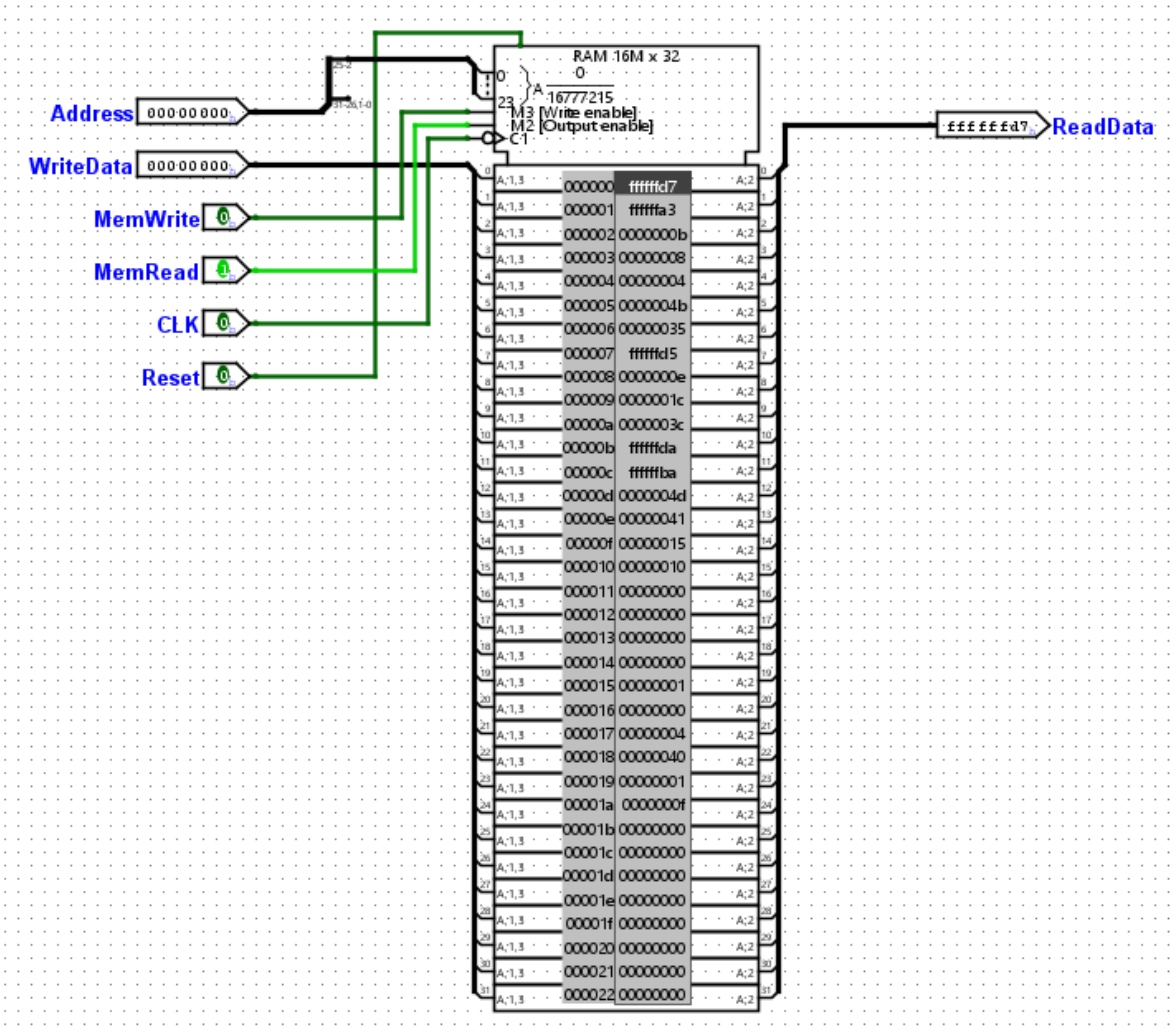
Configuración de la ROM

ROM (570,160)	
FPGA supported:	Supported
Número de bits de dir...	24
Número de bits de da...	32
Tamaño de la línea	Individual
Allow misaligned?	No
Contenidos	(clic para editar)
Etiqueta	HDL Required
Fuente de etiqueta	SansSerif Negrita 16
Etiqueta visible	No
Apariencia	Logisim-Evolution

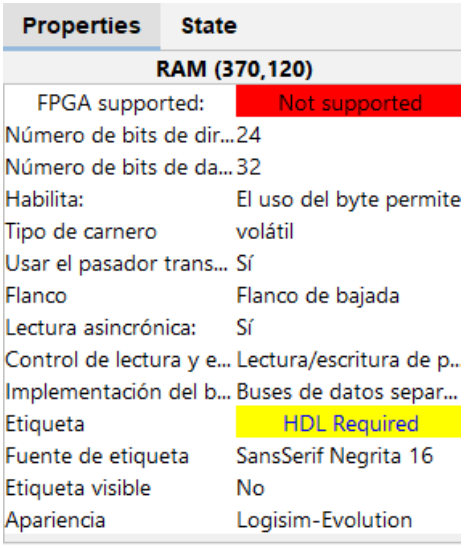
DataMemory

Ya hemos configurado la memoria de las instrucciones, por lo que faltan los datos, en esta se emplea una RAM, ya que esta, a parte de permitir lectura, también tiene la posibilidad de escribir sobre ella, que es una característica de los datos o segmento .data del código.

Los datos de entrada de este bloque únicamente serán los mismos que la de la memoria, un dato que indique la dirección a la que se apunta, otra para el dato que se escribirá, pero para que esto funcione necesitamos agregar igualmente una entrada de activación de escritura y también una de lectura, para finalizar debemos conectar el reloj y el clear o Reset. Para la salida solo será el ReadData de 32Bits que contendrá el contenido de la direccion a la que se esté apuntando.



Configuración de la memoria RAM



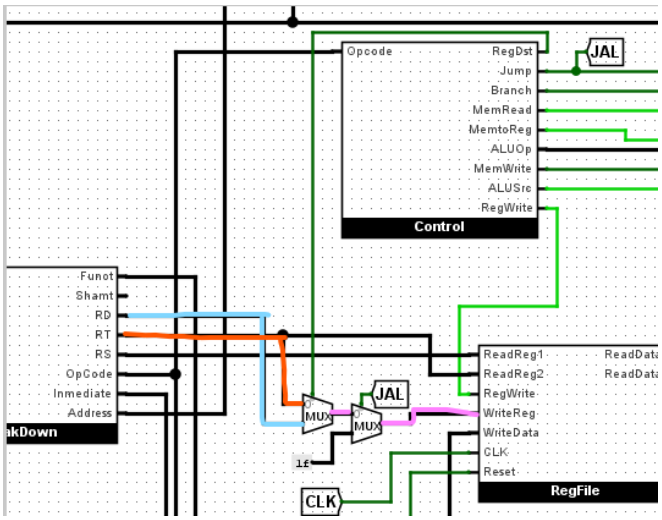
Una información importante sobre esta configuración es la propiedad Lectura Asincrónica y Flanco, con los valores indicados en la imagen permiten la perfecta sincronización con la ejecución del programa.

Conexiones del circuito

Algunas conexiones entre bloques tienen una lógica un poco compleja, aunque otros no tanto, por lo que a continuación mencionaremos estos para el correcto funcionamiento del procesador.

RD y RT con WriteReg

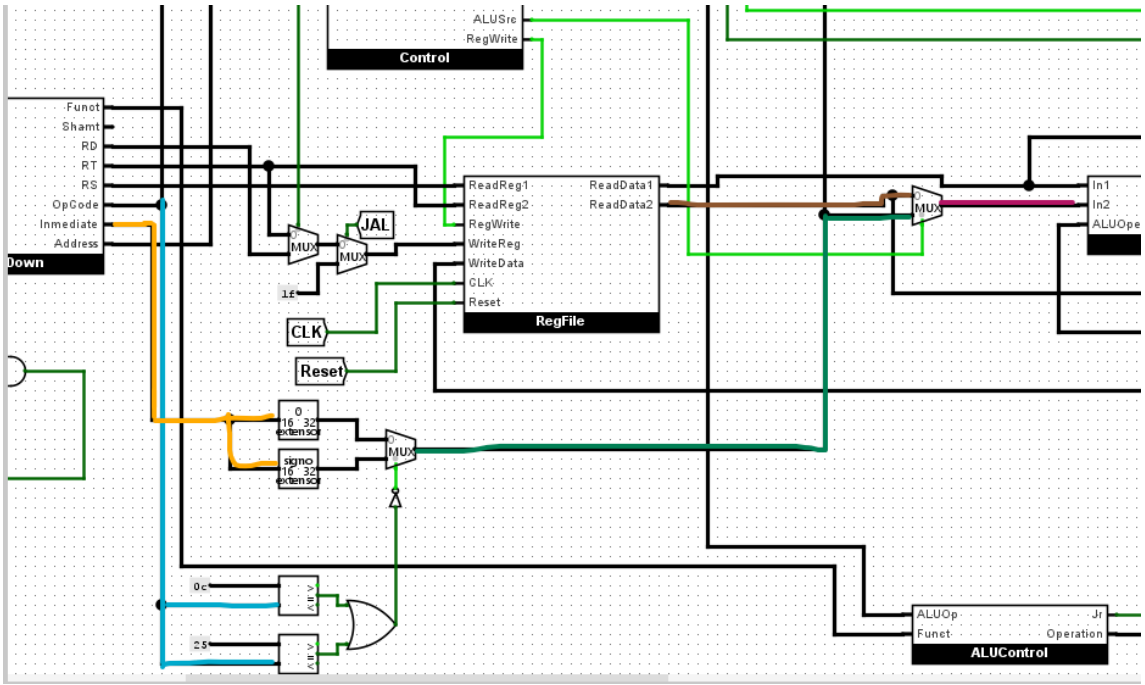
Las instrucciones Rd y Rt con WriteReg, no son directas sino que se determina por medio de multiplexores, que dependiendo de la instrucción es que se permitirá el paso de una o de otra.



Además los multiplexores tendrán los bits de selección RegDst y Jump(JAL), este último permitiría el paso de rs, rt o en caso de estar activado, la constante 0x1f de 5 bits (11111) será la que entre al bloque RegFile, haciendo referencia al registro ra.

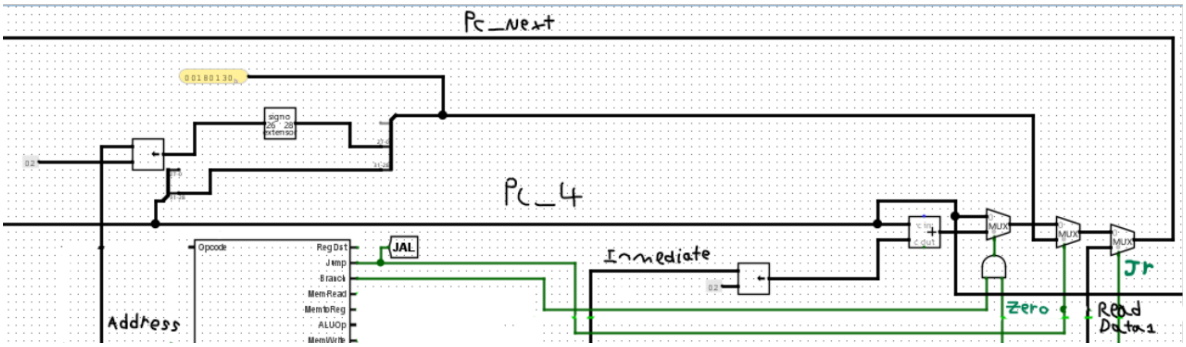
Opcode, Inmediate y ReadData2 con ALU

Para las instrucciones andi, lhu y beq, operaremos con el dato Inmediate, pero beq realiza una extensión de signo de 16 bits, mientras que las otras 2, una extensión de ceros, mientras que para las demás, no harán uso de este, por lo que dependiendo del Opcode se decidiría la extensión a realizar y a continuación ALUSrc dirá si quien pasa al ALU será uno u otro.



Los comparadores que interactúan con Opcode, se encargan de verificar si la instrucción es lhu o andi, en caso de ser alguna de los dos, permitirá el parso del extensor de ceros, luego, el ALUSrc decidirá si quien entra a ALU es readData2 o el valor Inmediate luego de su extensión.

Cálculo de PC_Next

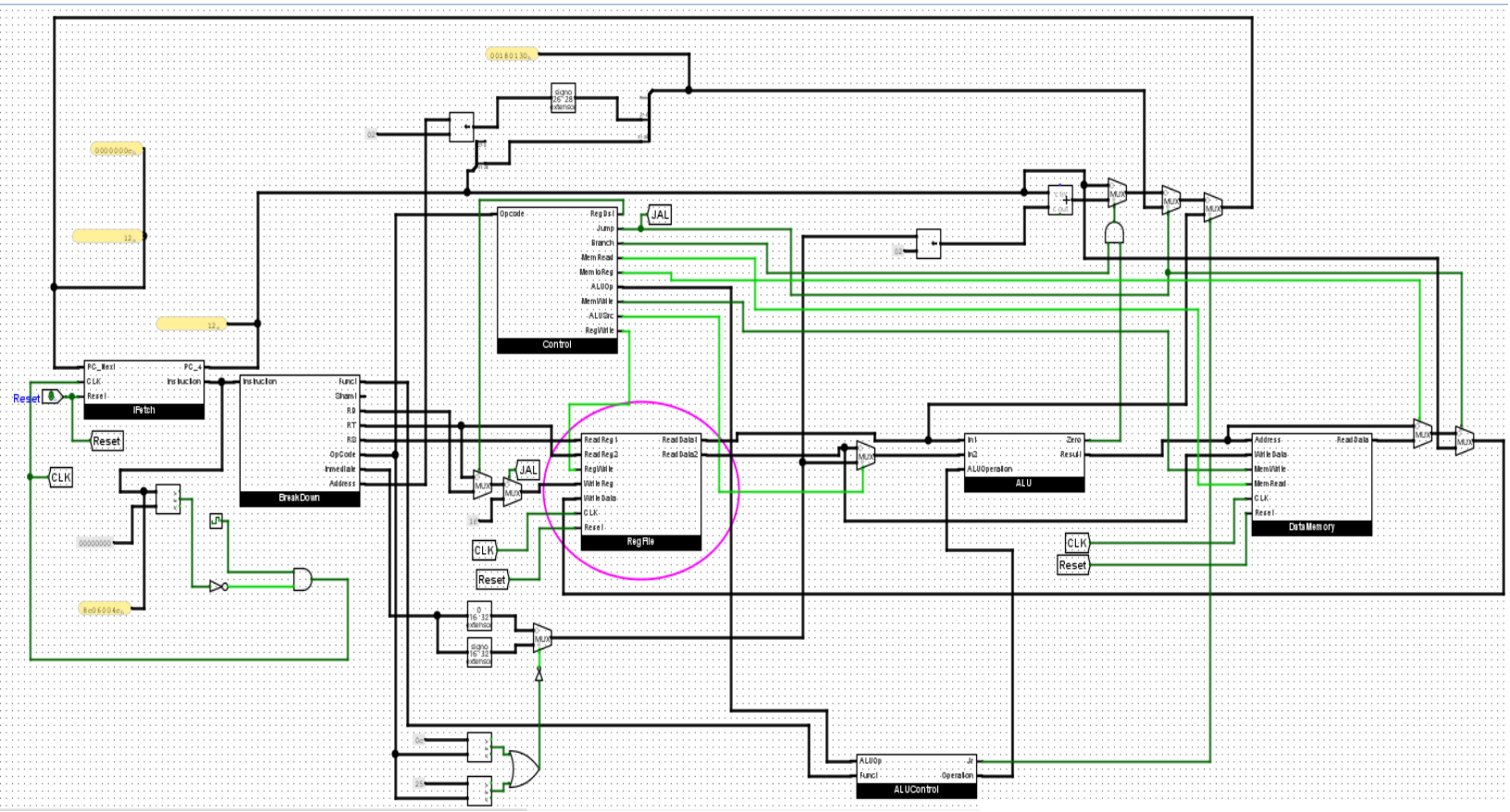


Por detalles de Logisim Evolution, para calcular pc_next en caso de haber un Branch, debemos pasarlo por un sumador, entre el pc_4 sumado a el valor inmediate desplazado dos bits a la derecha, de lo contraria, el pc_4 será el calculado en el Ifetch, luego en caso de haber un Jump, se decidirá entre la dirección calculada por el multiplexor anterior o el calculo de la dirección del salto.

Esta dirección se calcula desplazando 2 bits a la derecha el contenido de address, para luego extender con signo, de 26 a 28 bits, para finalmente concatenar estos bits menos significativos, con los 4 bits mas significativos de pc_4, dando como resultado un valor de 32 bits. Si el bloque BreakDown ha sido creado correctamente, estas direcciones deben apuntar al objetivo deseado.

Si se está ejecutando un salto, pasará la dirección calculada o el valor del anterior multiplexor, para finalizar, el ultimo mux decide entre un valor leído o la dirección anterior, en caso de que jr esté activado, ReadData1 sería el valor que hay almacenado en $ra, de lo contrario el pc_next será la siguiente instrucción.

Una vez aclarado estas conexiones, el resto de conexiones son simples y directas, por lo que únicamente se debe encargar de conectar los datos correspondientes a cada bloque.



Código del problema

Para la simulación del procesador, realizaremos un programa sencillo para implementarlo a la arquitectura para ver su funcionamiento, el programa será el siguiente: Dado un vector de 16 números aleatorios entre -100 y 100, ordenarlo de manera descendente y luego indicar cuantos de sus elementos son números pares y cuantos son impares.

Una buena práctica a la hora de programar en bajo nivel, es realizar un prototipo en alto nivel, por lo que, en este caso, se ha realizado un código en Python que cumpla con lo solicitado:

```python
vector = [-41, -93, 11, 8, 4, 75, 53, -43, 14, 28, 60, -38, -70, 77, 65, 21]
n = len(vector)
pares = 0
impares = 0
print("Vector desordenado:", vector)

for i in range(n):
    for j in range(n - 1):
        if vector[j] < vector[j + 1]:
            auxiliar = vector[j]
            vector[j] = vector[j + 1]
            vector[j + 1] = auxiliar

# Contador de paridad
for elemento in vector:
    if elemento % 2 == 0:  # par
        pares += 1
    else:  # impar
        impares += 1

print("Vector ordenado:", vector)
print("Número de pares:", pares)
print("Número de impares:", impares)
```

```
Vector desordenado: [-41, -93, 11, 8, 4, 75, 53, -43, 14, 28, 60, -38, -70, 77, 65, 21]
Vector ordenado: [77, 75, 65, 60, 53, 28, 21, 14, 11, 8, 4, -38, -41, -43, -70, -93]
Número de pares: 7
Número de impares: 9
```

Ahora implementando a programación a bajo nivel:

```
.data
vector: .word -41, -93, 11, 8, 4, 75, 53, -43, 14, 28, 60, -38, -70, 77, 65, 21
n: .word 16
pares: .word 0
impares: .word 0
incremento: .word 0 # i
incremento_2: .word 0 #j
decremento: .word 1
auxiliar: .word 0
aumento: .word 4
pila: .word 64
comparador: .word 1 # comparador para el loop exterior
comparador_2: .word 15 # comparador para el loop interior (len(arreglo)-1)
.text

main:
        lhu $a0, 64($zero) # Cargar el valor de n en $a0 usando lhu # $a0 = n (longitud del vector)

        lw $a1, 0($zero) # $a1 = base del vector (primer elemento)
        lw $a2, incremento # i
        lw $a3, incremento_2 # j
        lw $s0, decremento
        lw $s1, pila # 64
```

```
        lw $s2, n # tamaño del vector
        lw $s3, comparador # este valor es 1, para usarlo como comparador en los loops
        lw $s4, comparador_2 # comparador para el loop interior (len(arreglo)-1)
        lw $s5, aumento
        lw $s6, auxiliar

        add $t0, $zero, $zero # Inicializamos el contador de numeros pares en 0
        add $t1, $zero, $zero # Inicializamos el contador de numeros impares en 0
        add $t2, $zero, $zero # Inicializamos el indice en 0
        lw $a1, 0($zero) # $a1 = base del vector (primer elemento)
        and $a1, $zero, $zero # Inicializar $a1 a 0 usando `and`

loop_exterior:
        slt $t3, $a2, $s2 # si $a2 < $s2 (el indice (i) es menor a el tamaño del vector) entonces $t3 = 1, sino, $t3 = 0
        add $a3, $zero, $zero
        add $a1, $zero, $zero
        beq $t3, $s3, loop_interior # Si la condicion anterior se cumple, pasamos a loop_interior

        j fin_loop

loop_interior:
        slt $t3, $a3, $s4 # si $a3 < $s4 (el indice (j) es menor a el tamaño del vector - 1) entonces $t3 = 1, sino, $t3 = 0
        beq $t3, $s3, condicional_interna # Si la condicion anterior se cumple, pasamos a loop_interior
```

```
        add $a2, $a2, $s3
        j loop_exterior

condicional_interna:
        lw $t6, 0($a1) # guardamos en $t6, el elemento actual vector[j]
        lw $t7, 4($a1) # guardamos en $t7, el elemento siguiente vector[j+1]

        slt $t3, $t6, $t7 # si $t6 < $t7 (vector[j] < vector[j+1]) entonces $t3 = 1, sino, $t3 = 0
        beq $t3, $s3, ordenar # usaremos -s3 = comparador = 1 para reciclarlo en este beq, es decir, si se cumple lo anterior, ordenamos
        add $a3, $a3, $s3 # aumentamos en 1 el contador j
        add $t5, $t4, $s5 # pasamos al siguiente siguiente elemento y sera el nuevo elemento siguiente
        add $a1, $a1, $s5
        j loop_interior


ordenar:
        lw $s6, 0($a1) # auxiliar = vector[j]
        lw $t4, 4($a1) # vector[j] = vector[j+1]
        or $t5, $s6, $zero # Copiar $s6 a $t5 usando `or`  add $t5, $zero, $s6  vector[j+1] = auxiliar
        sw $t4, 0($a1)
        sw $t6, 4($a1)
        add $a3, $a3, $s3 # aumentamos en 1 el contador j
        add $t5, $t4, $s5 # pasamos al siguiente siguiente elemento y sera el nuevo elemento siguiente
```

```
        add $a1, $a1, $s5

        j loop_interior

fin_loop:



        # inicializamos los registros de pares e impares
        lw $s0, pares
        lw $s1, impares


        add $a1, $zero, $zero # Inicializar a1 en 0, que será la direccion base del vector
        add $t2, $zero, $zero

contador_paridad:

        beq $t2, $a0, salir # si el contador es igual al tamaño del vector, salimos

        lw $t0, 0($a1) # Valor base del vector
```

```
        andi $t1, $t0, 1        # Hacemos and en el bit menos significativo entre $t0 y 1 y 1 bit menos significativo se guarda en $t1
                                # Si el bit menos significativo es 1, significa que es impar, si es 0, es par

        beq $t1, $zero, es_par # Si $t1 == 0, el número es par
        j es_impar


es_par:
        add $s0, $s0, $s3 # aumentamos en 1 el numero de pares
        add $a1, $a1, $s5 # pasamos al siguiente elemento del vector
        add $t2, $t2, $s3 # aumentamos el indice del loop
        j contador_paridad

es_impar:
        add $s1, $s1, $s3
        add $a1, $a1, $s5
        add $t2, $t2, $s3 # aumentamos el indice del loop
        j contador_paridad
salir:

        # find del programa
        # los numeros pares e impares estarian guardados en $s0 y $s1 respectivamente
        sw $s0, pares
        sw $s1, impares
```

Ejecutar este programa nos lanza estos valores de los diferentes registros y direcciones:

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	16
$a1	5	64
$a2	6	16
$a3	7	0
$t0	8	-93
$t1	9	1
$t2	10	16
$t3	11	0
$t4	12	77
$t5	13	81
$t6	14	-70
$t7	15	-93
$s0	16	7
$s1	17	9
$s2	18	16
$s3	19	1
$s4	20	15
$s5	21	4
$s6	22	75
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	6144
$sp	29	12284
$fp	30	0
$ra	31	0
pc		12532
hi		0
lo		0

Vector antes de ser ordenado:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	-41	-93	11	8	4	75	53	-43
0x00000020	14	28	60	-38	-70	77	65	21

Vector ordenado:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	77	75	65	60	53	28	21	14
0x00000020	11	8	4	-38	-41	-43	-70	-93

Observe que el código no hace uso de las instrucciones sub, nor, jal y jr, pero para probar el correcto funcionamiento de este, se ha decidido extender el código con instrucciones innecesarias para el problema, para probar la ejecución de estas en el circuito, además se ha agregado un pedazo de código para agregar el vector ordenado en los registros $a0-$t7 y $s2-$s5 ($s0 y $s1 son usados para almacenar los números pares e impares respectivamente).

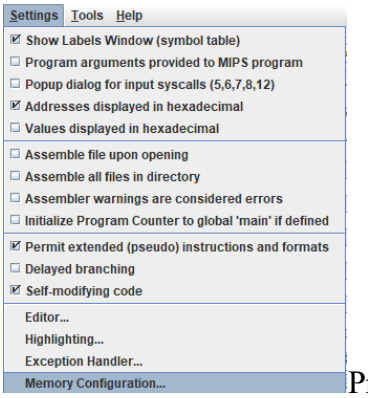
```
        sw $s1, impares
        jal probar
        j fin_programa
probar:
        nor $t9, $a0, $t3 # prueba de instrucciones
        sub $t8, $s4, $s5 # restamos $s4(15) - $s5(4) = 11
        # traer el vector ordenado a los registros para visualizarlo en logisim evolution
        lw $a0, 0($zero)
        lw $a1, 4($zero)
        lw $a2, 8($zero)
        lw $a3, 12($zero)
        lw $t0, 16($zero)
        lw $t1, 20($zero)
        lw $t2, 24($zero)
        lw $t3, 28($zero)
        lw $t4, 32($zero)
        lw $t5, 36($zero)
        lw $t6, 40($zero)
        lw $t7, 44($zero)
        lw $s2, 48($zero)
        lw $s3, 52($zero)
        lw $s4, 56($zero)
        lw $s5, 60($zero)
        jr $ra
fin_programa:
```

$a0	4	77
$a1	5	75
$a2	6	65
$a3	7	60
$t0	8	53
$t1	9	28
$t2	10	21
$t3	11	14
$t4	12	11
$t5	13	8
$t6	14	4
$t7	15	-38
$s0	16	7
$s1	17	9
$s2	18	-41
$s3	19	-43
$s4	20	-70
$s5	21	-93
$s6	22	75
$s7	23	0
$t8	24	11
$t9	25	-17
$k0	26	0
$k1	27	0
$gp	28	6144
$sp	29	12284
$fp	30	0
$ra	31	12536
pc		12616

Simulación

Segmento .data

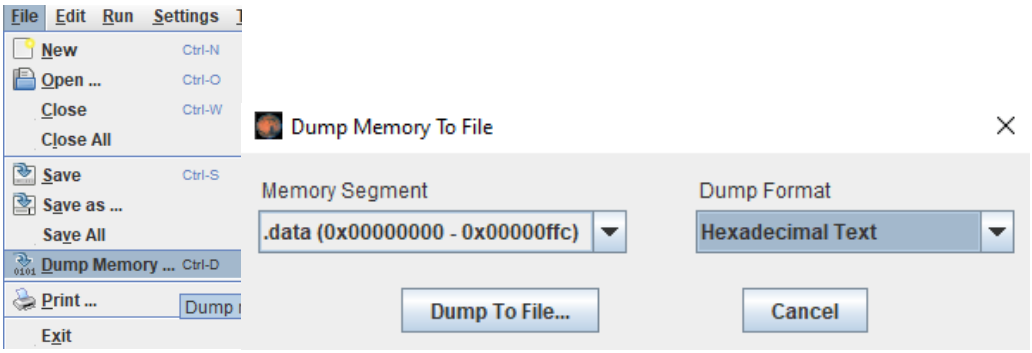
El procesador a sido diseñado para que la dirección del segmento .data inicie en la dirección 0x00000000, por lo que, a la hora de ensamblar el código para pasar el segmento a un archivo de texto, debemos iniciar las direcciones de los datos en 0 directamente desde MARS.



Presionar en Memory Configuration…

Seleccionar Compact, Data at Address 0

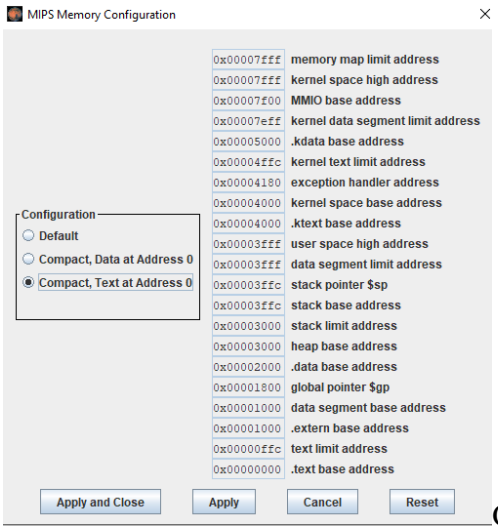
Luego presionar en Apply and Close y podremos ensamblar y llevar a un archivo de texto.



Luego seleccionar .data y formato Hexadecimal Text, a continuación, presionar Dump To File y nombrar el archivo.

Segmento .text

Debemos seguir el mismo flujo que con la memoria RAM, las instrucciones del segmento .text, deben iniciar en la dirección 0x00000000.

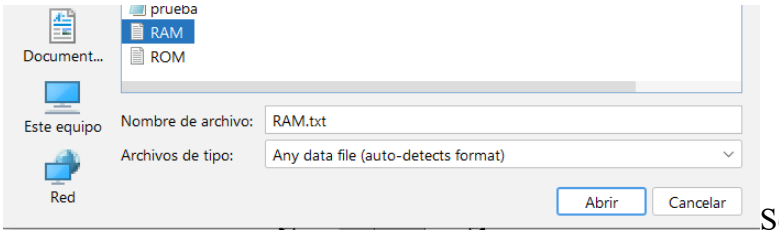


Compact, Text at Address 0

Luego de aplicar este cambio, igualmente exportamos las instrucciones como un archivo de texto.

Memoria RAM

Debes dirigirte a la memoria RAM en el circuito, presionar click derecho y seleccionar cargar imagen, luego de esto, buscar el archivo de texto del segmento .data

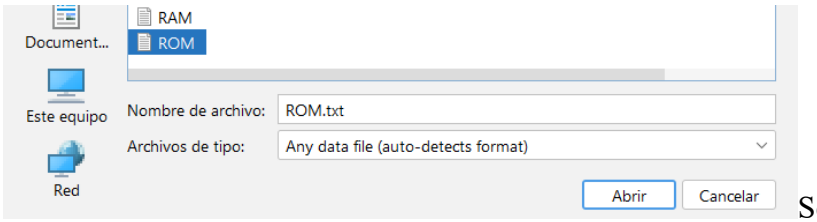


Seleccionar el segmento .data

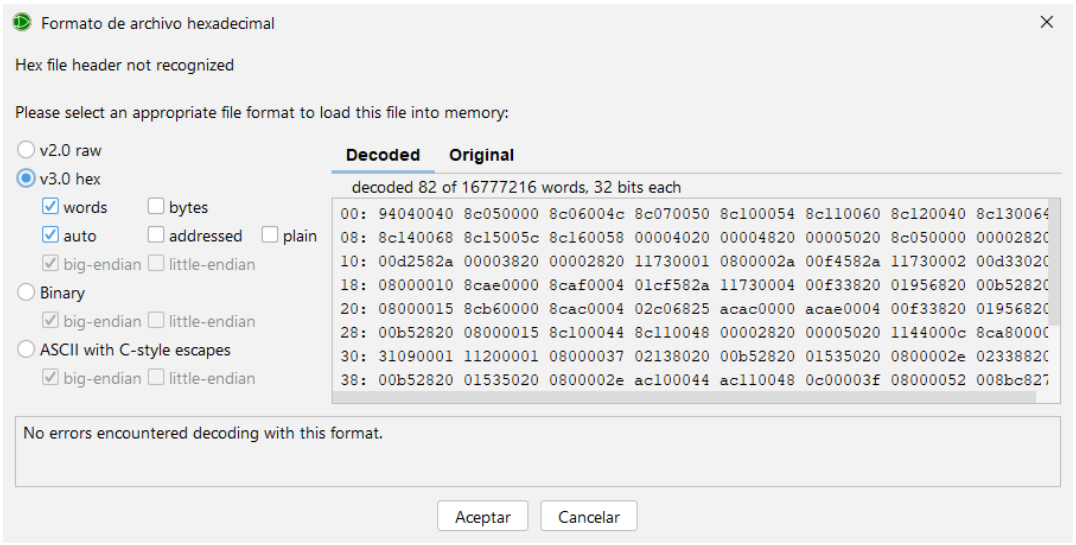
Si solicita versión, seleccionar v2.0 raw.

Memoria ROM

En el bloque Ifecth, presionar a la memoria ROM y cargar imagen



Seleccionar el segmento .text

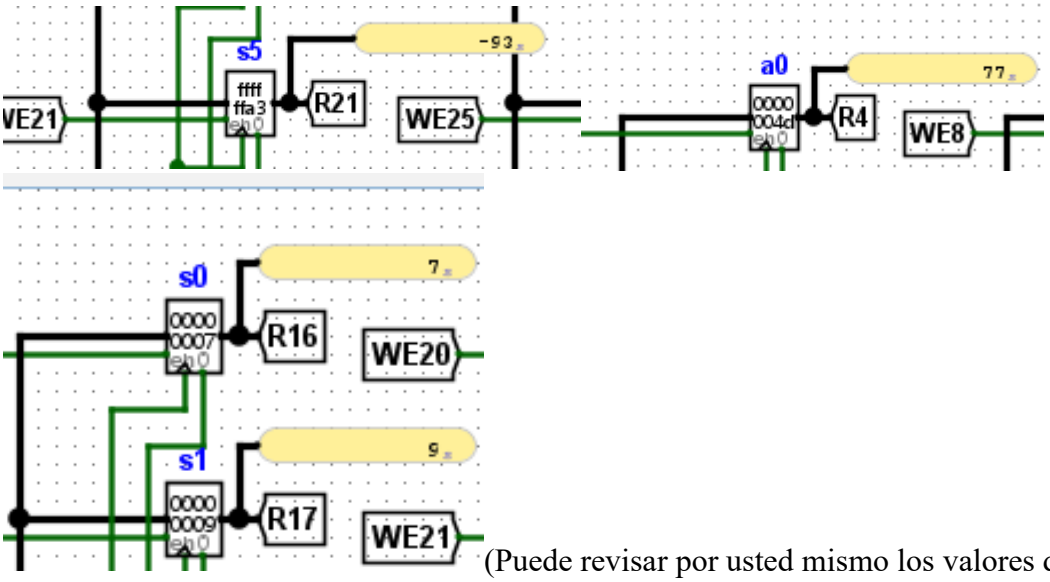


Seleccionar la versión v3.0 hex y aceptar

Ejecución

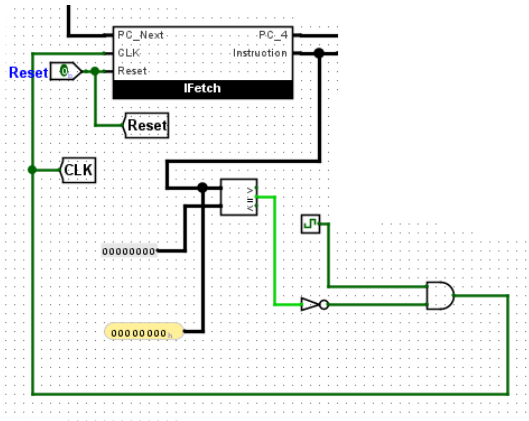
Puede seleccionar una frecuencia de reloj de 16.0 Hz y activar el reloj y ver en ejecución el procesador, el programa se ha ejecutado por completo cuando el pc_4 no cambia su valor a pesar de que el reloj siga teniendo flancos de subida y bajada, por lo que en ese momento se revisará los valores de los registros, quienes deben coincidir, con los valores que hay en MARS.

Para este caso, vamos a verificar los valores de los registros $s0 y $s1 que contienen los valores de los números pares e impares, 7 y 9 respectivamente, los registros $a0 y $s5 que contienen los valores mas altos y mas bajos del vector (77 y -93) respectivamente.



(Puede revisar por usted mismo los valores de otros registros)

Detalles de simulación



Se ha implementado la anterior arquitectura para los flancos del reloj, únicamente se encarga de comparar la instrucción a ejecutar con el código 0x00000000, donde, en caso de que la instrucción sea esa, no habrá más flancos de subida, indicando que el programa ha llegado a su final.

Como segundo detalle, se debe recalcar que, a la hora de ensamblar el código con la dirección de las instrucciones iniciadas en 0x00000000, y exportarlo como un archivo de texto para ser cargado en la memoria ROM, genera problemas con las direcciones de las instrucciones tipo J, ya que estas, dependiendo del momento del flujo en el que se ejecuten, llevará el pc a una dirección no existente, por lo que manualmente se debe cambiar el Code de estas instrucciones. Para la simulación ejecutada en ese informe ya se ha realizado esta tarea, por lo que puede ejecutarlo sin problema alguno.

Archivos de simulación

Descargar el Procesador MIPS 32 Monociclo

Descargar ROM

Descargar RAM

Descargar Código Ensamblador MIPS

Visualizar video de simulación

Conclusión

En este laboratorio se ha podido realizar un procesador MIPS 32 monociclo funcional, donde es posible utilizar las instrucciones add, and, or, nor, slt, sub, lw, sw, lhu, andi, j, jal, jr y beq. A pesar de la limitación de instrucciones útiles de la arquitectura MIPS como mul, addi, la, li, entre otras, gracias a los conocimientos adquiridos durante el semetre sobre programación en bajo nivel y diseño de sistemas digitales, se pudieron superar los diferentes desafíos que se iban presentando.

Se pudo demostrar un gran dominio de los softwares empleados, como los simuladores Logisim Evolution y MARS, las diferentes opciones y herramientas que proporcionan estos y la programación en bajo nivel.

Las diferentes estructuras tuvieron un diseño e implementación adecuada, asegurando una correcta conexión y comunicación entre los demás bloques. este proyecto representa un avance significativo en el aprendizaje práctico de sistemas digitales, ofreciendo una experiencia integral desde el diseño conceptual hasta la implementación funcional de un procesador en un entorno de simulación.

Referencias

Bibliografía

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

Velez, F. A. (2024). *Diapositivas del curso Arquitectura de Computadores.* Medellin.