**Laboratorio No. 3**

**Diseño de procesador monociclo**

**Autores:**

**Omar Alberto Torres**

**Leidy Castaño Castaño**

**Profesor:**

**Fredy Alexander Rivera Velez**

**Arquitectura de Computadores y Laboratorio**

**Universidad de Antioquia**

**Facultad de Ingeniería**

**Ingeniería de Sistemas**

**Ude@**

**2023**

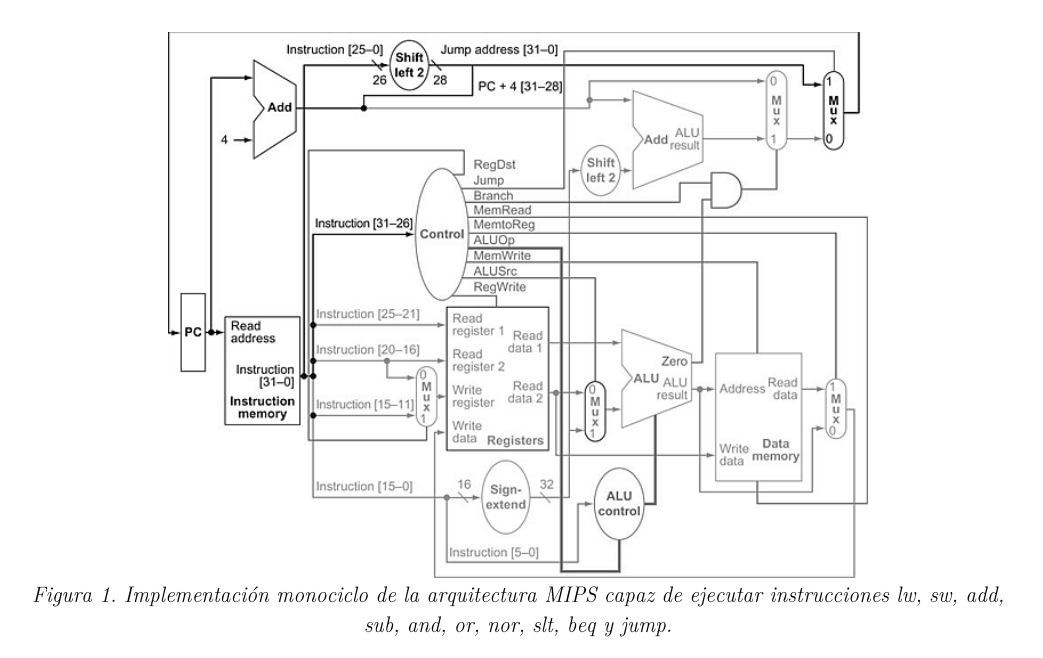
**Laboratorio No. 3**

**Link del video:**

[**https://udeaeduco-my.sharepoint.com/personal/leidy\_castanoc\_udea\_edu\_co/\_layouts/15/stream.aspx?id=%2Fpersonal%2Fleidy%5Fcastanoc%5Fudea%5Fedu%5Fco%2FDocuments%2FUDEA%2F2023%2D1%2FArquitectura%20de%20Computadores%2FLab3%5Fvideo%5FArquitectura%2Emp4&ga=1**](https://udeaeduco-my.sharepoint.com/personal/leidy_castanoc_udea_edu_co/_layouts/15/stream.aspx?id=%2Fpersonal%2Fleidy%5Fcastanoc%5Fudea%5Fedu%5Fco%2FDocuments%2FUDEA%2F2023%2D1%2FArquitectura%20de%20Computadores%2FLab3%5Fvideo%5FArquitectura%2Emp4&ga=1)

**Descripción**

En esta práctica cada equipo realizará una implementación como procesador monociclo de una versión reducida de la arquitectura MIPS32 en la herramienta Logisim Evolution. Partiendo de la implementación básica mostrada en la Figura 1, cada equipo de trabajo deberá extender el repertorio de instrucciones soportadas en hardware. Una vez implementadas la ruta de datos y la unidad de control, el procesador será puesto a prueba para verificar el correcto funcionamiento de todas las instrucciones incluidas, y usado para resolver un problema en particular asignado a cada equipo de trabajo.



**Desarrollo**

**Diseño Banco de Registros:**

Se implementó un banco de 32 registros, con las siguientes características:

* **Zero** con número 0, su valor es constante, puede ser accedido, pero no puede ser modificado.
* Los registros [**to** hasta **t7**] con valores [8 a 15 y 24,25] pueden ser leídos y escritos.
* Los registros [**so** hasta **s7**] con valores [16 hasta 23], pueden ser leídos y escritos
* Los registros [**v0**, **v1**], con valores [2,3] pueden ser leídos y escritos, usado para retornar resultados de funciones.
* Los registros [**a0** hasta **a3**] con valores [4 hasta 7] pueden ser leídos y escritos, usado para el paso de parámetros.
* El registro **at** con valor 1, es un ensamblador temporal.
* Los registros **k0**, **k1** con valores[26, 27], son usados para el núcleo del sistema operativo.
* El registro **gp** con valor 28, puntero global
* El registro **sp** con valor 29, es el apuntador de pila.
* El registro **fp** con valor 30, es el puntero marco.
* El registro **ra** con valor 31, usado para el retorno.
* La referencia a que tiene un valor es para indicar el número del registro dentro del banco

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

**Acceso sincrónico (Escritura de registros en cada pulso de reloj)**

Para escribir en un registro se requiere que en el puerto write\_data exista un valor, previamente se escribe en el registro de 5 bits Write\_reg un número que corresponde al registro del banco que se quiere acceder, el decodificador toma el valor presente en Write\_reg y selecciona al registro deseado, dadas estas condiciones al llegar un flanco de subida de reloj el valor write\_data se escribe en el registro.

**Acceso asíncrono**

Los registros pueden ser leídos de manera asíncrona desde dos puertos de salida diferentes (Read\_Data1, Read\_Data2), los multiplexores (Read\_Reg1, Read\_Reg2) transfiere el contenido de los registros que hayan sido seleccionados en el respectivo puerto de salida.

**Decisiones de diseño**

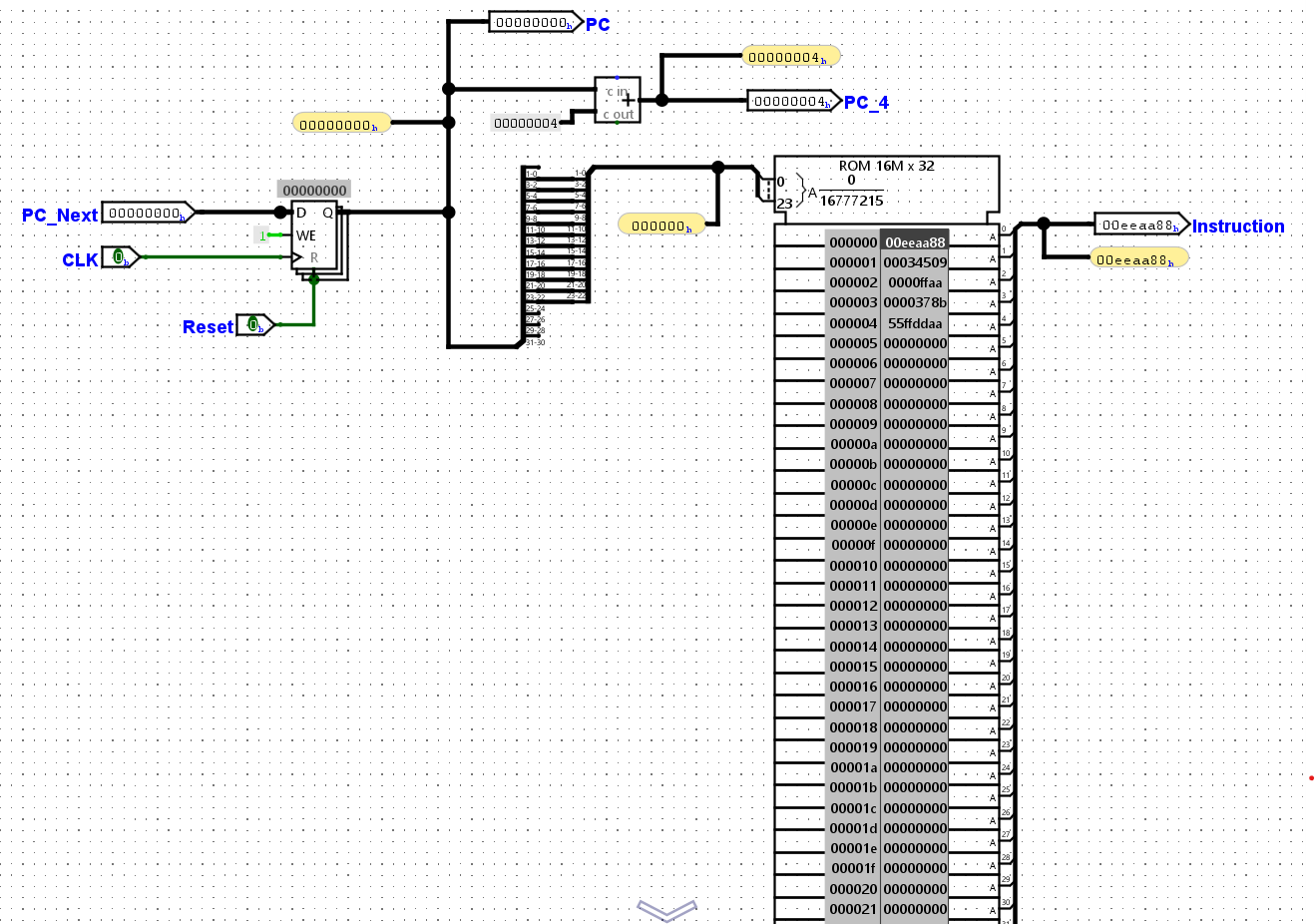
La decisión de diseño para el desarrollo de este módulo ha sido la de observar estrictamente la información técnica, que se encuentra en la tabla de datos de referencia de la arquitectura MIPS, los nombres de los registros son similares a los que ha establecido el fabricante del chip MIPS, por ejemplo para los registros $t0, $t1,..... nosotros adoptamos los nombres t0,t1,-------.

Para efecto de tener un diseño que ofrezca facilidad de entender se ubicó el banco de registros en un modulo llamado BANCO\_REG, este módulo ofrece pines de entrada y salida para las señales, [Reg\_Write, Write\_Reg, Read\_Reg1, Read\_Reg2, Read\_Data 1, Read\_Data2]]

**Circuitos etapa FETCH**

La circuitería de esta etapa tiene como objetivo poder acceder durante los flancos de subida de reloj a la memoria de instrucciones, de tal forma que en un ciclo de reloj se saca una instrucción de la memoria del programa para dejar la instrucción disponible y pueda ser utilizada por los circuitos que forman la etapa de decodificación y acceso al banco de registros.

Cuando el sistema inicia la operación, el PC tendrá la dirección de la primera instrucción a ser ejecutada, a la llegada de un pulso de reloj y durante el flanco de subida la dirección se coloca en el bus, y estando la memoria activada para ser leída, la instrucción correspondiente se coloca en el bus, esta instrucción en un ciclo posterior pasará a la etapa de decodificación. De otro lado la dirección en proceso, entra a un circuito sumador que le adiciona el valor 4, generándose el PC + 4, que corresponde a la dirección donde se ubica la siguiente instrucción a ser leída.



**Decisiones de diseño**

Se integra en el módulo Fetch la memoria de programa y el proceso del contador de programa. Por limitaciones del sistema se utilizan 24 bits para el direccionamiento de la memoria del programa. El módulo deja los pines disponibles de PC\_Next, CLK, Reset, PC\_4 y Instruction con el objeto de poder integrarlos al sistema general.

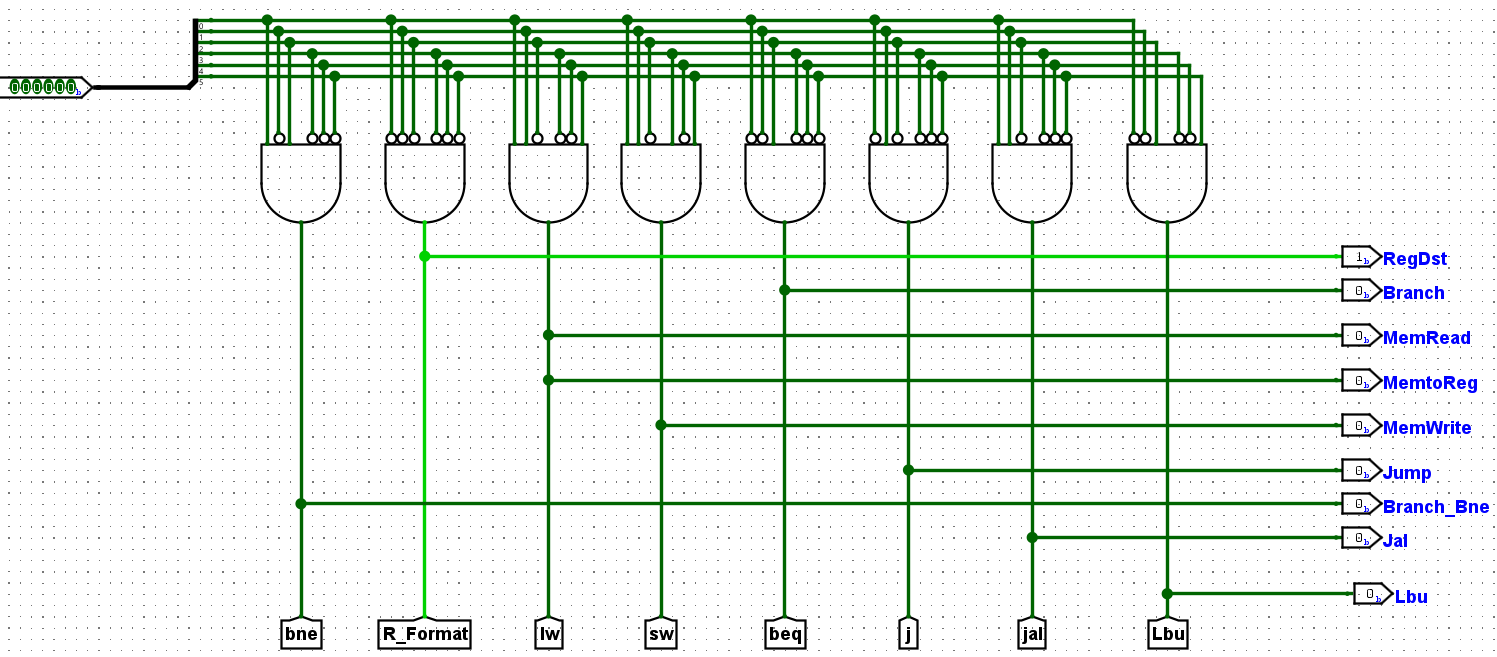
**Unidad de control**

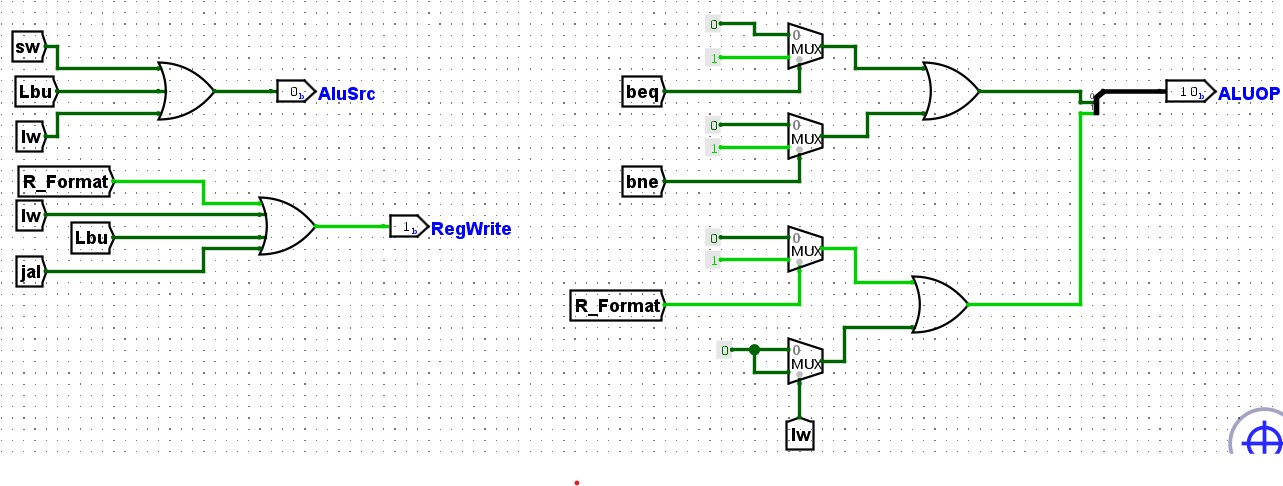
La unidad de control es la responsable de generar señales eléctricas que sincronizan y facilitan la realización de las tareas de acceso al banco de registros, acceso a la memoria de datos, acceso a la memoria de datos, operación de la unidad aritmética y lógica.

Para diseñar esta etapa se utilizó la siguiente tabla de códigos.

| Signal name | R format | lw | sw | beq | bne | j | jal | lbu |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| op5 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| op4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| op3 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| op2 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| op2 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| op0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |
| RegDst | 1 | 0 | x | x | x | x | x | 0 |
| AluSrc | 0 | 1 | 1 | 0 | x | x | x | 1 |
| MentoReg | 0 | 1 | x | 0 | x | x | x | 1 |
| RegWrite | 1 | 1 | 0 | x | x | x | x | 1 |
| MenRead | 0 | 1 | 0 | 0 | x | x | x | 1 |
| MenWrite | 0 | 0 | 1 | 0 | x | x | x | 0 |
| Branch | 0 | 0 | 0 | 1 | x | 1 | x | 0 |
| AluOp1 | 1 | 0 | 0 | 0 | 1 | x | x | 0 |
| AluOp2 | 0 | 0 | 0 | 1 | x | x | x | 0 |
|  |  |  |  |  |  |  |  |  |

**Lógica combinacional de la unidad de control.**





Como se observa en las gráficas anteriores la unidad de control está implementada mediante lógica combinacional.

Supongamos que se está en la etapa de decodificación de la instrucción: lw $t0, offset($s0). Esto significa que se ejecuta una función de lectura de la memoria de datos, el $s0 apunta a la dirección base donde se ubica el dato (PC + offset).

Para que esto sea posible los circuitos decodificadores obtienen de la instrucción el código de operación, (23H) que en binario es codop = 100011. La lógica combinacional genera los valores siguientes:

RegDst = 0, AluSrc=1, MentoReg = 1, RegWrite = 1, MemRead. Memwrite = 0, Branch =0, AluOp1 =0, AluOp2=0.

AluSrc permite que pase a la ALU el contenido de Read\_data2 o el valor que de 32 bits proveniente del circuito extensor de signo, con este valor la ALU permite que el valor leído de la memoria pase hacia Write\_data (Registro destino), de otro lado el valor Writeread habilita la lectura de la memoria y el valor RegWrite, habilita escribir en el registro destino.

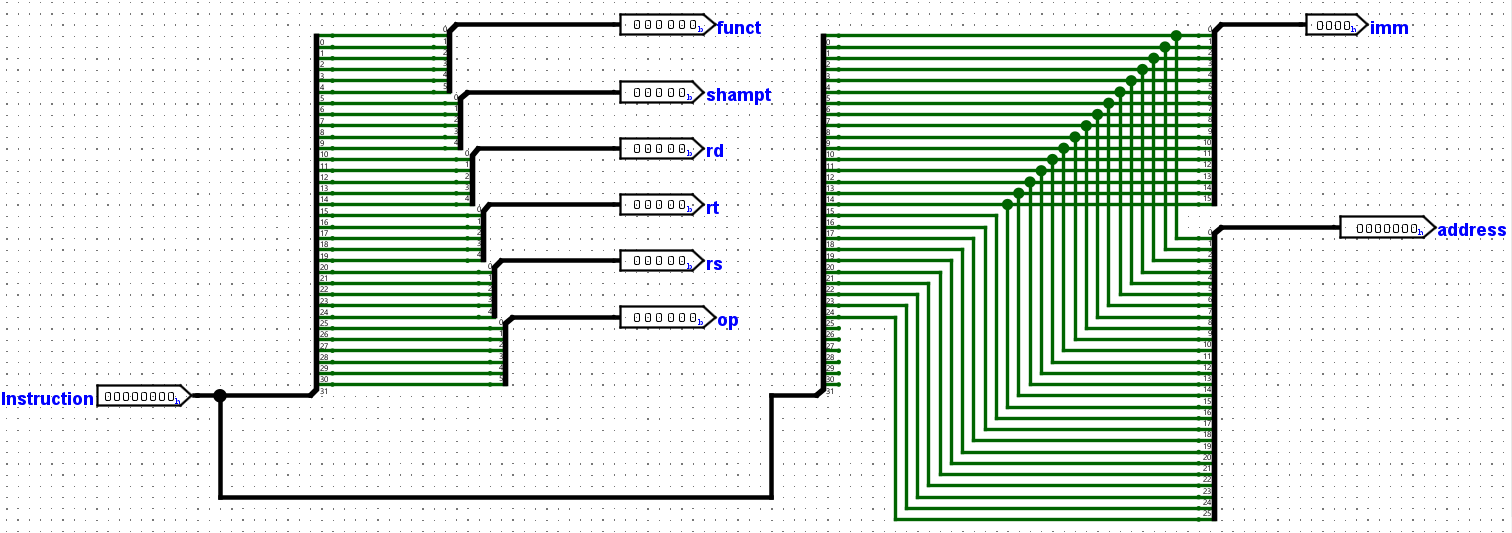
**Decisiones de diseño**

Para el diseño de la unidad de control se hace uso de la tabla de verdad, descrita en el documento 20231-ACtyLab-C23.pdf, y se agrega las columnas Lbu, que corresponden con las instrucciones que se nos solicita adicionar a este sistema.

No fué necesario implementar mapas de Karnaugh dado que cada señal se implemente con una compuerta lógica and, negando sus entradas según sea el código de operación tal que se garantiza un 1 cuando la señal se activa.

**Módulo LBD\_DOWN**

Este módulo es el responsable de separar cada instrucción para poder hacer la decodificación correcta de la misma.



La información generada es:

* Funct: b5………b0.
* rs: b25………….b21
* rt: b20………….b16
* rd: b15………….b11
* shamt: b10…….b5
* address: b0…….b25

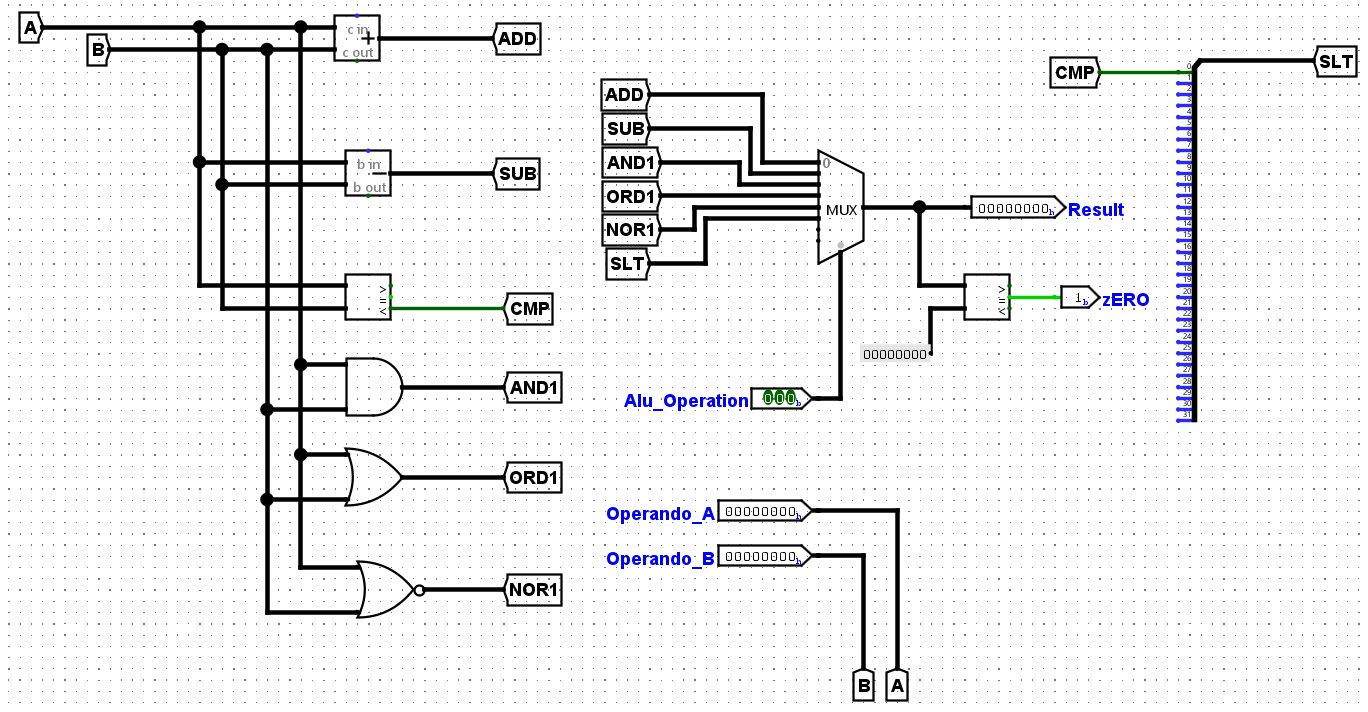
Cada uno de estos segmentos de la instrucción (Representada en bits) es usada para generar señales de control, acceso al banco de registros y activación de la ALU Control.

**Consideraciones de diseño.**

No fué necesario implementar una lógica para fragmentar el código de la instrucción para su posterior decodificación dado que se hizo uso de los módulos splitter de logisim.

**Unidad Aritmética y Lógica.**

La unidad aritmética y lógica es la responsable de realizar operaciones matemáticas y lógicas.

****

Para este trabajo sólo se implementó un número limitado de operaciones.

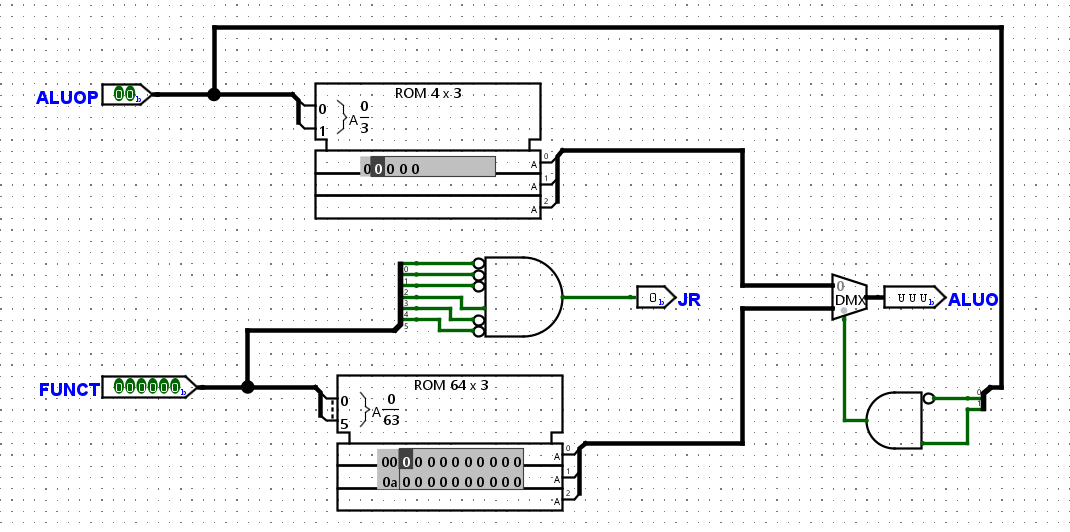
* Suma
* Resta
* Operación And
* Operación Or
* Operación Nor
* Comparación
* Operación set, SLT

**Decisiones de diseño.**

El diseño de esta etapa requiere de la entrada de los operandos A y B respectivamente, y un código de tres bits (que viene de la ALU\_CONROL ) para indicarle a la CPU que operación es requerida (Suma, resta, comparación SLT, AND, or, nor).

**ALU CONTROL**

Este módulo es auxiliar y paralelo a la unidad de control, que nos permite tener una unidad de control razonablemente más pequeña.



Esta unidad recibe como entrada las señales ALUOP, Function, generando un código de tres bits que permite (**ALUO**), permitiendo a la ALU saber qué operación debe ejecutar. De otro lado decodifica la entrada FUNCT para obtener la señal JR (Salto, por retorno (jr $ra) desde una función o procedimiento).

**Tablas de codificación**

| **opcode** | **ALUOP** | **Operation** | **funct** | **Función ALU** | **ALUO** |
| --- | --- | --- | --- | --- | --- |
| **lw** | **00** | **load word** | **xx** | **add** | **000** |
| **lbu** | **00** | **load byte** | **xx** | **add** | **000** |
| **sw** | **00** | **store word** | **xx** | **add** | **000** |
| **beq** | **01** | **salto if igual** | **xx** | **sub** | **001** |
| **bne** | **01** | **salto si diferente** | **xx** | **sub** | **001** |
| **TIPO R** | **10** | **add** | **20h** | **add** | **000** |
|  | **10** | **sub** | **22h** | **sub** | **001** |
|  | **10** | **and** | **24h** | **and** | **010** |
|  | **10** | **or** | **25h** | **or** | **011** |
|  | **10** | **nor** | **27h** | **nor** | **100** |
|  | **10** | **slt** | **2Ah** | **set** | **101** |

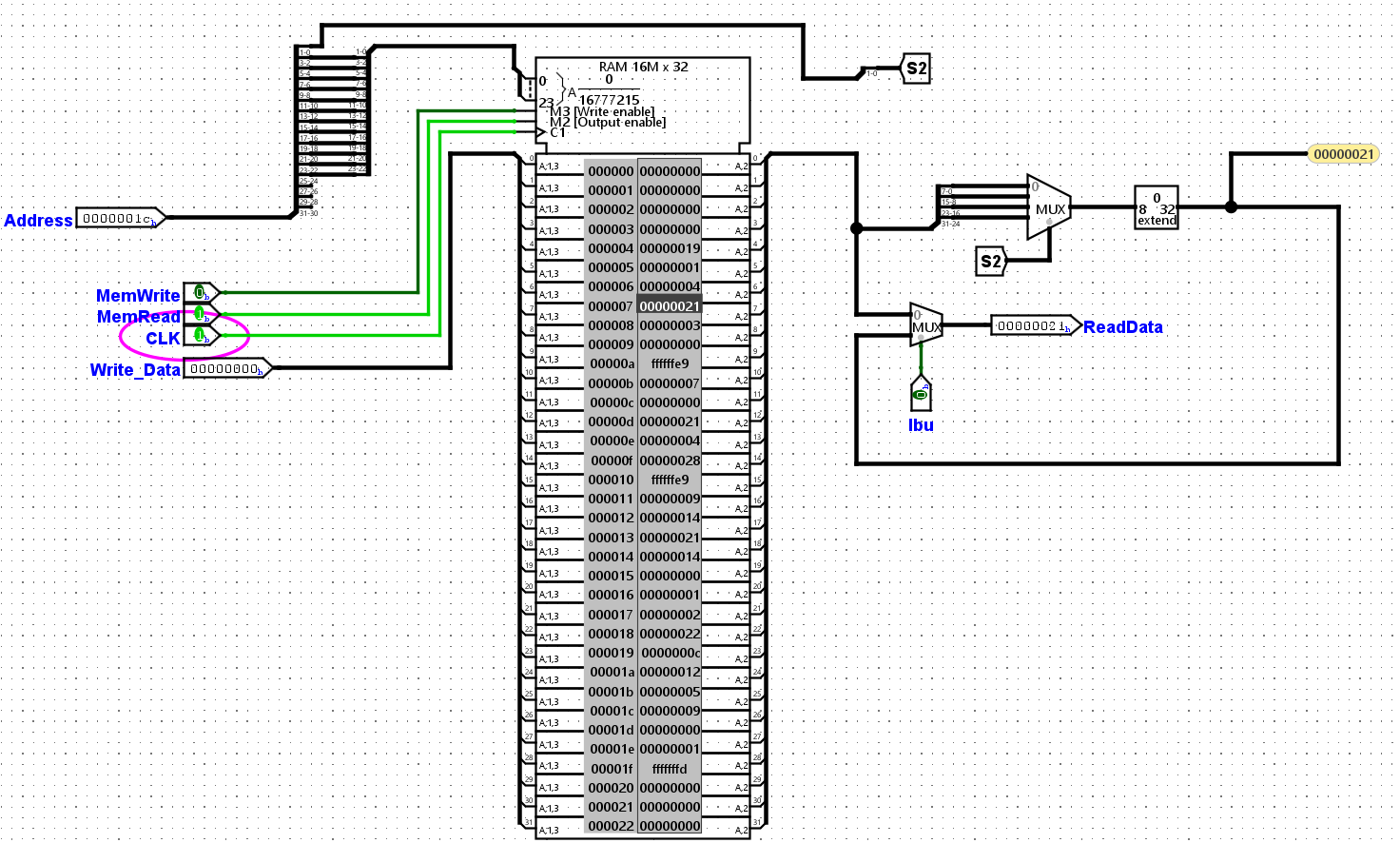
**Decisiones de diseño**

Para efectos de tener un código más simple los codigos relacionados con ALUOperatión se almacenan unidades de memoria rom, cuando llega la señal ALUOP y con la ayuda de las compuertas lógicas se decide si lo que debe ejecutar la **ALU** es una operación correspondiente al formato R o instrucciones de lectura o escritura de memoria, igualmente se decodifica la señal Jr, para poder efectuar el retorno desde la finalización de una función o procedimiento.

**Módulo de memoria RAM**

Es la memoria de trabajo para la cpu, en ella se almacena temporalmente información fundamental durante la ejecución de un programa como puede ser: lista de datos, arreglos , imágenes, cualquier tipo de información en formato numérico. La memoria de datos para este trabajo almacena 256 palabras**.**

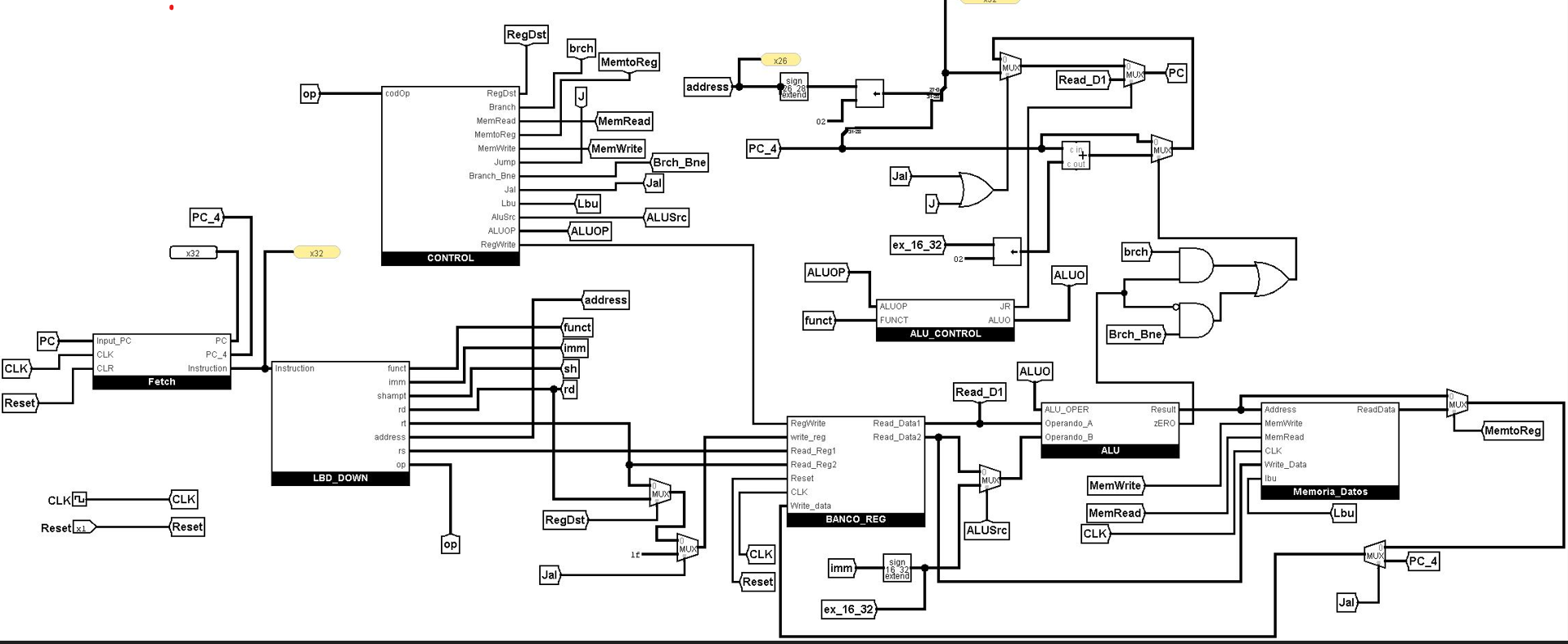
En la figura se puede observar la memoria ram cargada con el segmento de datos. al introducir la dirección 32 (0x00000010), sale al bus de datos el valor 33 (0x00000021) que corresponde al valor 33 del primer elemento del arreglo.

****

**Decisiones de diseño.**

La decisión fué la de incluir en este módulo la lógica necesaria para que retorne un byte o una palabra, tal como se muestra en la figura del circuito anterior y adaptar el direccionamiento a la cantidad de memoria disponible en el simulador.

**Figura Procesador Monociclo.**

****

**Operación general**

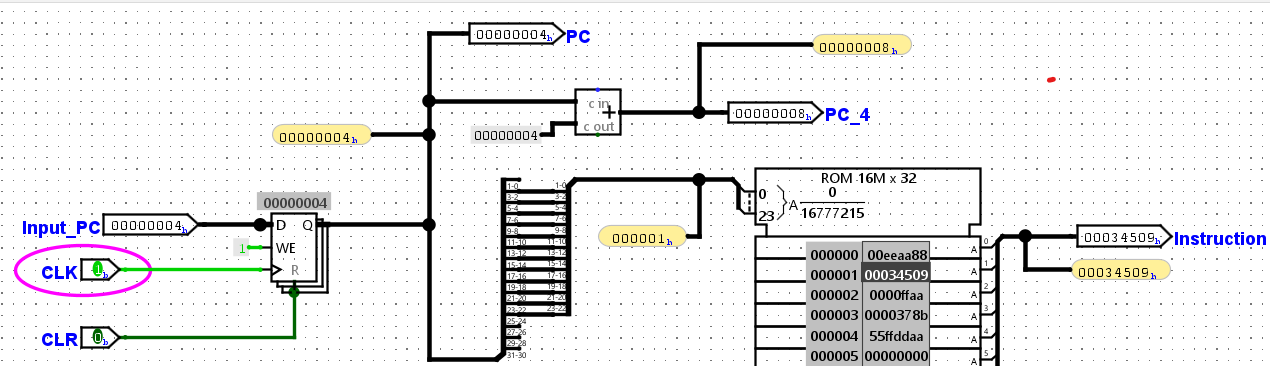
**Módulo Fetch**

En la figura del **procesador monociclo**, se observa la conexión de los módulos de búsqueda Fetch (Responsable de hacer la búsqueda de una instrucción en la memoria del programa) y el módulo que hemos llamado LBD DOWN (Responsable de tomar la instrucción de programa leída en el módulo Fetch, para obtener las partes que la componen (Código de operación, registros rs, rt, rd, funct, address, imm) según sea el caso es decir si se trata de una instrucción con formato R, I, J.

Igualmente se puede apreciar cada una de las señales generadas por el módulo de control fundamentales, para la sincronización de todas las operaciones del procesador como: lectura/escritura de memoria, operaciones aritméticas y lógicas, saltos condicionales y no condicionales acceso al banco de registros, llamado a funciones y retornos entre otras.

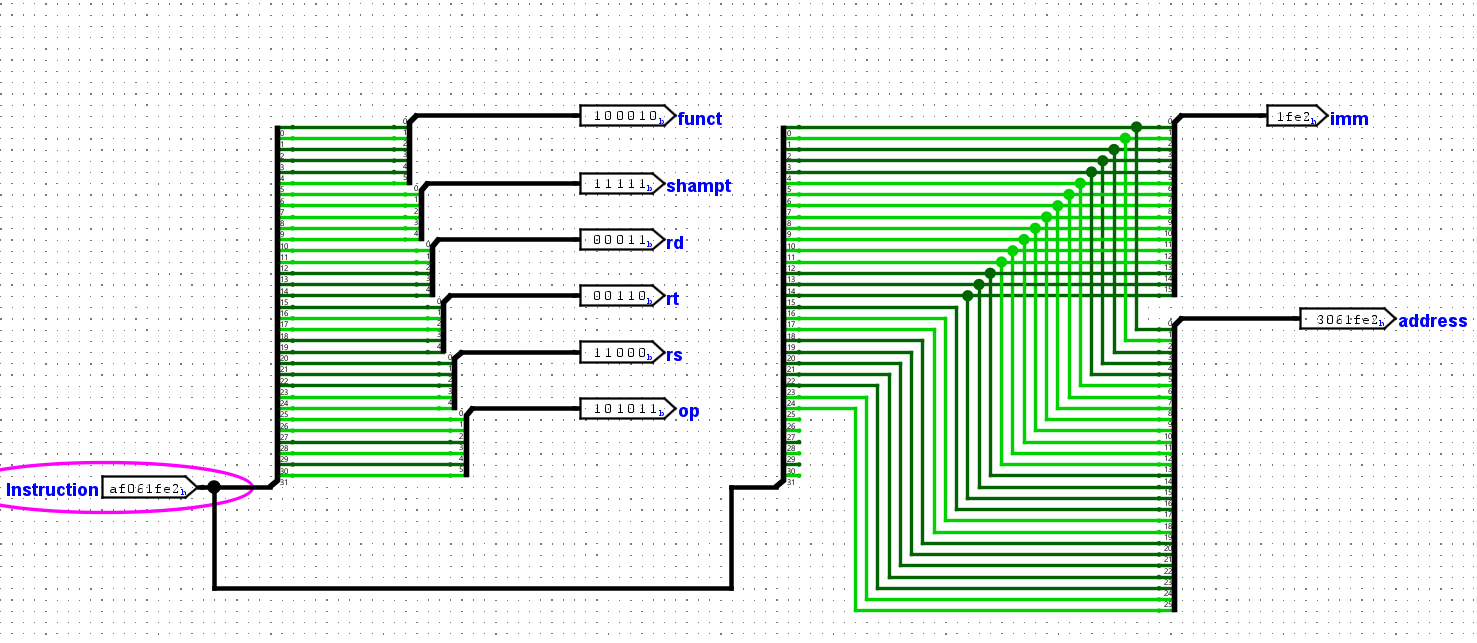
**Evidencia de operación del módulo Fetch**

Asumiendo que el contador de programa tiene la dirección 0x00000004, al llegar un pulso de reloj durante el flanco de subida el valor 0x00034509 queda disponible en el campo Instruction esto es, asumiendo que el valor anterior representa una instrucción, este valor quedará disponible para ser posteriormente decodificado.



**Evidencia de operación del módulo LBD\_DOWN**

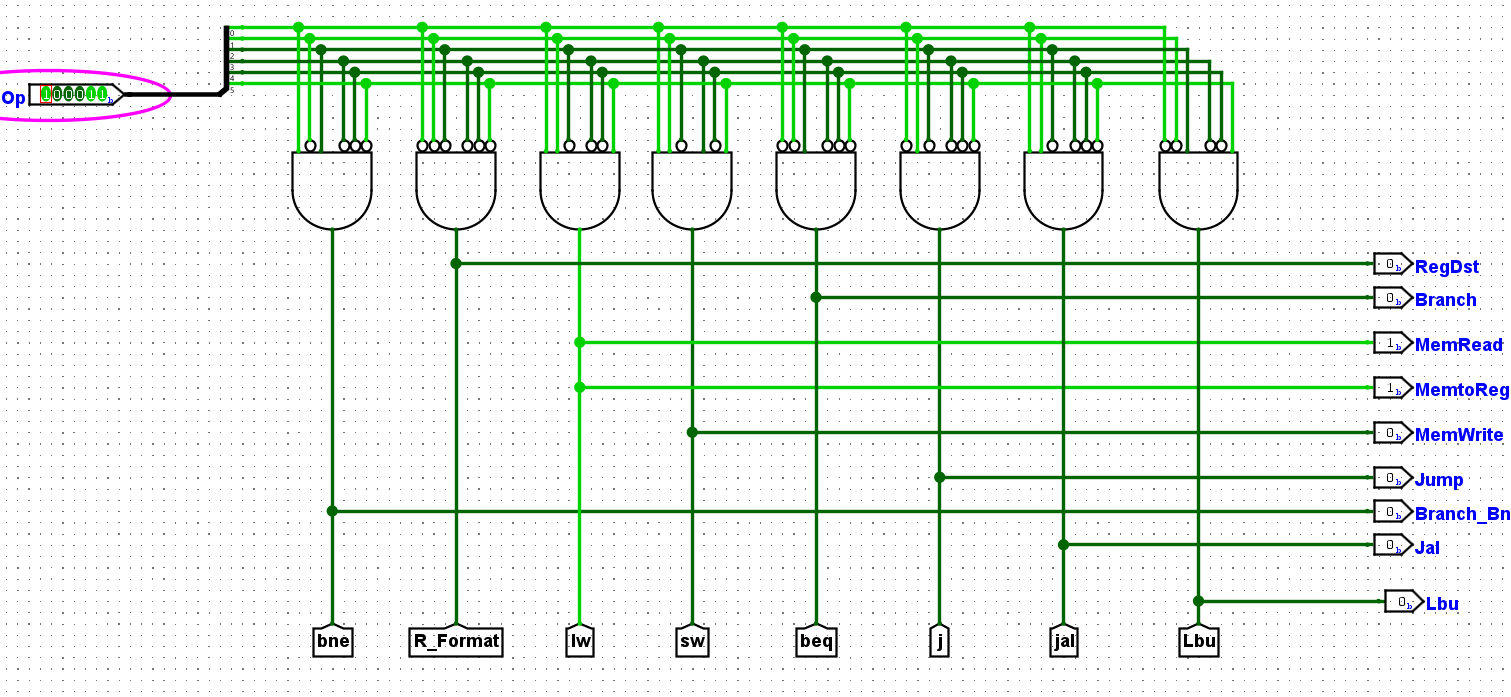
Como se puede apreciar en la figura del **procesador monociclo**, este módulo se conecta con el módulo Fetch, la unidad de control, y el banco de registros.

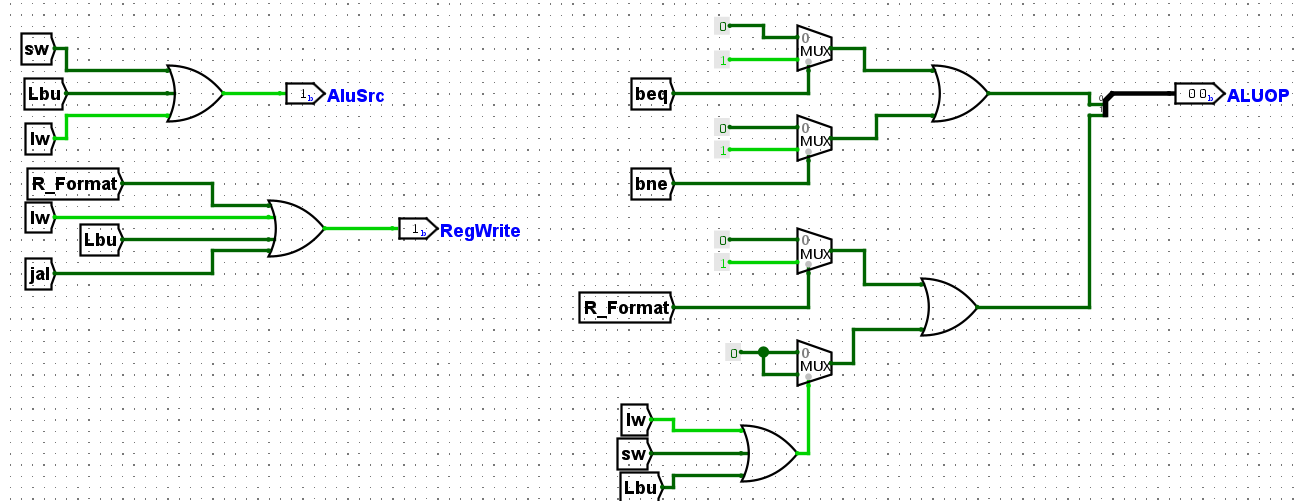
****

Com se puede apreciar en la figura anterior cuando un valor de entrada (correspondiente al código de una instrucción), el valor es fragmentado, para obtener los valores correspondientes al código de operación, el valor de función, los valores de los registros que se quieren acceder, valores de dirección y valores extendidos, valor shamt. Esto es fundamental para el proceso de decodificación de la instrucción.

**Evidencia de operación de la unidad de control.**

Esta unidad es muy importante, es como la parte inteligente del sistema, ella genera todas las señales de control necesarias para sincronizar todas las operaciones de la máquina, tales como operaciones aritméticas, lógicas y acceso a memoria.

****

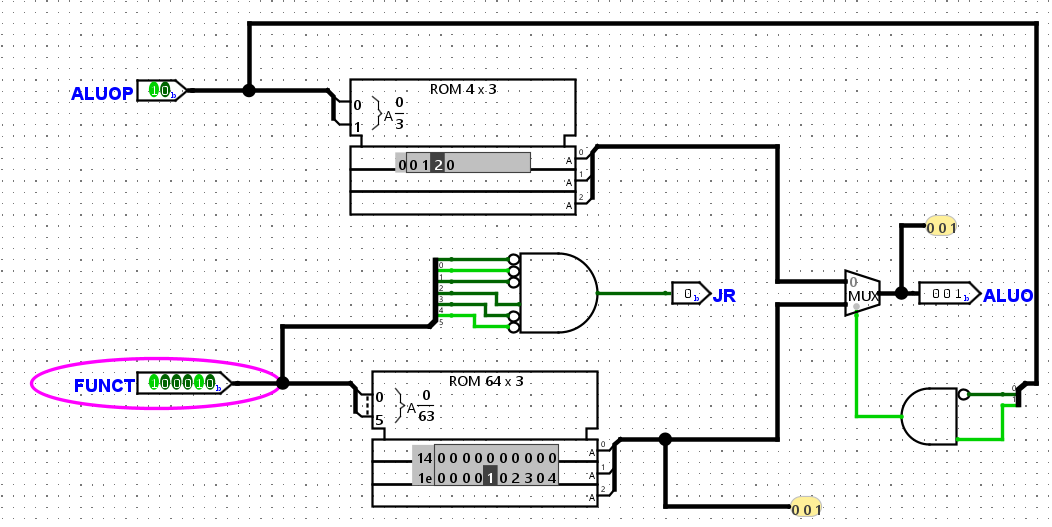
****

Suponiendo que se ejecutó una instrucción de carga cómo lw, vemos en la figura anterior que se activaron las señales lw, MemRead, MentoReg, RegWrite, AluSec.

Siendo esta instrucción muy compleja, dado que implica acceder a registros, acceder a un dato de memoria, escribir en un registro, calcular direcciones de acceso a la memoria.

**Evidencia de operación ALU\_Control**

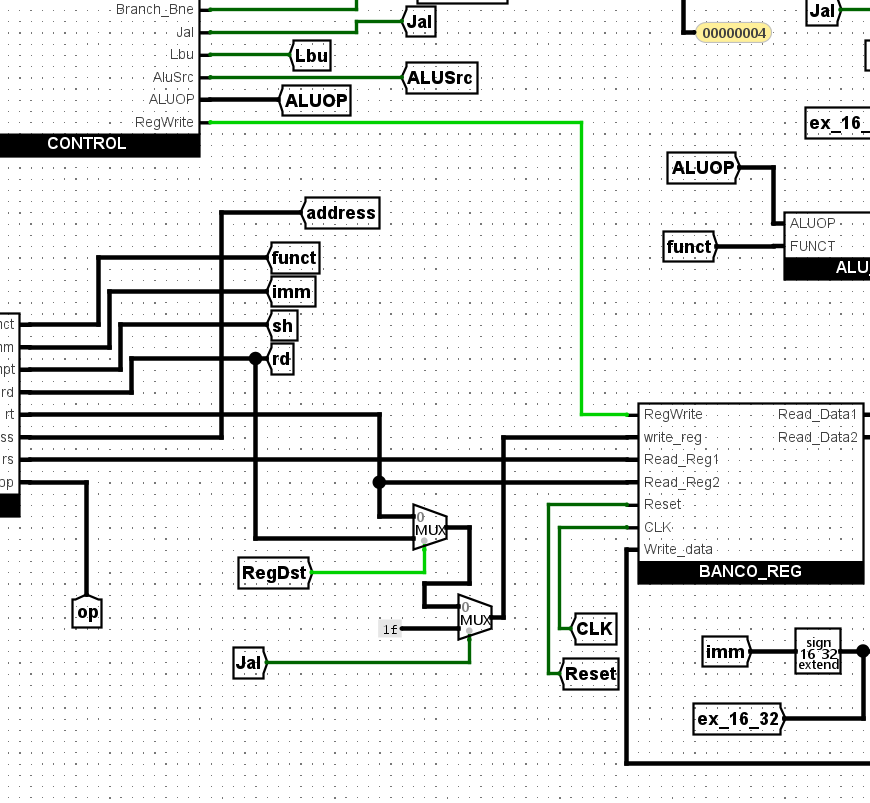
En la figura del **procesador monociclo** podemos apreciar la conexión entre los módulos del banco de registros, la ALU, unidad de control, la ALU CONTROL, y el módulo LBD\_DOWN.



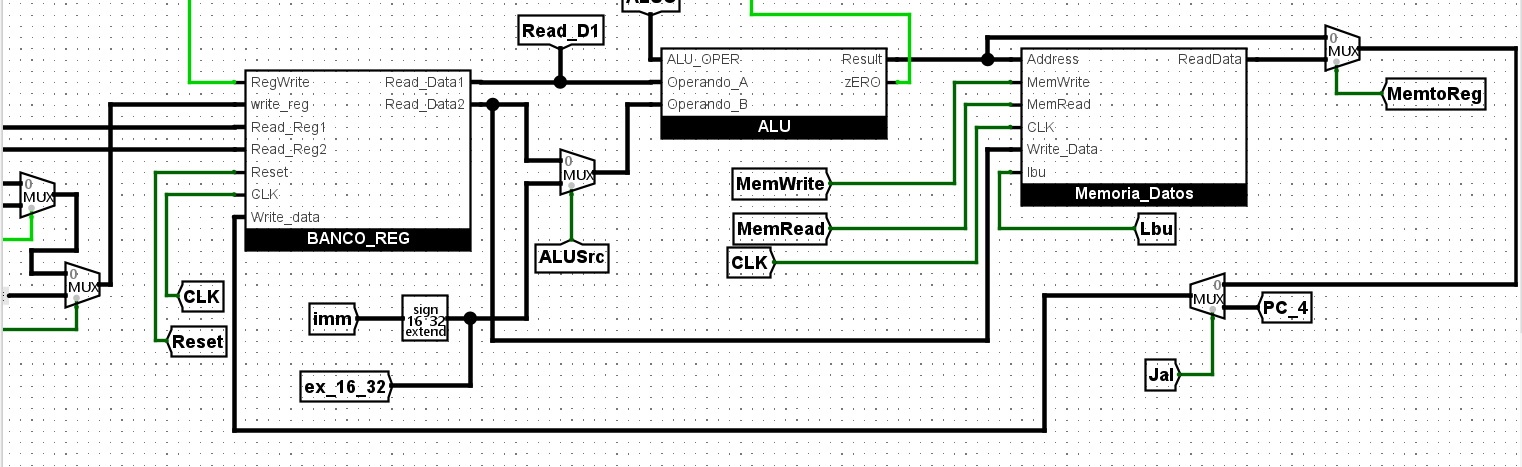
Supongamos que se ejecuta la instrucción sub $s0, $t1, $t2. En este caso la unidad de control interactúa con el módulo ALU\_CONTROL, de tal forma que al ser una instrucción de formato **R con código de operación 0 y código de la función 22h**, se busca en la posición de la memoria 22h(32 decimal) el código **001**, este código, se coloca en la salida ALUO, de tal forma que la ALU ahora sabe que debe ejecutar una resta.

**Consideraciones de la instrucción Jal y Jr.**

Cuando se ejecuta la instrucción Jal nameFunction, se hace un llamado a función y con **Jr $ra** se retorna a la dirección siguiente a donde se produjo el salto, en este proceso es muy importante que se preserve la dirección a donde se debe retornar después de que finaliza el código de la función que fué invocada, es de resaltar que el registro el registro $ra se carga con el valor de la dirección de retorno sin que el programador intervenga conscientemente en eso, la carga de este registro se realiza en background asi:



Como se puede observar en la grafica, al ejecutar jal nameFunction, la unidad de control genera las señales Jal, RegWrite, actúan sobre el banco de registros de tal forma que se habilita la escritura del registro $ra, (cuya valor de posición en el banco es 31 o 0x1f).

****

Como se observa en la gráfica el valor que se escribe en el registro $ra, que procede de PC\_4.

**Problema asignado.**

Escribir un programa para la arquitectura MIPS, se quiere buscar en un vector de enteros el número mayor, el número menor, el número de veces que se repite el menor y el mayor dentro del arreglo. Los resultados se deben escribir en memoria. los números debe estar en el rango [-100, 100]

**Mapeo de memoria**

**.data**

**num\_mayor: .space 4 #Dir 0**

**nun\_menor: .space 4 #Dir 4**

**cantmay: .space 4 #dir 8**

**cantmen: .space 4 #dir 12**

**long: .word 25 #dir 16**

**const: .word 1,4,32 #Dir inicio 20**

**vect: .word 33,3,0,-23,7,0,33,4,40,-23,9,20,33,20,0,1,2,34,12,18, 5,9,0,1,-3**

El inicio del espacio de memoria será la dirección 0x00000000 de la memoria ram

Se reserva espacio para almacenar los valores:

* num\_mayor, espacio para almacenar el número mayor encontrado.
* num\_menor, espacio para almacenar el número menor encontrado.
* cant\_may, espacio para almacenar la cantidad de veces que se repite el mayor.
* cant\_menores, espacio para almacenar la cantidad de veces que se repite el menor.
* Long: contiene la longitud del arreglo que es 25.
* const: las constantes contiene el número 1 para hacer incrementos por 1, el valor 4 para hacer incrementos por 4, y el valor 32 es usado para sumarlo con la dirección base del vector y obtener el inicio del arreglo.

**Algoritmo**

El programa principal asigna memoria a las variables que se usaran en el programa para preservar el mayor valor, el menor, cantidad de repetidos .

direcciona el puntero s0 al inicio del vector de enteros, llama a la rutina mayor\_menor(s0) y llama a la rutina cuantas\_veces\_rep(s0). Finalmente salva en memoria los valores de mayor, menor cantMeno.

**principa**l(){

mayor, menor, cantMayor, cantMeno = asignar Memoria()

s0 = direccionar Puntero Vector

mayor = mayor(s0)

menor = menor(s0)

cantMayor = veces\_rep(s0, mayor)

cantMenor = veces\_rep(s0, menor)

salvarEnMemoria(mayor, menor, cantMan, cantMeno

}

**La rutina calcula y retorna el menor y mayor menor presente en el arreglo de enteros.**

**mayor**(s0){

may=0

punt = s0

longitudvector = leerMemtamaño()

cont = 0

mientras(cont < longitudVector){

valor = leerValorVector(punt)

**si**(may < valor){

mayr = valor

}

cont = cont +1

punt = +4

}

**retorne** mayor}

**menor**(s0){

menor=0

punt = s0

longitudvector = leerMemtamaño()

cont = 0

mientras(cont < longitudVector){

valor = leerValorVector(punt)

**si**( menor > valor){

menor = valor

}

cont = cont +1

}

retorne menor

}

La rutina calcula y retorna las veces que un valor está repetido

veces\_rep(s0, repe){

rep = repe

longitudvector = leerMemtamaño()

punt = s0

cont = 0

cantrep =0

mientras(cont < longitudvector){

aux = leerMem(punt)

if(rep == aux){

cantrep = contrep +1

}

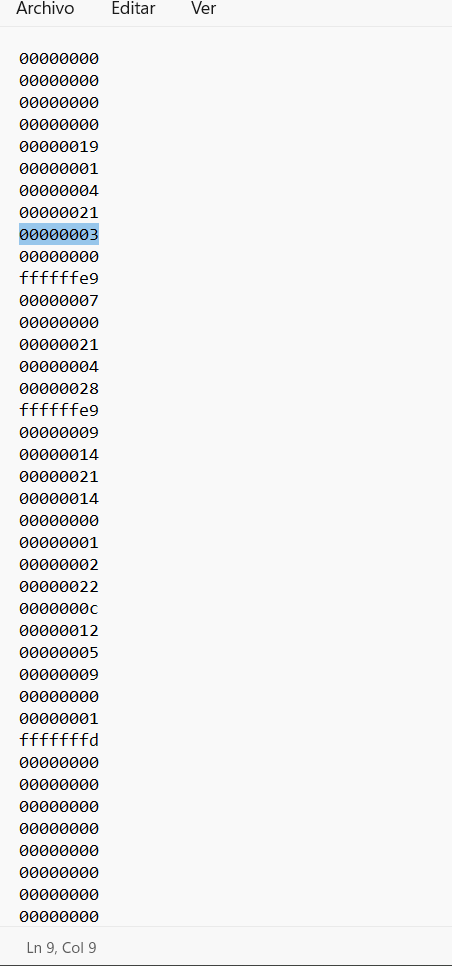
punt = punt +4

}

retorne cantrep

}

**Segmento de datos cargado en la memoria**

****

**Código de máquina y cálculos de los saltos jal. j.**

Para generar el código de máquina, fue necesario usar el simulador MARS4.5 y hacer un recálculo de las direcciones para las instrucciones jal, j.

Para el cálculo parte del hecho que nuestro código queda instalado en la memoria Rom (Memoria de instrucciones a partir de la dirección 0x00000000).

**Recalculo para instrucción jal mayor\_menor**

**Con la instrucción anterior se produce un salto o llamado a la rutina mayor\_menor.**

**la rutina mayor\_menor se ubica en la dirección ox00000034**

**ox00000034 = 0000:address:00**

**00000000000000000000000000110100 (Los bit resaltados se quitan)**

**PC = 0x0000000c**

**pc+4=0x00000010**

**codop (jal) = 0x03**

**address = 00001100000000000000000000001101**

**código (**jal mayor\_menor**) = 0x0c0000d**

**Recalculo para la instrucción jal cuantas\_veces\_rep.**

**cuantas\_veces\_rep = 0x8c**

**PC = 0x0000008c**

**PC+4 = 0x00000090**

**0x0000008c = 0000:address:00**

**00000000000000000000000010001100 (Se quitan bits resaltados).**

**codOp(jal) = =x3**

**00001100000000000000000000100011**

**código(** jal cuantas\_veces\_rep **) = 0x0c000023**

**Recalculo j loop3**

**loop3 = 0x00000050**

**PC=0x0000006c**

**PC+4 = 0x00000070**

**0x00000050 = 0000:address:00**

**ox00000050 = 00000000000000000000000001010000**

**codop (j) = 0x2**

**00001000000000000000000000010100**

**código** (j loop3**) = 0x08000014**

**Recalculo j loop1**

**loop1 = 0x000000a0**

**PC = 0x000000b8**

**PC+4 = 0x000000bc**

**0x000000a0 = 0000:address:00**

**0x000000a0 = 00000000000000000000000010100000 (Se quitan los bits resaltados)**

**opcode (j) 0x2**

**00001000000000000000000000101000**

**código( j loop1 ) = 0x08000028**

**Programa en código binario**

8c10001c

20050019

8c050010

0c00000d

ac020000

ac030004

02204020

0c000023

ac020008

02404020

0c000023

ac02000c

0800000c

8c0f0014

8c180018

02002020

000f7020

00004020

01806020

00004820

1125000b

8c8a0000

0148582a

156e0004

00982020

012f4820

01406020

08000014

01404020

00982020

012f4820

08000014

01001020

01801820

03e00008

8c0c0014

8c0d0018

02002020

00005820

00004820

11250006

8c8a0000

150a0001

016c5820

012c4820

008d2020

08000028

01601020

03e00008

**Conclusiones**

Es muy importante saber cómo está diseñada la arquitectura de un computador. Creemos que esta asignatura al proporcionarnos una visión amplia de cómo se integran los circuitos de un procesador, como interactúan entre ellos y cómo se obtienen los cálculos es fundamental para que nosotros ampliemos el arsenal de herramientas para ser mejores diseñadores de software.

Muy probablemente muchos de nosotros no tengamos que programar en lenguaje ensamblador pero el haber entendido cómo funciona un procesadores nos llevará a ser más conscientes del poder y las limitaciones de la máquina y en consecuencia pensaremos siempre la mejor manera de hacer un código limpio, extendible, y eficiente, en términos de hacer un mejor uso de los recursos de la máquina.

Nos llevamos de esta asignatura una gran riqueza conceptual que hará de nosotros mejores profesionales en la ingeniería de sistemas.

**Bibliografía**

Materiales de la clase

Hojas técnicas del MIP

Estructura y diseño de computadores, David A Patterson