Laboratorio #3
Procesador monociclo

Jonathan Andrés Granda Orrego
Jhon Alexander Botero Gomez

UNIVERSIDAD DE ANTIOQUIA
Facultad de Medicina

Objetivo

1. Analizar los requerimientos de una arquitectura de conjunto de instrucciones (ISA)
2. MIPS que permita la implementación de un procesador monociclo que ejecute la ISA.
3. Codificar, ensamblar y simular un programa para verificar el comportamiento correcto del procesador.
4. Emplear herramientas de software para el diseño y la simulación de computadores digitales.

Descripción

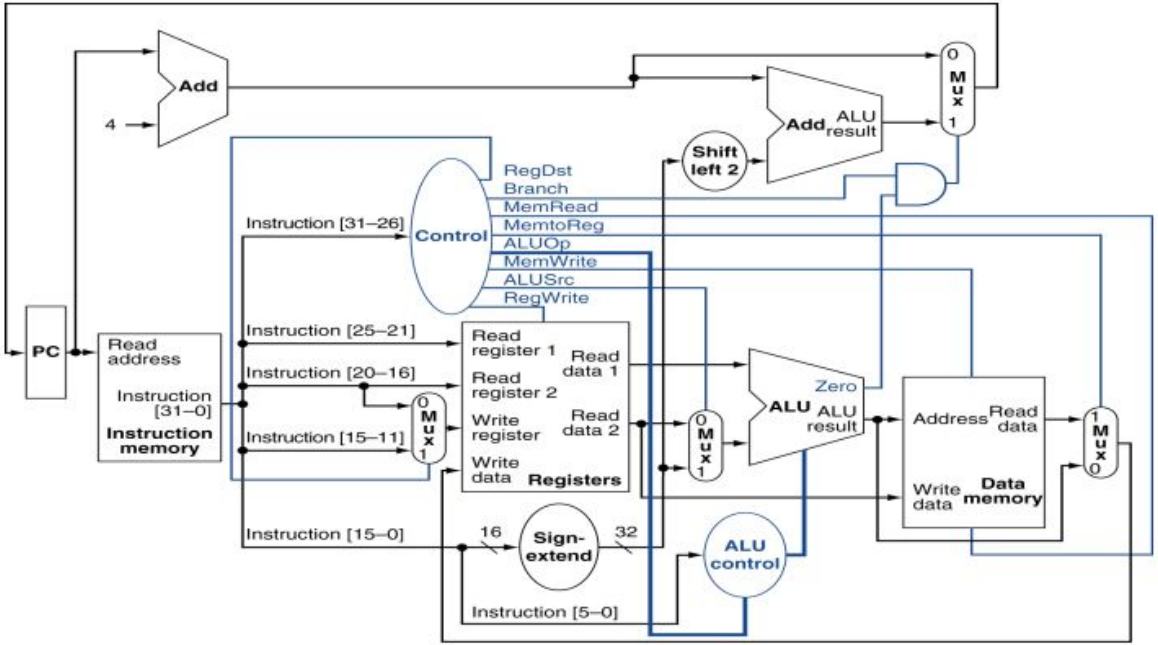
Se implementa el datapath del MIPS de 32 BITS que permita ejecutar las instrucciones de 32 bits que se muestran en la Tabla 1, si se desea agregar instrucciones adicionales lo puede hacer. La figura 1 muestra un datapath básico, si se necesita añadir otros componentes adicionales lo puede hacer. La arquitectura tendrá la capacidad para ejecutar algoritmos funcionales

Instrucciones

Tipo de instrucción	Nemónico	Formato	Ejemplo	Operación
Acceso a memoria	**lw**	I	`lw rd, offset(rs)`	R[rd] = M[R[rs] + signExtImm]
	sw	I	`sw rd, offset(rs)`	M[R[rs] + signExtImm] = R[rd]
Aritmético-lógicas	**add**	R	`add rd, rs, rt`	R[rd] = R[rs] + R[rt]
	sub	R	`sub rd, rs, rt`	R[rd] = R[rs] - R[rt]
	and	R	`and rd, rs, rt`	R[rd] = R[rs] & R[rt]
	slt	R	`slt rd, rs, rt`	R[rd] = (R[rs] < R[rt]) ? 1 : 0
	addi	I	`addi rd, rs, INM`	R[rd] = R[rs] + INM
Salto condicional	**beq**	I	`beq rs, rd, label`	If(R[rs]==R[rt]) PC = PC+1+BranchAddr
Salto y enlace (OPCIONAL)	**jal**	j	`jal funcion`	r a = PC+4 PC= (PC & 0xf0000000) | (target << 2)
Salto a registro (OPCIONAL)	**jr**	R	`jr rs`	PC=R[rs]

Tabla 1. Lista de Instrucciones

Datapath



Datapath base y unidad de control

Procedimiento

Diseño e implementación del procesador usando Logisim y Prueba funcional

a) Implemente un procesador monociclo de la arquitectura de 32 bits, incluyendo la ruta de datos y las señales de control, para poder ejecutar cada una de las instrucciones de la tabla 1.

b) Diseñe una unidad de control para el procesador monociclo que permita ejecutar las instrucciones de la tabla 1, e implemente la CPU en su totalidad.

c) Pruebe una a una las instrucciones de la tabla 1 en la arquitectura implementada.

Procedimiento

Problema a resolver

d) En el apéndice A se encuentra el problema asignado a cada equipo. Deben realizar el pseudocódigo del algoritmo que resuelve el problema, luego debe ser codificado en ensamblador, traducido a lenguaje de máquina y ejecutado en el procesador.

Puede usar el MARS como herramienta de apoyo pero tenga en cuenta que el código máquina producido por el MARS no es, necesariamente, 100% compatible con su arquitectura. Realice los ajustes necesarios al código máquina.

e) El código debe tener el llamado por lo menos a un método (OPCIONAL).

Requerimientos adicionales

Requerimientos adicionales

Para el desarrollo de la práctica se imponen las siguientes condiciones:

- Ser riguroso al momento de dar solución a los apartados del procedimiento. Todas las decisiones de diseño deben estar ampliamente justificadas.
- Los demás componentes necesarios para implementar el procesador y que se considere no están disponibles en las bibliotecas de componentes de Logisim, deben ser diseñados con la misma herramienta.
- El código ensamblador debe estar comentado. Se valorará la creatividad y elegancia en la programación a este nivel.

Problema a resolver

- El vector debe tener un marcador de finalización, el programa no puede saber previamente el tamaño del vector.
- Se valorará adicional si el programa contiene la implementación y el llamado por lo menos a una función.

equipo = equipo 17

problema = (17 mod 16) + 1 = 2

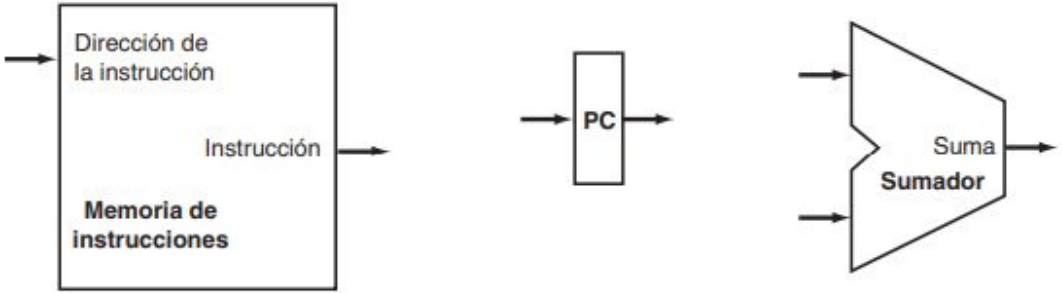
2. La operación 80x, x es el penúltimo elemento del vector.

Diseño e implementación procesador monociclo

UNIVERSIDAD DE ANTIOQUIA
Facultad de Medicina

Implementación ruta de datos

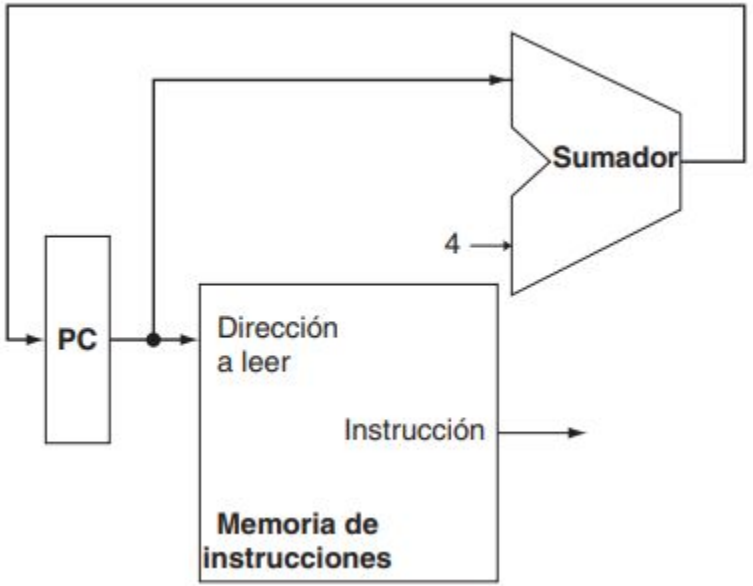
Se necesitan dos elementos de estado para almacenar y acceder a las instrucciones, y un sumador para calcular la dirección de la instrucción siguiente.



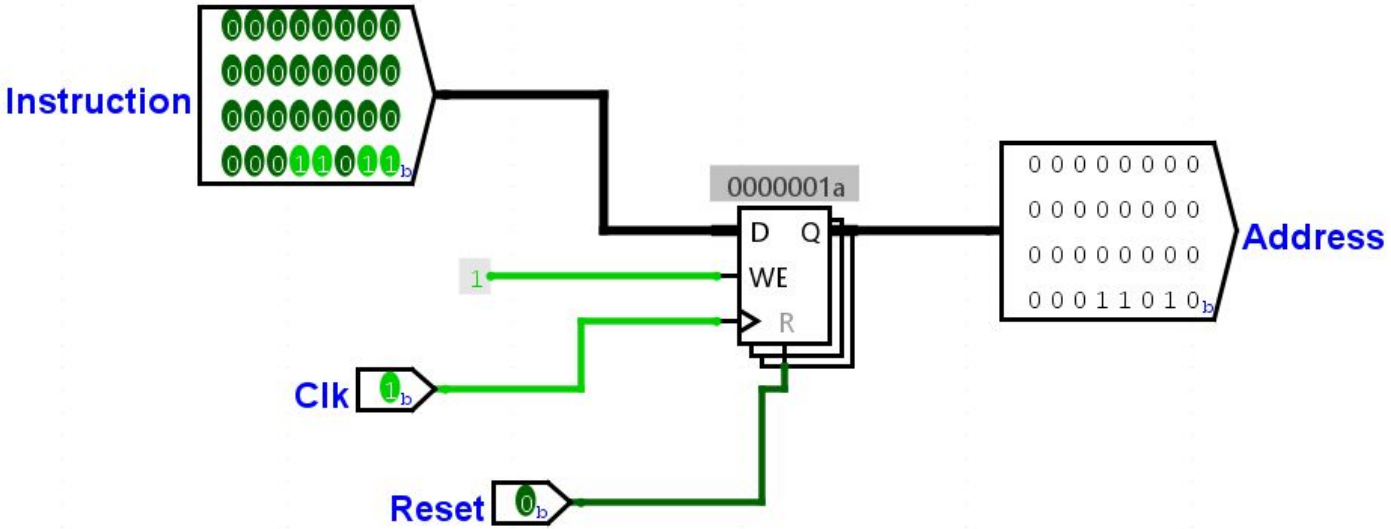
a. Memoria de instrucciones b. Contador de programa c. Sumador

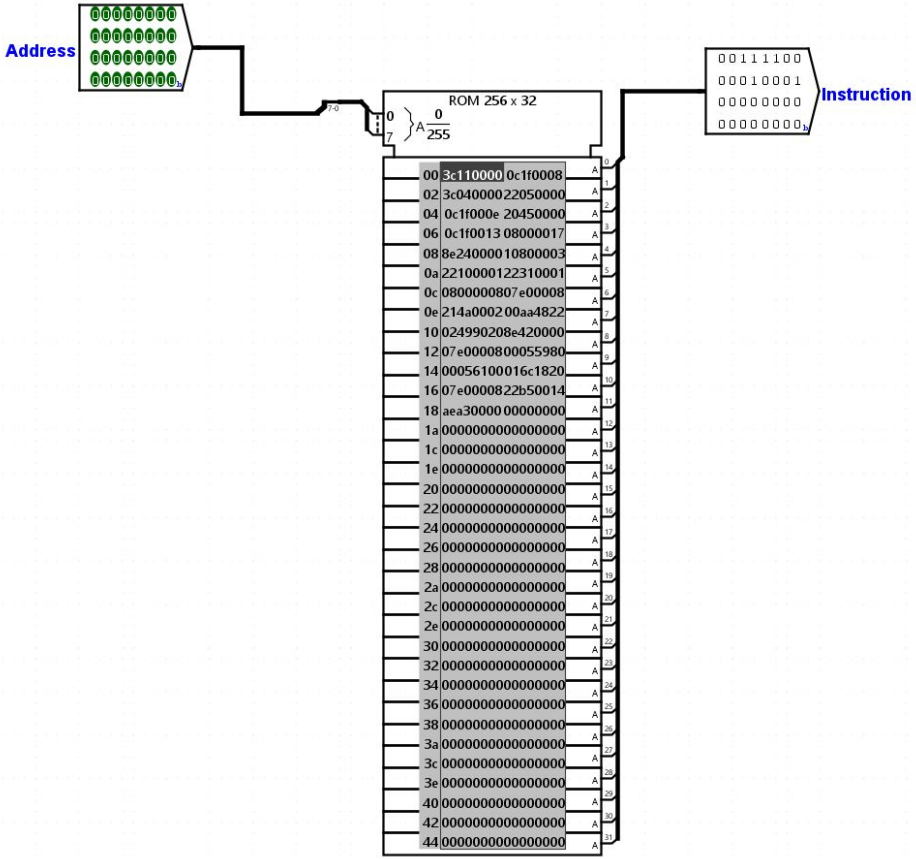
Implementación ruta de datos

En nuestro caso el almacenamiento de la instrucción, dado que la memoria está almacenando 4 bytes, guarda una instrucción por casilla. Entonces el PC aumentado (+1)



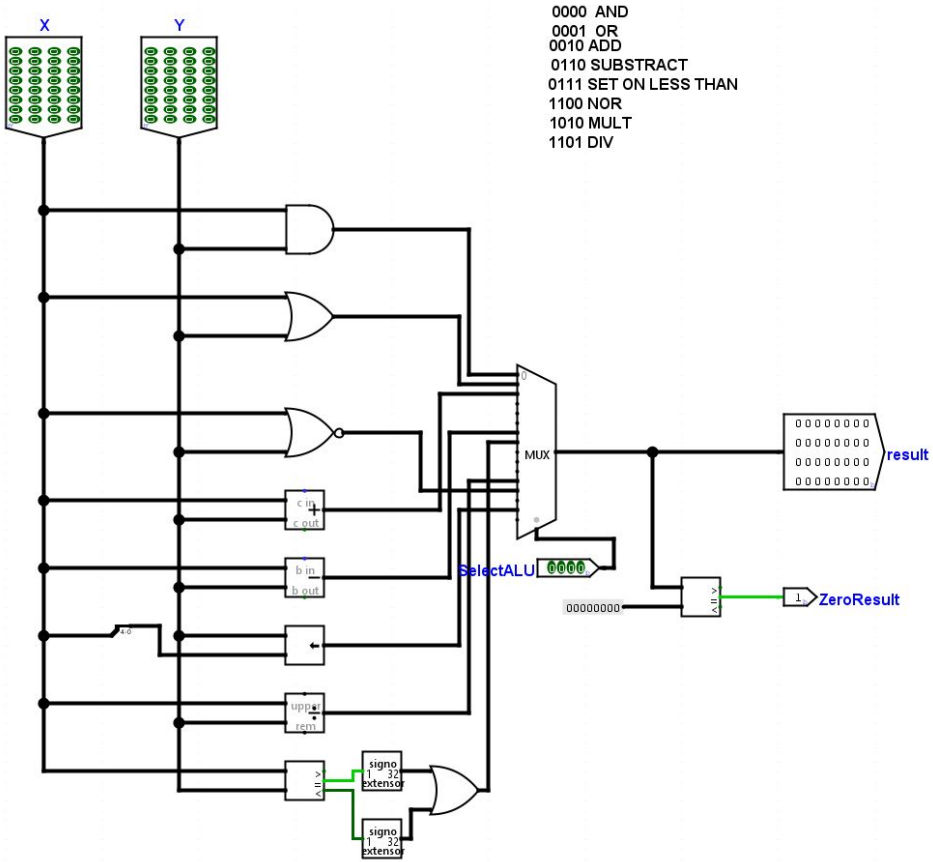
Program Counter





Memoria de Instrucciones

En este caso podemos ver que las instrucciones están en cada casilla, los bits de datos están en 32, por lo que cabe el contenido completo de la instrucción

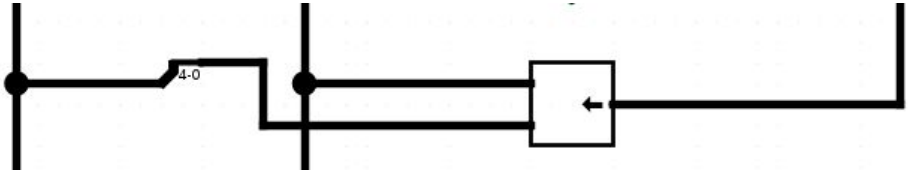


Unidad Aritmética - Lógica

Tenemos 8 operaciones en la ALU, 3 lógicas, 4 aritméticas, 1 comparativa,pero usamos un selector de 4 bits, se hace así para la integración con la Unidad de control y demás componentes

Consideraciones ALU

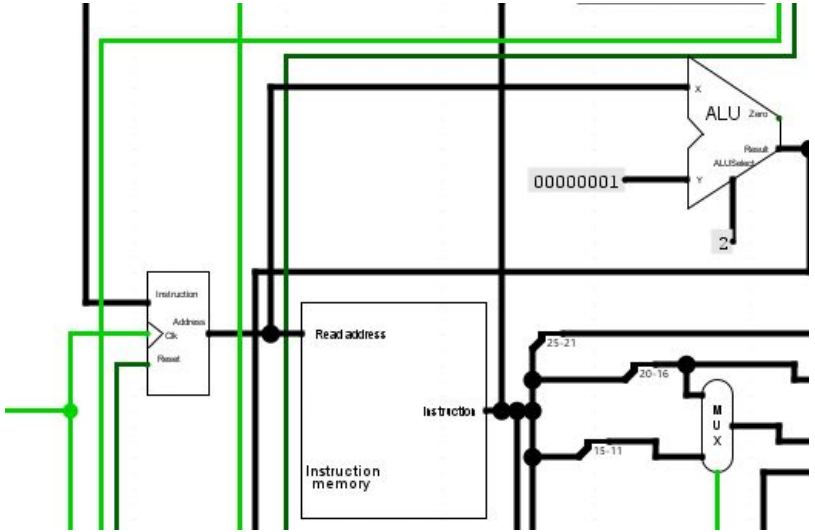
Dada la asignación específica del problema para el equipo 17, la operación de multiplicaciones se hace con desplazadores, por lo que en el esquema no sale un multiplicador.



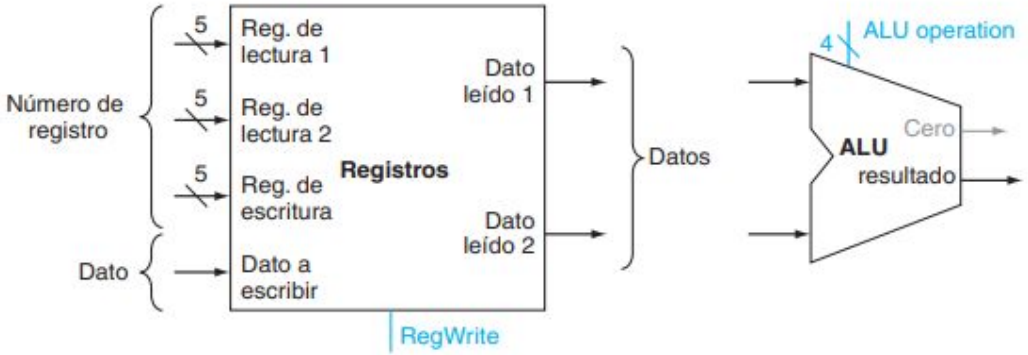
Shift left logical

Ensamble parcial

Hasta este punto se tiene la implementación del manejo de las instrucciones, para que el PC nos de la dirección de la instrucción.
La ALU tiene constante 2 por qué es la operación de suma y el 1 indica el aumento del PC en cada **Clk**



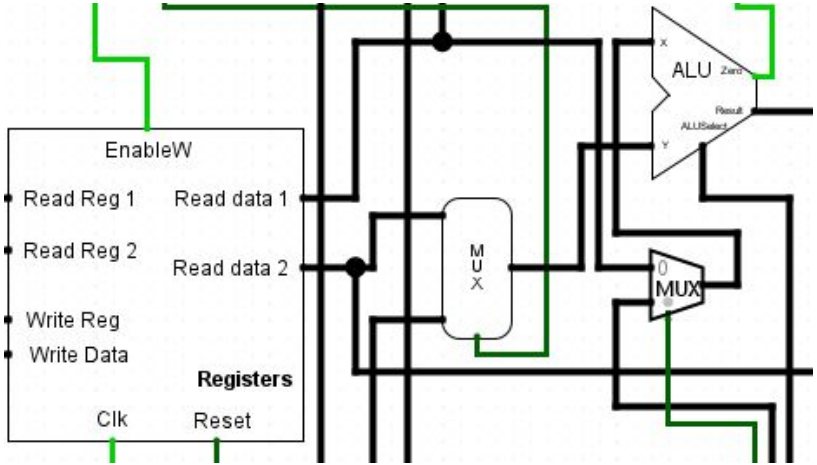
Implementación BR + ALU



El banco de registros y la ALI son los dos componentes necesarios para el funcionamiento de instrucciones de tipo R. El banco de registros devuelve la salida el contenido de los registros correspondientes a los datos que ingresan en principio(direcciones)

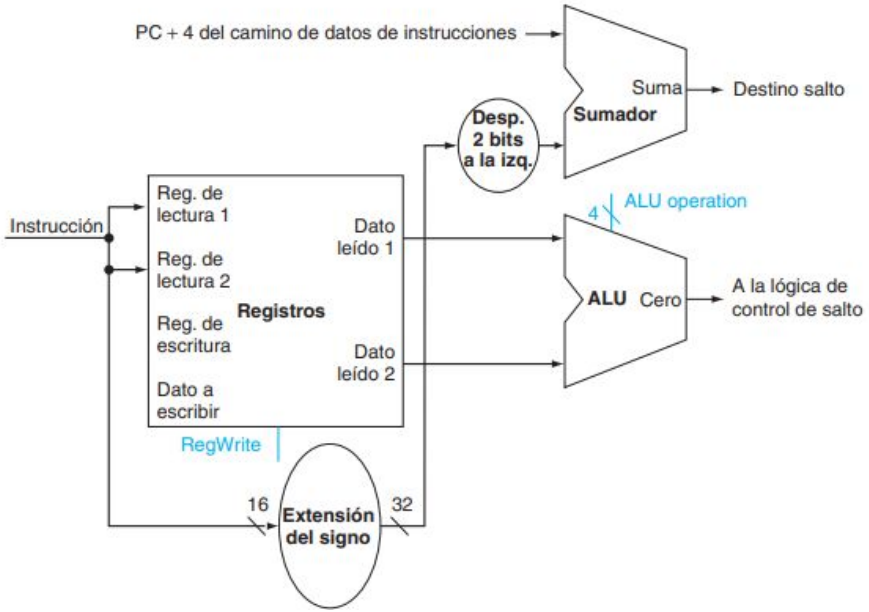
Implementación BR + ALU

Se hace uso del Banco de registrados suministrado para el laboratorio, y se usa el mismo componente ALU mostrado anteriormente



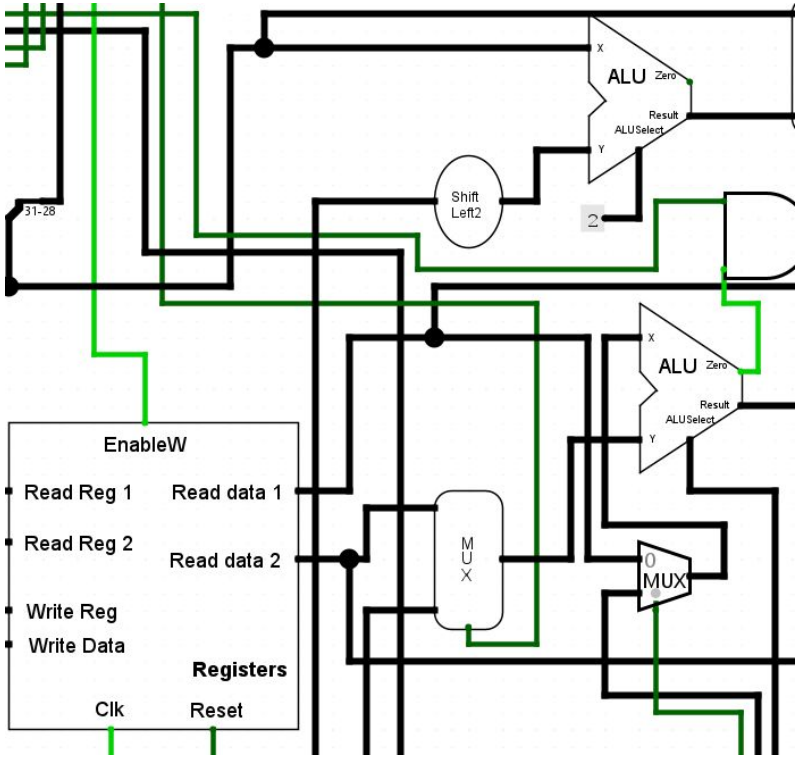
Implementación BR + ALU- Salto condicional

El camino de datos para un salto condicional utiliza la ALU para evaluar la condición de salto y un sumador aparte para calcular la dirección destino del salto como la suma del PC incrementado



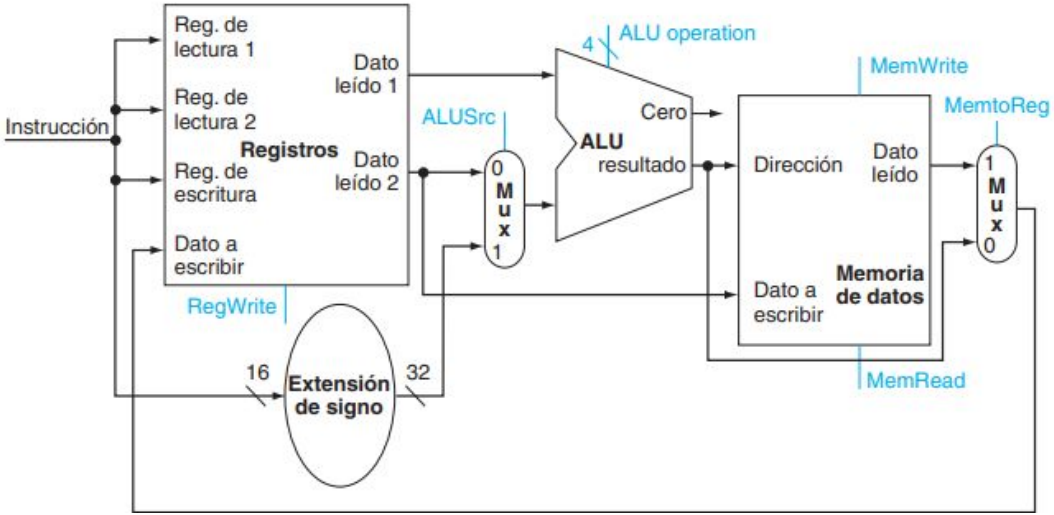
Implementación BR + ALU- Salto condicional

Como anotación adicional se hacen uso de 2 multiplexores, el segundo implica la comparación de una instrucción tipo R y un desplazamiento



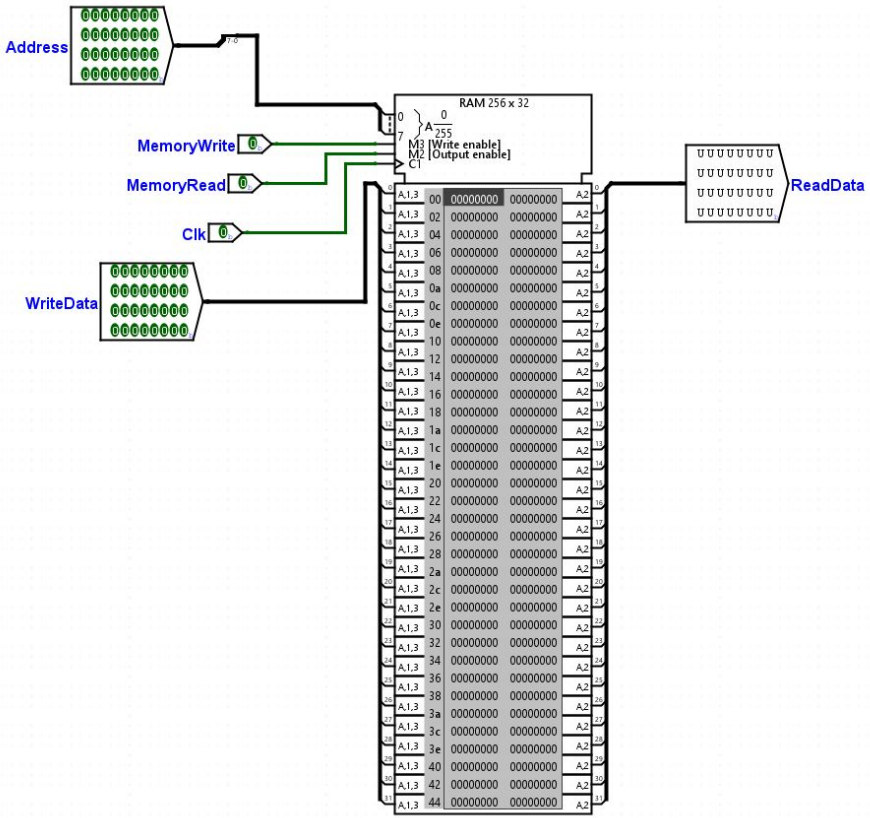
Combinación parcial de las piezas

El camino de datos para las instrucciones de memoria y tipo R.
En este caso ya involucramos l memoria de datos.



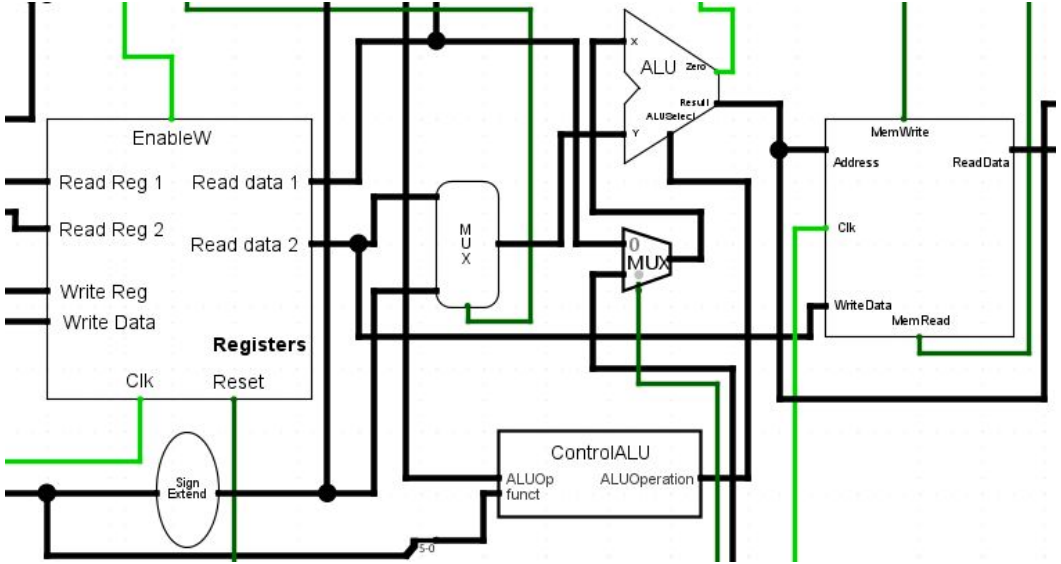
Memoria de datos

Esta nos va a servir para almacenar el vector, y recibe las salidas de los registros para escribir o leer información.



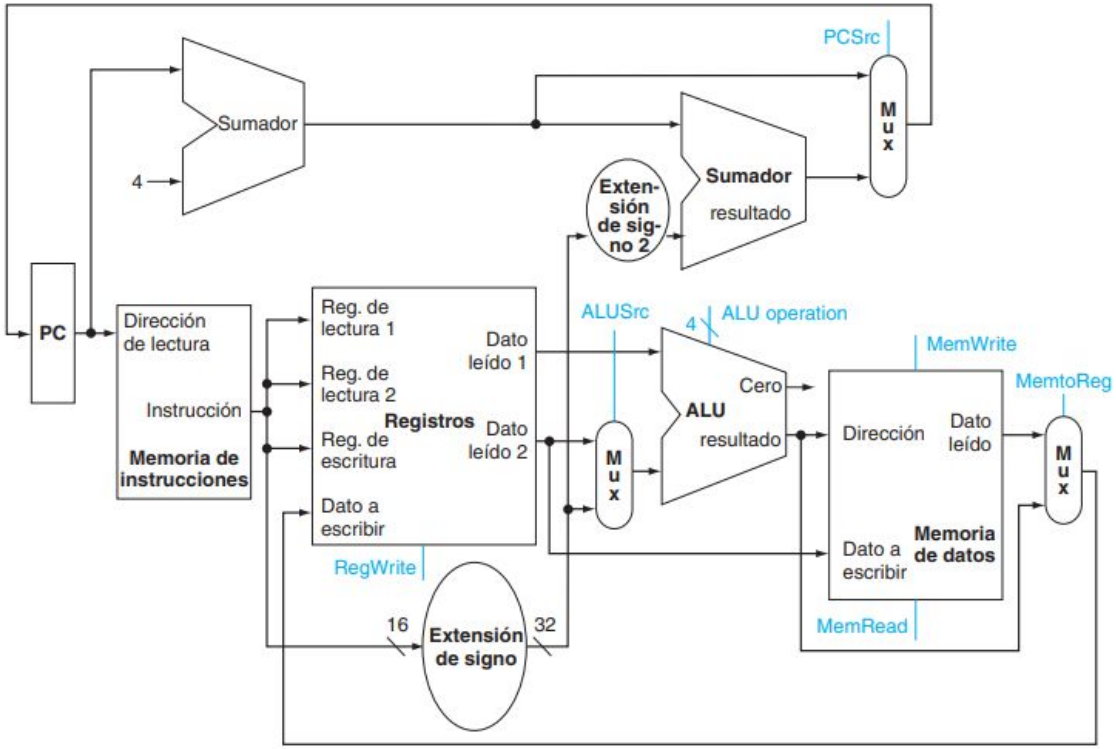
Combinación parcial de las piezas

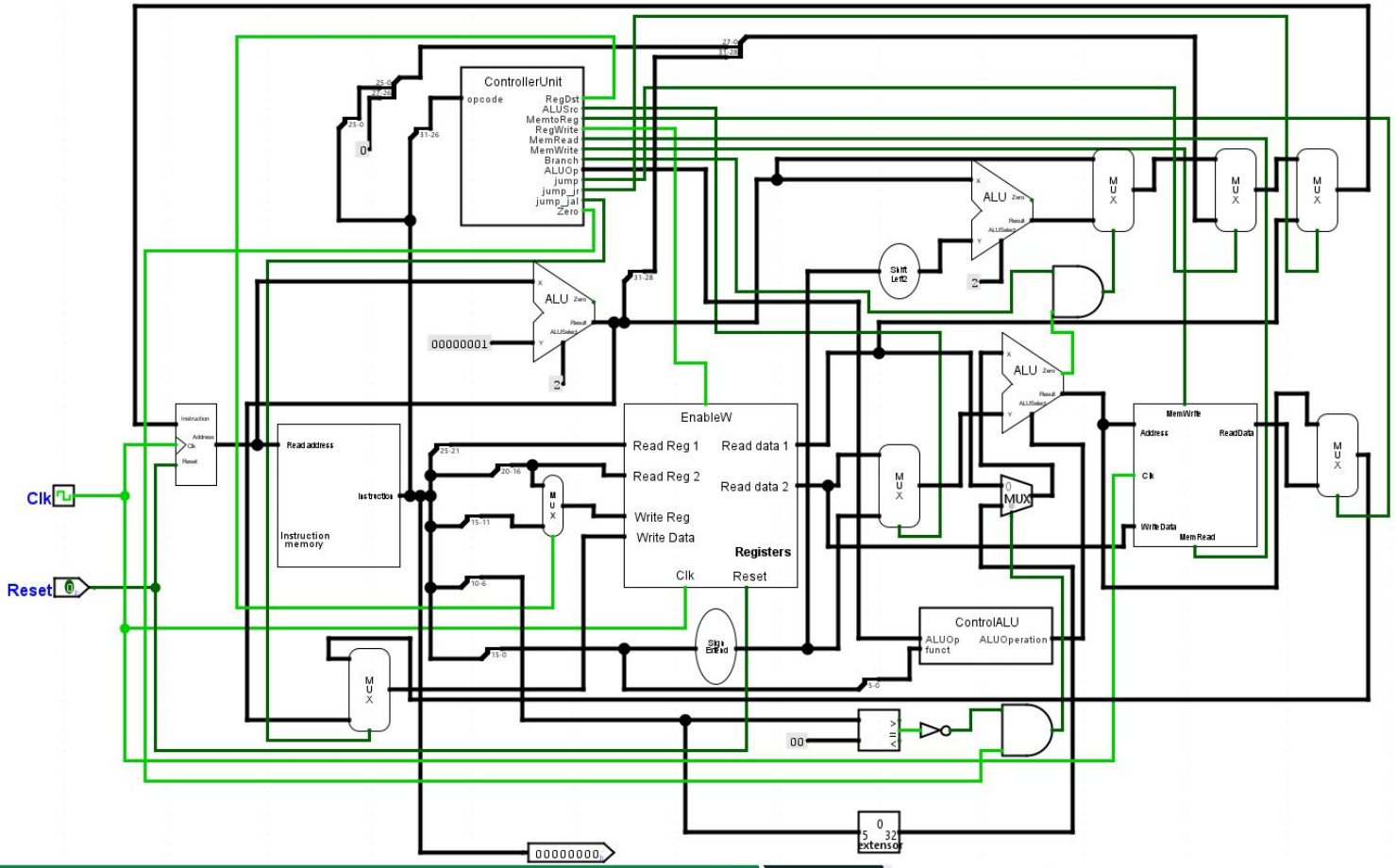
El camino de datos para las instrucciones de memoria y tipo R.



Esquema general

El esquema general implica la conexión de las 2 memorias con los registros y el PC, haciendo uso de 2 sumadores y la ALU





Esquema general
Implementación Logisim

UNIVERSIDAD DE ANTIOQUIA
Facultad de Medicina

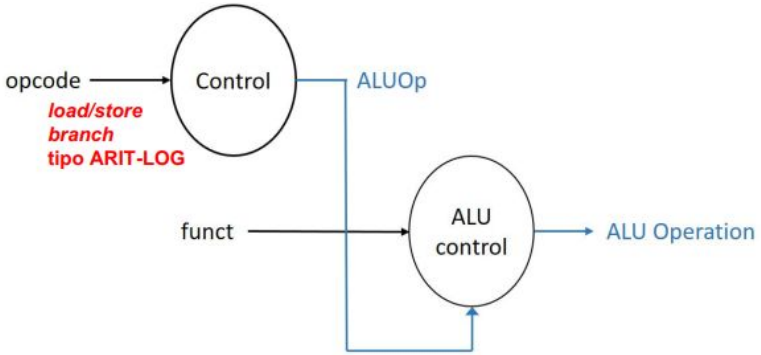
El control de la ALU

Definición de 6 combinaciones de 4 entradas de control

Líneas de control de la ALU	Función
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

El control de la ALU

Las instrucciones de referencia a memoria utilizan la ALU para calcular la dirección de memoria por medio de una suma. Para las instrucciones tipo R, la ALU debe ejecutar una de las cinco operaciones (and, or, add, sub y slt)



opcode	ALUOp	Operación	funct	Función ALU	ALU Operation
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
Tipo R	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111
		NOR	100111	NOR	1100
		shift left logical	000000	multiplication	1110
addi	11	Add inmediate	xxxxxx	add	0010
	11	Load Upper Inmediate	xxxxxx	lui	0010

Módulo ALU control
Tablas de verdad

Logica Combinacional

ALUOp-Function se toman como entradas del circuito combinacional

ALUOp [1:0]

Function[5:0]

Y la salida que debo controlar va a ser el ALU Operation, que está en 4 bits.

Tengo 2+6 bits = 8 entradas

$2^8 = 256$ Combinaciones

$$ALUOperation_3 = ALUOp_1 \cdot \overline{ALUOp_0} \cdot \overline{funct_5} + ALUOp_1 \cdot \overline{ALUOp_0} \cdot funct_2 \cdot \overline{funct_1}$$

$$ALUOperation_2 = \overline{ALUOp_1} \cdot \overline{ALUOp_0} \cdot \overline{funct_5} + ALUOp_1 \cdot \overline{ALUOp_0} \cdot funct_1 + \overline{ALUOp_1} \cdot ALUOp_0$$

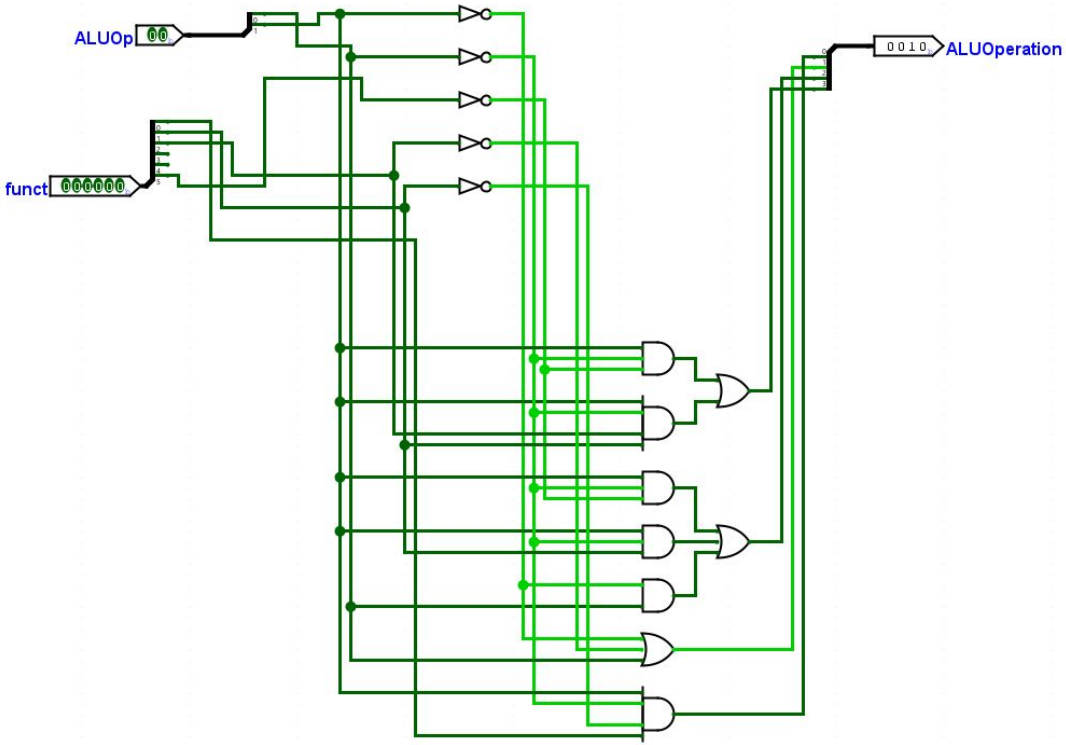
$$ALUOperation_1 = \overline{ALUOp_1} + \overline{funct_2} + ALUOp_0$$

$$ALUOperation_0 = ALUOp_1 \cdot \overline{ALUOp_0} \cdot \overline{funct_1} \cdot funct_0$$

El control de la ALU

El circuito implementado de la ALU, para simular el esquema se separa el ALUOp y el funct y como la instrucción.

La optimización de Logisim excluye 2 variables del funct.



Efectos de las señales de control

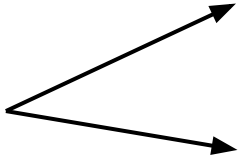
De manera inicial tenemos la activación de 9 líneas de control determinado por el código de operación de las instrucciones

Entrada o salida	Nombre de la señal	Formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Salidas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Unidad de control

Instrucción	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
Formato R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

En el caso de esta unidad de control se requiere agregar más consideraciones



Formato I

Las de salto:

- Jump
- Jump register
- Jump and link

Unidad de control

Instrucción	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
Formato R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1
Formato I	0	1	0	1	0	0	0	1	1

jump = 0

Formato J	x	x	x	0	0	0	0	x	x

jump = 1

Entrada o salida	Nombre de la señal	Formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Salidas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

jump
jr
jal

Form I	J	Jr	Jal
0	0	0	0
0	0	0	0
1	0	0	0
x	0	0	0
x	1	0	1
x	0	1	1
0	0	x	0
1	x	x	x
0	x	x	x
1	0	0	1
0	0	0	0
0	0	0	0
0	0	0	0
1	x	x	x
1	x	x	x
0	1	0	1
0	0	1	0
0	0	0	1

Módulo Unidad control
Tablas de verdad

Logica Combinacional

Opcode se toma como entradas del circuito combinacional

Opcode [5:0]

Tengo 6 bits = 6 entradas

$2^6 = 64$ Combinaciones

La salida que debo controlar en la unidad de control, implica tener las 12 combinaciones en la tabla de verdad.
Además de la salida **Zero** que se usó para la implementación de la instrucción Shift left logical.

Logica Combinacional

$$RegDst = \overline{opcode_5 \cdot opcode_4 \cdot opcode_3 \cdot opcode_1 \cdot opcode_0}$$

$$ALUSrc = opcode_5 \cdot opcode_4 \cdot \overline{opcode_2} \cdot opcode_1 \cdot opcode_0 + opcode_5 \cdot opcode_4 \cdot opcode_3$$

$$MemtoReg = opcode_5 \cdot opcode_4 \cdot \overline{opcode_2} \cdot opcode_1 \cdot opcode_0$$

$$RegWrite = opcode_5 \cdot opcode_4 \cdot \overline{opcode_2} \cdot opcode_1 \cdot opcode_0 + opcode_4 \cdot opcode_3 \cdot opcode_2 \cdot opcode_1 \cdot opcode_0 + opcode_5 \cdot opcode_4 \cdot opcode_3$$

$$MemRead = opcode_5 \cdot \overline{opcode_4} \cdot opcode_3 \cdot opcode_2 \cdot opcode_1 \cdot opcode_0$$

$$MemWrite = opcode_5 \cdot \overline{opcode_4} \cdot opcode_3 \cdot \overline{opcode_2} \cdot opcode_1 \cdot opcode_0$$

$$Branch = opcode_5 \cdot opcode_4 \cdot \overline{opcode_3} \cdot opcode_2 \cdot \overline{opcode_1} \cdot opcode_0$$

$$ALUOp_1 = opcode_5 \cdot opcode_4 \cdot \overline{opcode_2} \cdot opcode_1 \cdot \overline{opcode_0} + opcode_5 \cdot opcode_4 \cdot opcode_3$$

$$ALUOp_0 = opcode_5 \cdot opcode_4 \cdot \overline{opcode_2} \cdot opcode_1 \cdot opcode_0 + opcode_5 \cdot opcode_4 \cdot opcode_3$$

$$jump = \overline{opcode_5} \cdot \overline{opcode_4} \cdot opcode_3 \cdot opcode_2 \cdot \overline{opcode_1}$$

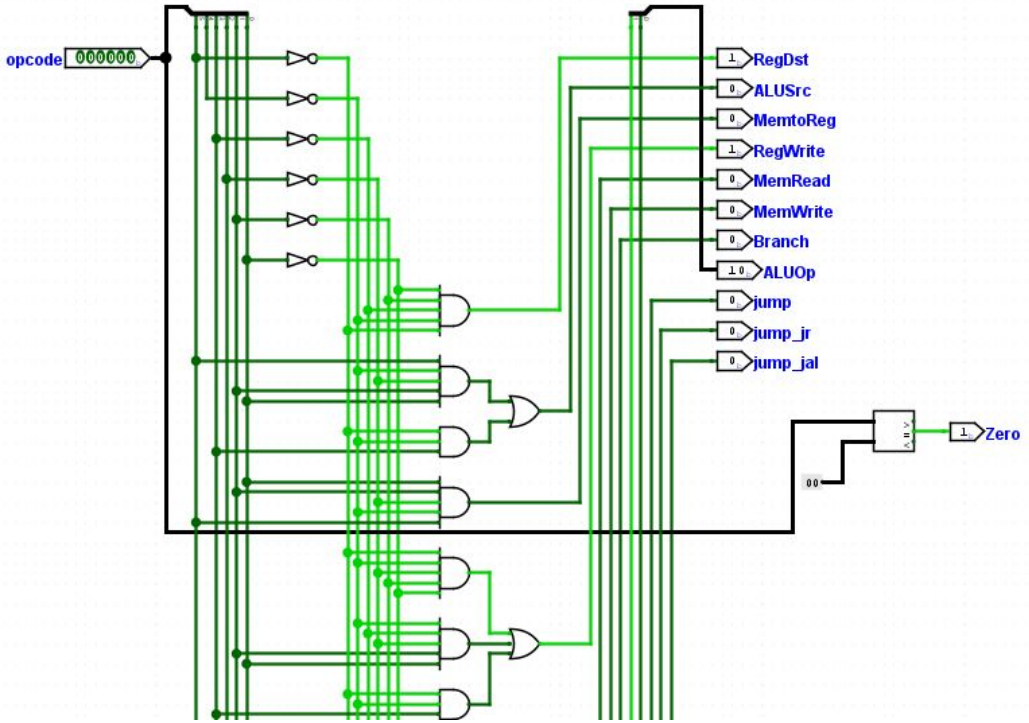
$$jump\_jr = \overline{opcode_5} \cdot \overline{opcode_4} \cdot opcode_3 \cdot \overline{opcode_2} \cdot \overline{opcode_1} \cdot \overline{opcode_0}$$

$$jump\_jal = \overline{opcode_5} \cdot \overline{opcode_4} \cdot opcode_3 \cdot opcode_2 \cdot \overline{opcode_1} \cdot \overline{opcode_0}$$

$$Zero = \overline{opcode_5 \cdot opcode_4 \cdot opcode_3 \cdot opcode_2 \cdot opcode_1 \cdot opcode_0}$$

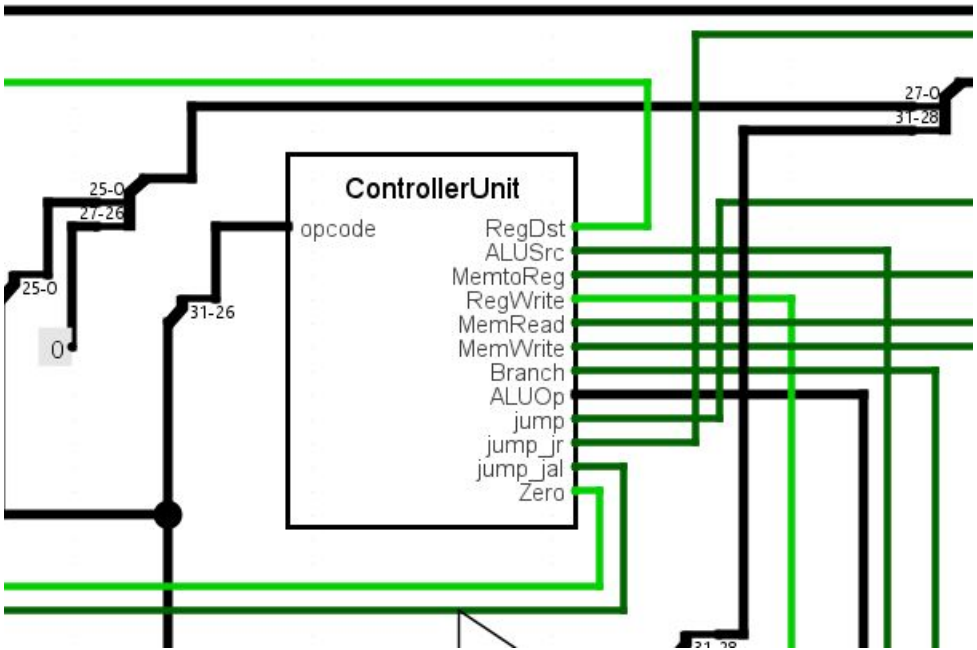
Unidad de control

El circuito implementado de la unidad de control, en este caso debemos tener en cuenta que ALUOp tiene 2 bits y que Zero es salida para sll.



Unidad de control

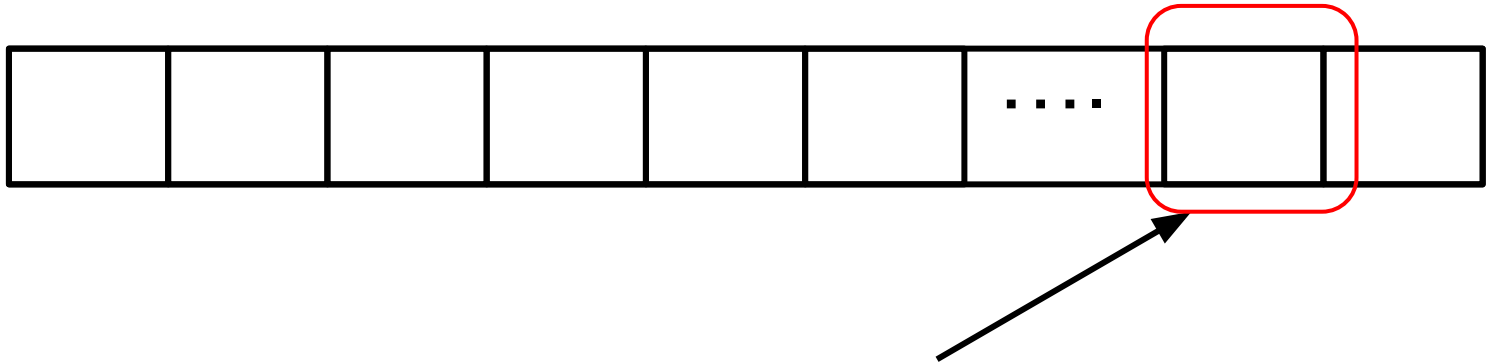
Esta es la implementación en Logisim integrada con el resto del procesador



Ejecución de una programa MIPS

UNIVERSIDAD DE ANTIOQUIA
Facultad de Medicina

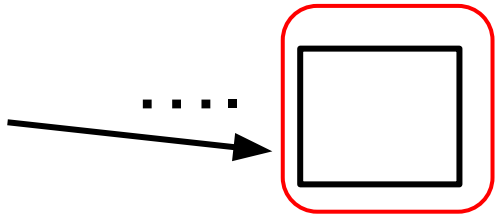
La operación 80x, x es el penúltimo elemento del vector.



Nos interesa hallar ese valor ubicado en la penúltima posición del vector

La operación 80x, x es el penúltimo elemento del vector.

Luego de tener el valor de esa posición, queremos realizar el producto de 80 veces ese número

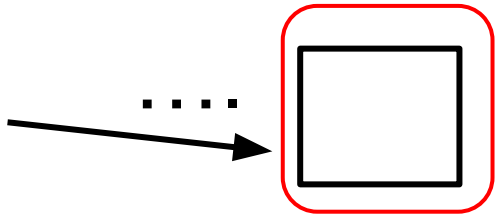


$$80x = 64x + 16x$$

$$64 = 2^6 \qquad 16 = 2^4$$

La operación 80x, x es el penúltimo elemento del vector.

Luego de tener el valor de esa posición, queremos realizar el producto de 80 veces ese número



$$80x = 64x + 16x$$

$$64 = 2^6 \qquad 16 = 2^4$$

Implementación algoritmo

En este caso nuestro algoritmo está compuesto por una sección principal (main) que hace el llamado a 3 funciones.

$$\text{main}()$$

$$\text{longitudArreglo}()$$

$$\text{valorPenultimo}()$$

$$\text{operaciónMultiplication}()$$

Memoria de datos y rutina Main

Declaramos un vector aleatorio que ejemplifica el funcionamiento del algoritmo. Podemos ver en el código el uso de 3 etiquetas **jal** para el llamado a funciones

```
.data
    vector: .word 5,1,2,6,5,5,6,7,9,10,11 #arr[]

.text
main:
    lui $s1, 0x1001 #direccion base del vector
    jal longitud_array #CalcularLongitudArray()
    lui $a0, 0x1001 #direccion base del vector en
a0
    addi $a1, $s0, 0 #a1=longitud
    jal penultimo #Obtener penultimo elemento
    addi $a1, $v0, 0 #arg1 = valor penult y=x
    jal mult_function # function_multiplicacion()
    j fin_ejecucion no seguir ejecutando
```

Cálculo de Longitud

Considere que al aprovechar que todos los registros de MARS y Logisim empiezan en cero, se usa esta condición como la condición de parada

```
longitud_array:
    lw  $a0, 0($s1) #a0=s1[i]
    beq $a0, 0, exit #if(i=0)exit
    addi $s0, $s0, 1 # longitud=longitud +1
    addi $s1, $s1, 4 #dirección=dirección+4
    j longitud_array

exit:
    jr $ra
```

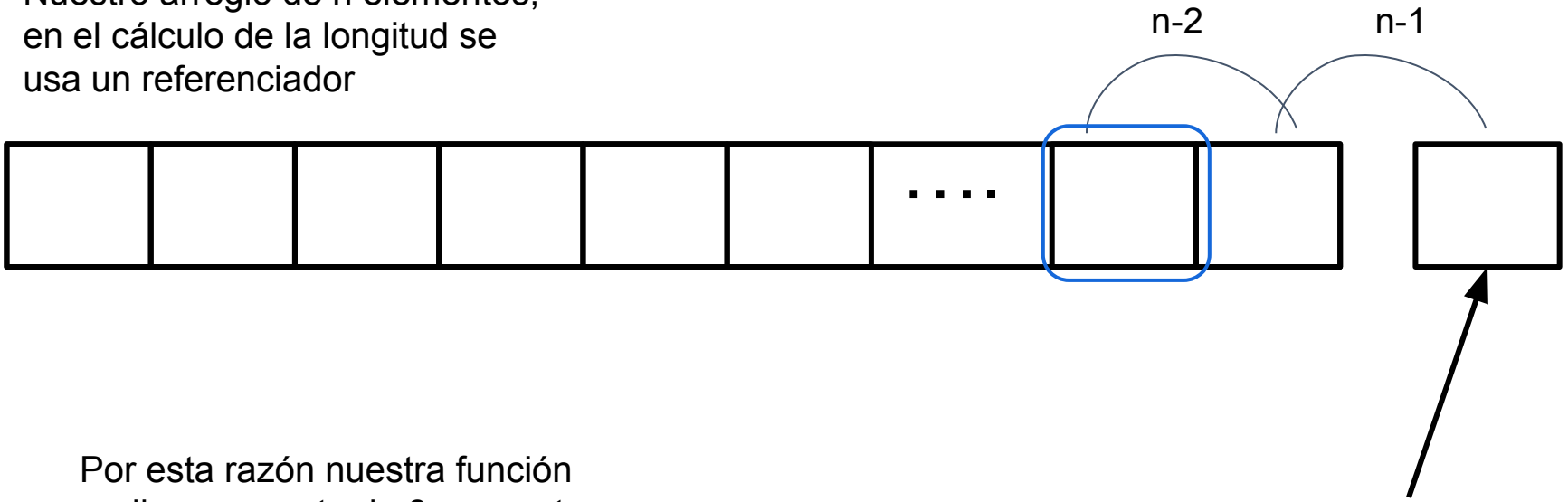
Entonces el elemento vector[i] = 0, es el marcador de finalización, cuando se encuentre un cero se va a detener la función

Valor de la penúltima posición

En este caso para evitar el uso de más argumentos y registros, usamos el referenciador de las direcciones, para calcular la posición del penultimo

```
penultimo:
    addi $t2, $zero, 8   # Cargar 8 en $t2
    sub $t1, $t1, $t2    # Restar 8 de $t1
    add $s1, $s1, $t1 # dirección del penúltimo elemento
    lw $v0,0($s1) #cargamos en v0 el valor del penúltimo
elemento
    jr $ra
```

Nuestro arreglo de n elementos, en el cálculo de la longitud se usa un referenciador



Por esta razón nuestra función realiza una resta de 8, con esto nos devolvemos 2 posiciones de la referencia del marcador, y nos situamos en el penúltimo.

Este es el registro final, situado después del último elemento, **marcador de finalización**

Calculo valor 80x

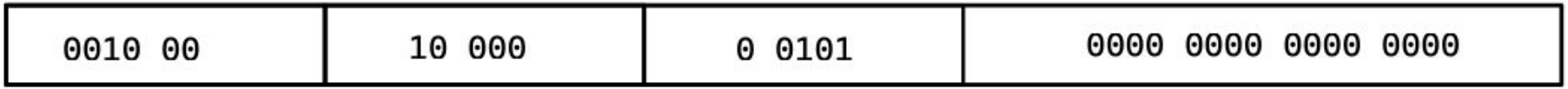
Como ya se había explicado usamos la instrucción **shift left logical,** para realizar los desplazamientos y guardar el número.

```
mult_function:
    sll $t3, $a1, 6 #x * 64
    sll $t4, $a1, 4 #x * 16
    add $v1, $t3, $t4 #(x*64) + (x*16) = x*80
    jr $ra
```

Ensamble de instrucciones

```
addi $a1, $s0, 0 #a1=longitud
```

Tipo i



op	RS	RT	Constant
6 bits	5 bits	5 bits	16 bits

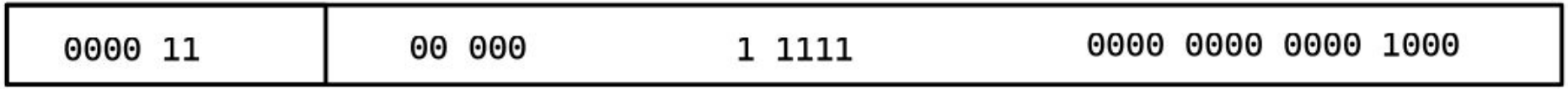
0x22050000 Instruccion hexadecimal

Ejemplificación Instrucciones tipo I

Ensamble de instrucciones

```
jal longitud_array #CalcularLongitudArray()
```

Tipo j



op | RS | RT | Constant

6 bits | 26 bits

0x0C1F0008 Instruccion hexadecimal

Ejemplificación Instrucciones tipo J

Ensamble de instrucciones

```
add $s2, $s2, $t1 ## direccion del penultimo
```

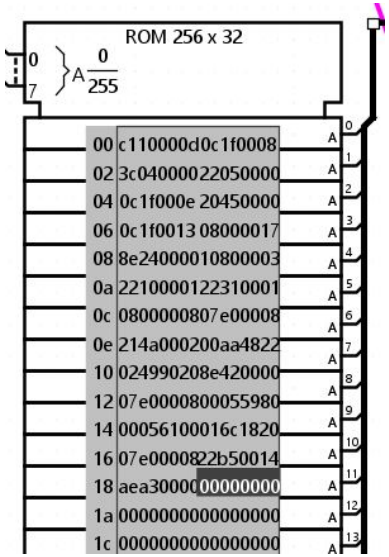
Tipo r

0000 00	10 010	0 1001	1001 0	000 00	10 000
op	RS	RT	RD	SHAMT	FUNCTION
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

0x02499020 Instruccion hexadecimal

Ejemplificación Instrucciones tipo R

Generalización



00	c110000d	0c1f0008	3c040000	22050000	0c1f000e	20450000	0c1f0013	08000017
08	8e240000	10800003	22100001	22310001	08000008	07e00008	214a0002	00aa4822
10	02499020	8e420000	07e00008	00055980	00056100	016c1820	07e00008	22b50014
18	aea30000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
28	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
38	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
58	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Se realiza el procedimiento mostrado para todo el resto de instrucciones, y logrando así ensamblar todo el algoritmo y mapearlo en la memoria de instrucciones

Anexos

Enlace video YouTube:

Enlace video

Enlace repositorio:

Enlace repositorio