Mattius Alexander Cardona Franco

Daniel Felipe Rivera Arroyave

Cuarto informe de laboratorio

2508355-

Arquitectura de computadores y laboratorio

Fredy Alexander Rivera Velez

2018-1

Para el diseño del procesador monociclo se siguió el diseño propuesto en la guia de laboratorio No.4 capaz de realizar las instrucciones lw, sw, add, sub, and, or, nor, slt, beq y jump, adicionalmente fue agregada la operación addi por su utilidad a la hora de desarrollar el algoritmo.

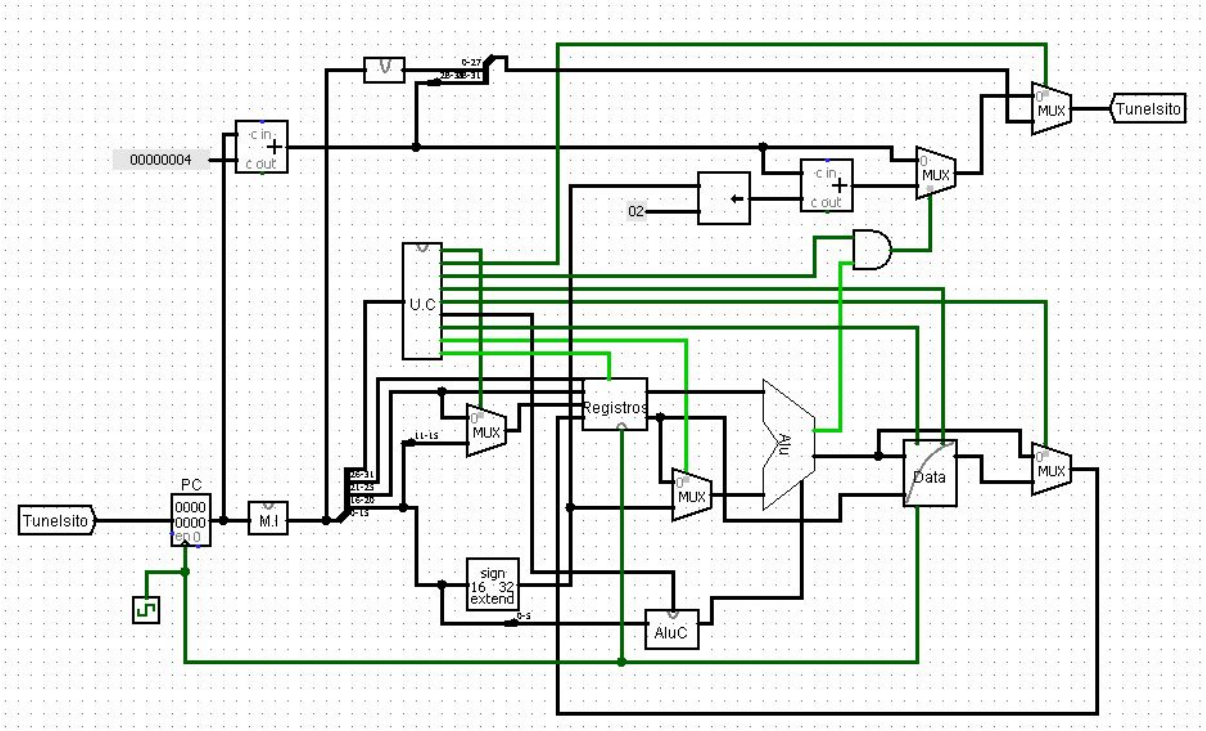


Figura 1: Procesador monociclo en logisim.

El proceso de diseño de los elementos más importantes del procesador será descrito a continuación.

Memoria de instrucciones:

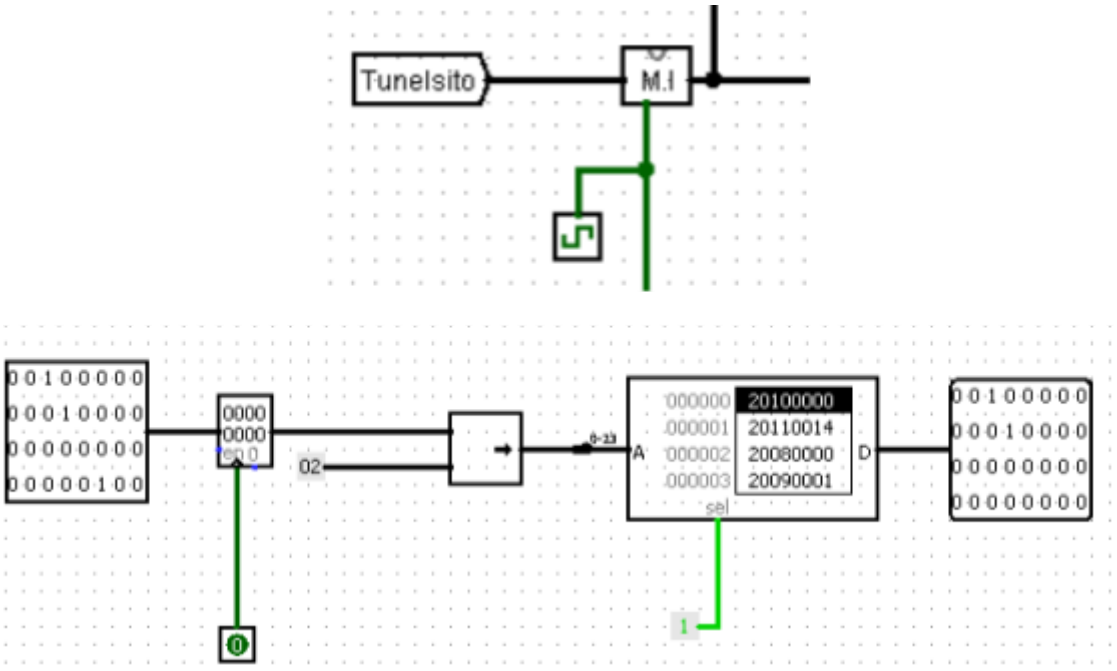
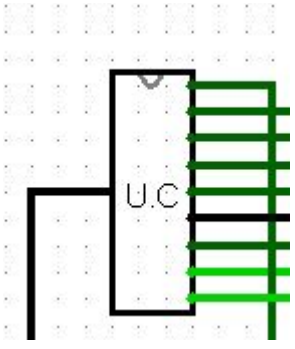


Figura 2: Memoria de instrucciones dentro y fuera

La memoria de instrucciones está compuesta por una entrada que corresponde a la siguiente posición y un registro de 32 bits que se comporta como un flip flop tipo D, como las memorias en logisim avanzan de a una unidad es necesario realizar un SRL al estado del registro(Normalmente se actualiza sumando 4) y posteriormente usar un splitter para ignorar sus 8 bits más significativos, esto es debido a que las memorias en logisim sólo admiten 24 bits de direccionamiento.

En la memoria ROM se encuentran cargadas las instrucciones del procesador en código máquina que salen inmediatamente se selecciona la dirección donde se encuentra ubicada.

Unidad de control:



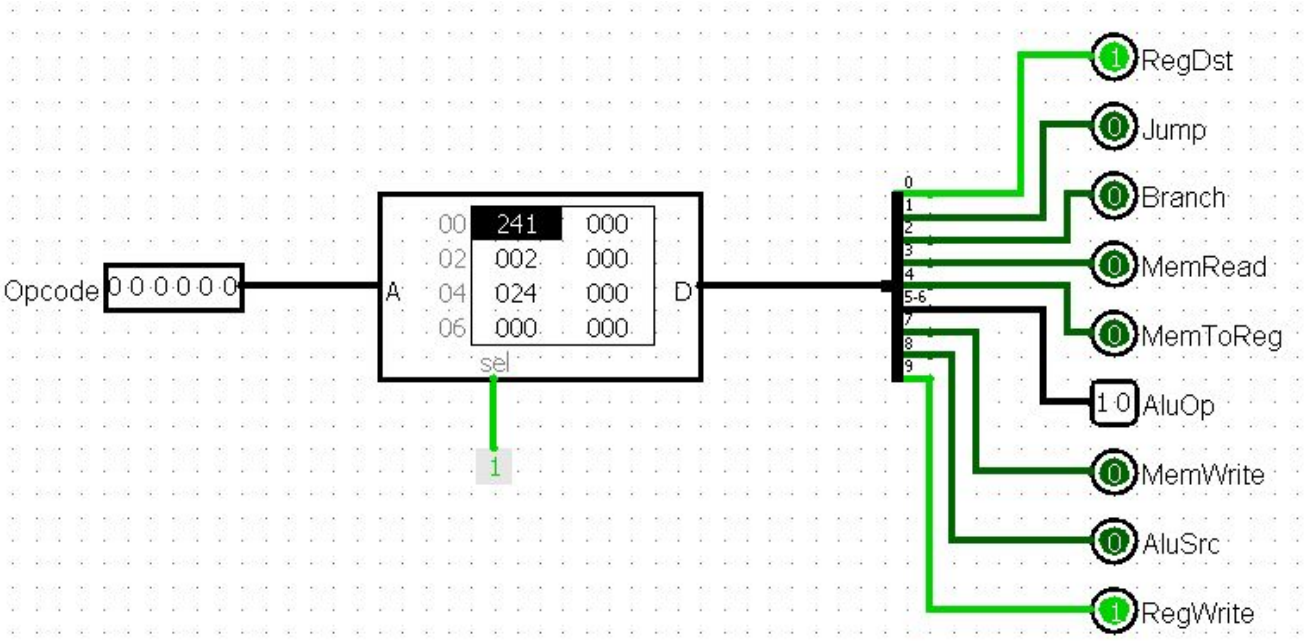


Figura 3:Unidad de control dentro y fuera.

La unidad de control como su nombre lo indica, se encarga de controlar todo lo que realiza el procesador, su entrada corresponde a los 6 bits más significativos de cada instrucción (Opcode). Para asignarle a cada entrada sus salidas correspondientes, había dos opciones posibles: realizar los mapas de karnaugh o usar una memoria, se escogió la segunda opción para simplificar el trabajo. los valores cargados a la memoria se escogieron de la siguiente manera :

Imput	*Reg Write*	*Alu Src*	*Mem Write*	*Alu Op1*	*Alu Op0*	*Mem ToReg*	*Mem Read*	*Branch*	*Jump*	*Reg Dst*	*HEX*
T. R	***1***	***0***	***0***	***1***	***0***	***0***	***0***	***0***	***0***	***1***	***241***
100011	***1***	***1***	***0***	***0***	***0***	***1***	***1***	***0***	***0***	***0***	***318***
101011	***0***	***1***	***1***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***180***
000100	***0***	***0***	***0***	***0***	***1***	***0***	***0***	***1***	***0***	***0***	***24***
000010	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***1***	***0***	***2***
001000	***1***	***1***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***300***

Las salidas de la Unidad de control corresponden a la representación en binario del número en el registro, siendo su bit más significativo la salida más abajo y el menor la primera salida.

AluControl:

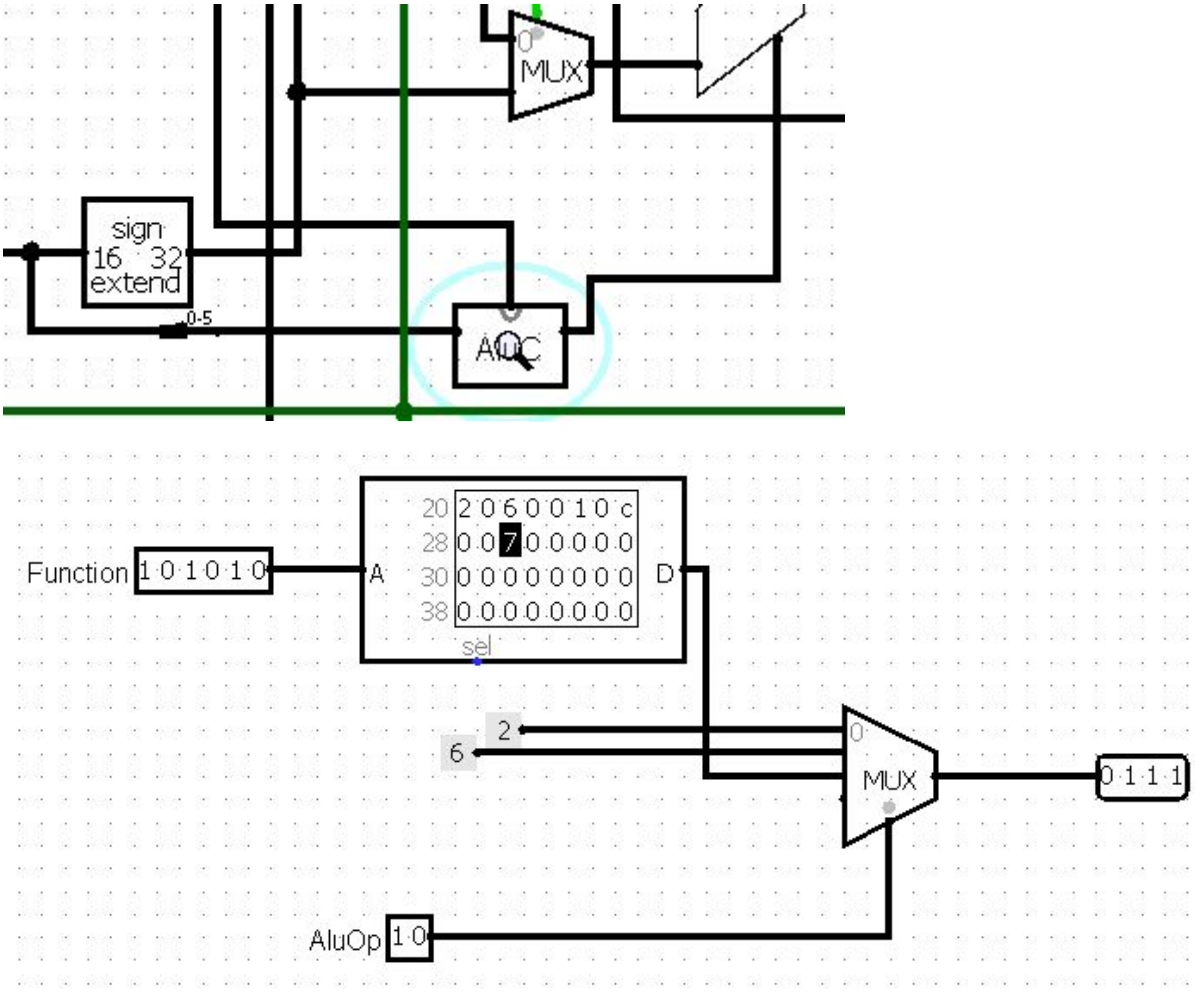


Figura 4: AluControl dentro y fuera.

El AluControl se realizó con los indicativos en la tabla 1 de la guia de laboratorio No. 4, la entrada AluOP que viene desde la unidad de control actúa como selector de un multiplexor cuya salida de 4 bits será el selector del Alu.

Para los valores 00 y 01 del AluOp se usaron las constantes 2 y 6 respectivamente, cuando el AluOp indica 10 es porque la instrucción actual es de tipo R y tiene un function asociado a sí misma(6 bits menos significativos) para estos se analizó el function de cada instrucción y se guardó nuevamente en una memoria ROM la salida correspondiente a cada function posible.

Alu:

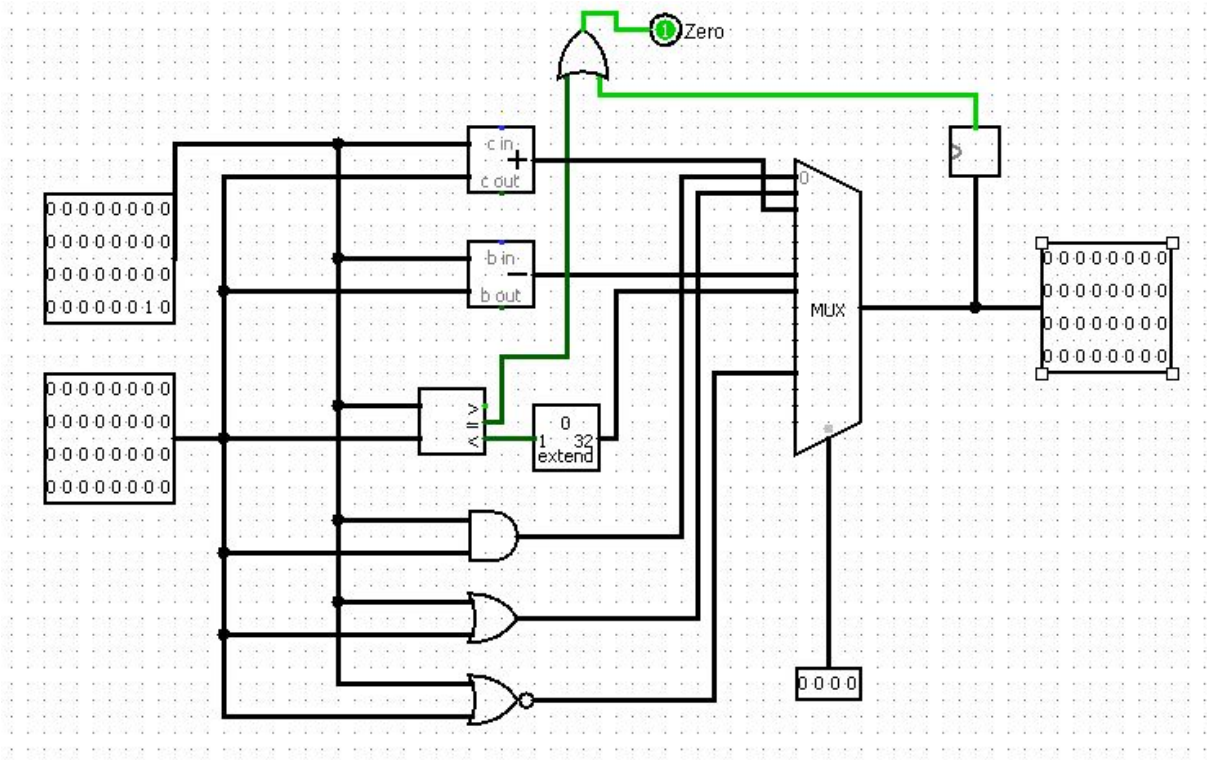


Figura 5:Alu(Unidad lógica aritmética)

Memoria de datos:

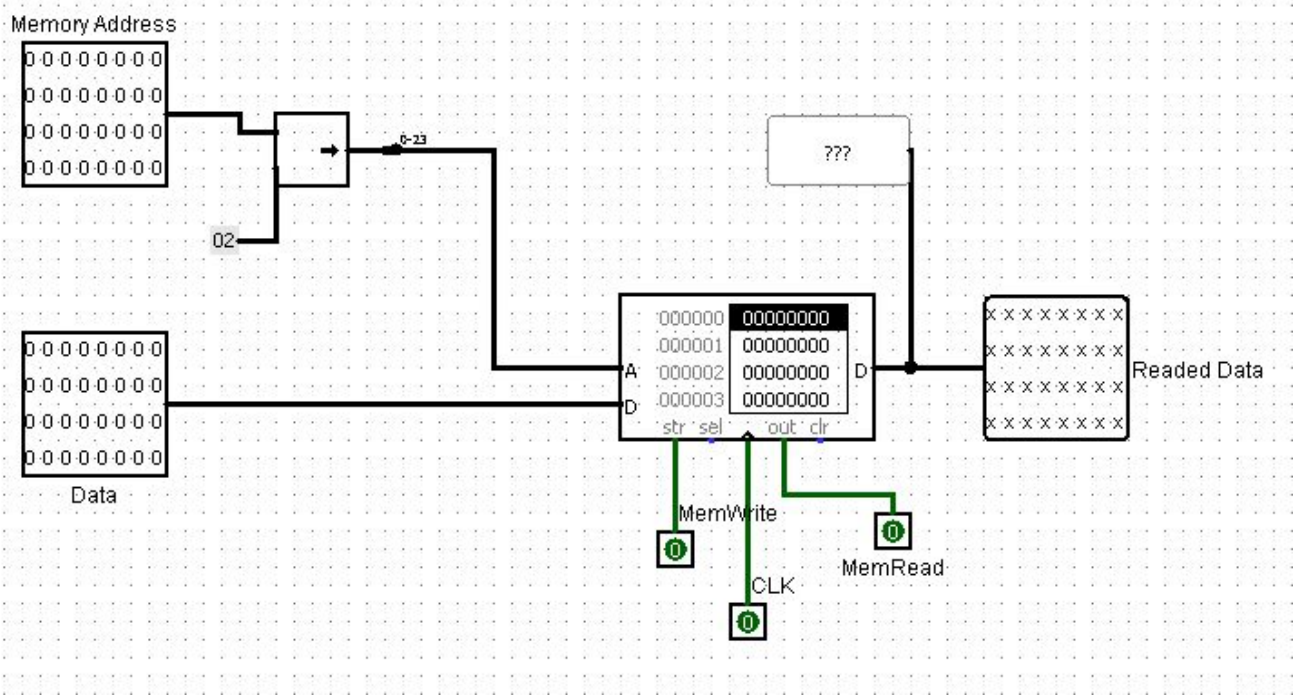


Figura 6:Memoria de datos.

Hasta el momento las memorias utilizadas han sido ROM, debido a que los valores guardados en esta son constantes y se preservan, para la memoria de datos se decidió utilizar una memoria Ram,(se modifica constantemente y no preserva los datos).

Se le aplica un SLR para controlar el hecho de que las posiciones en el logisim avanzan de a 1 y no de a 4 como se hace en el MarsMips y adicionalmente se le ignoran sus 8 bits más significativos porque solo permite 24 de direccionamiento. El habilitador de “Storage” depende de la salida MemWrite de la unidad de control y la de Out depende del MemRead también proveniente de la unidad de control, agregamos un visualizador para ver los datos y comprobar los datos cuando son leídos ya que estos en memoria se encuentran presentados en base hexadecimal.

Instrucciones de uso:

Si se desea hacer pruebas o usar el procesador, es importante tener las instrucciones a realizar en código máquina y agregarlas a la memoria de instrucciones.

La herramienta MarsMips permite exportar un archivo con este código, pero primero hay que realizar una configuración para evitar posibles errores en las instrucciones de tipo Jump.

Una vez se tenga el código completo escrito en el MARS, se deben seguir los siguientes pasos:

1)Ir a la pestaña settings y seleccionar el apartado de configuración de memoria.

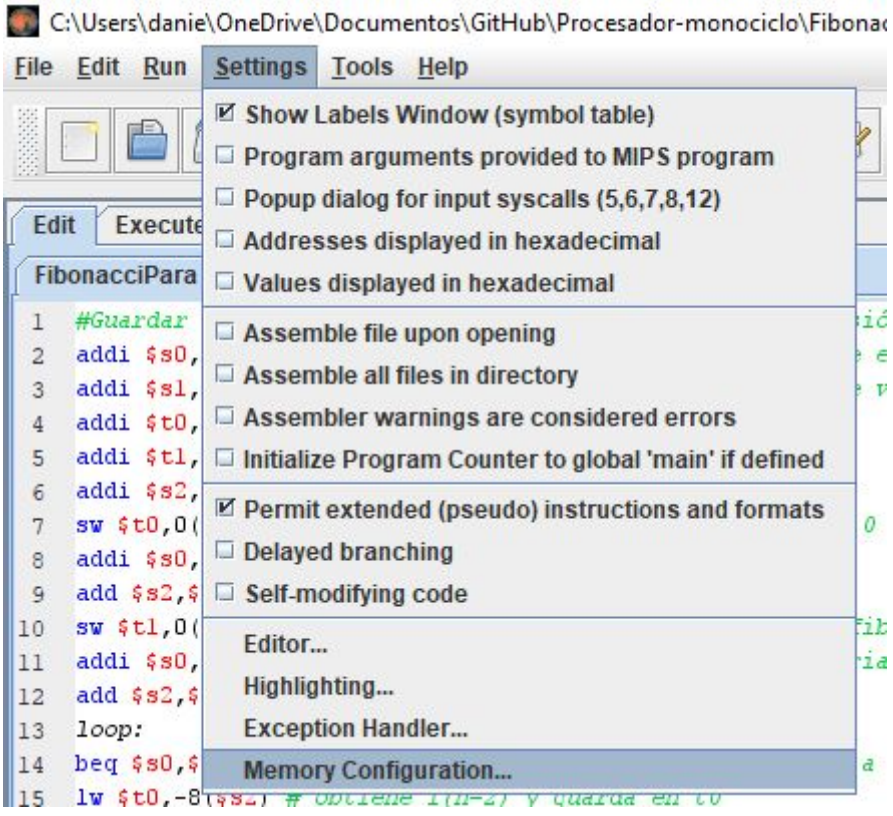
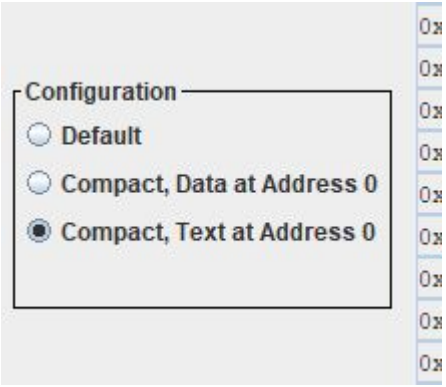


Figura 7: Configuración mars

2)Una vez aparezca la ventana emergente, seleccionar la tercera opción “Compact, Text at Address 0”



Esto debido a que la configuración predeterminada del MarsMips guarda la instrucción en posiciones de memoria muy elevadas, y al cargar estas instrucciones en la memoria de instrucciones de nuestro procesador monociclo son cargadas desde la posición 0, esto genera un error en las instrucciones de tipo jump cuyo apartado de target es la posición de memoria cargada en mars.

Una vez nos aseguremos que esto ha sido configurado correctamente, sólo queda ensamblar nuestro código y clickear el botón 

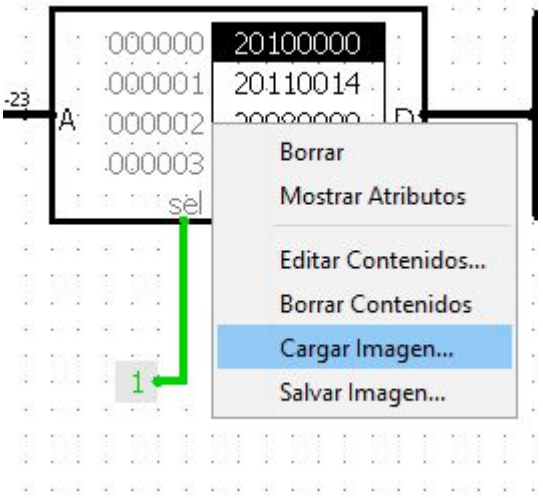
que nos brinda la opción de exportar el código máquina.

Aclaración: Es necesario agregar una directiva al código máquina para que pueda ser leído e interpretado correctamente en las memorias del procesador, dicha directiva es: “v2.0 raw”.

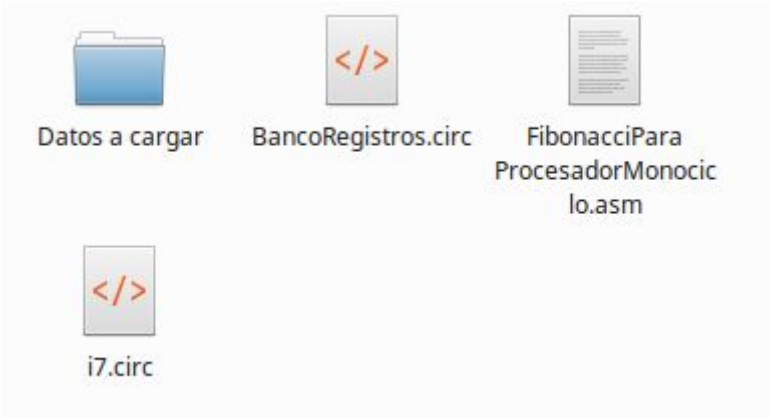
```
v2.0 raw
20100000
20110014
20080000
20090001
20120000
ae480000
22100001
22520004
ae490000
```

Ahora debemos ingresar al circuito del procesador, luego a la memoria de instrucciones, dar click derecho a la ROM y luego a la opción de cargar imagen, debemos seleccionar el archivo previamente exportado desde el MarsMips (cabe resaltar que las instrucciones también

pueden ser cargadas de forma manual seleccionando la posición de memoria y luego escribirla)



El circuito tiene cargados de forma predeterminada las instrucciones del problema asignado, así como también los valores correspondientes a las memorias de la unidad de control y el alu control, pero en caso de que estos sean borrados por equivocación o que no funcione correctamente es necesario volver a cargarlos para asegurarse que no sea este el problema, en la carpeta donde se encuentra el procesador (Proyecto de logisim “i7.circ”) hay una subcarpeta llamada “Datos a cargar” que guarda los respectivos archivos para las memorias(**Fibonacci** contiene el código para resolver nuestro problema asignado, **Unidad de Control** los números en hexadecimal equivalentes a los mapas de karnaugh para activar las salidas pertinentes en cada caso y **ALUC-R** los números en hexadecimal equivalentes a los mapas de karnaugh que se usan cuando se recibe una instrucción u OpCode de tipo R).



Al finalizar la ejecución del algoritmo previamente cargado, podemos observar en la memoria (circuito Data) que se guardaron correctamente los 20 primeros números de la sucesión de fibonacci en hexadecimal, incluyendo el 0, en orden y sin ningún fallo aparente:

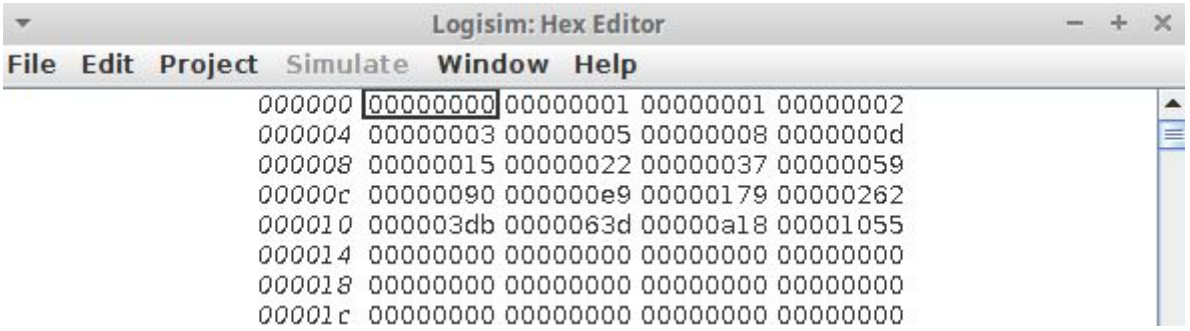


Figura 6: Resultados de la ejecución.

Diagrama de flujo de la solución:

Nuestro problema nos pedía generar y almacenar en memoria los 20 primeros elementos de la sucesión de fibonacci (0,1,1,2,3,5,8,13,21,34,55...), por lo que para la solución a éste problema implementamos la siguiente lógica:

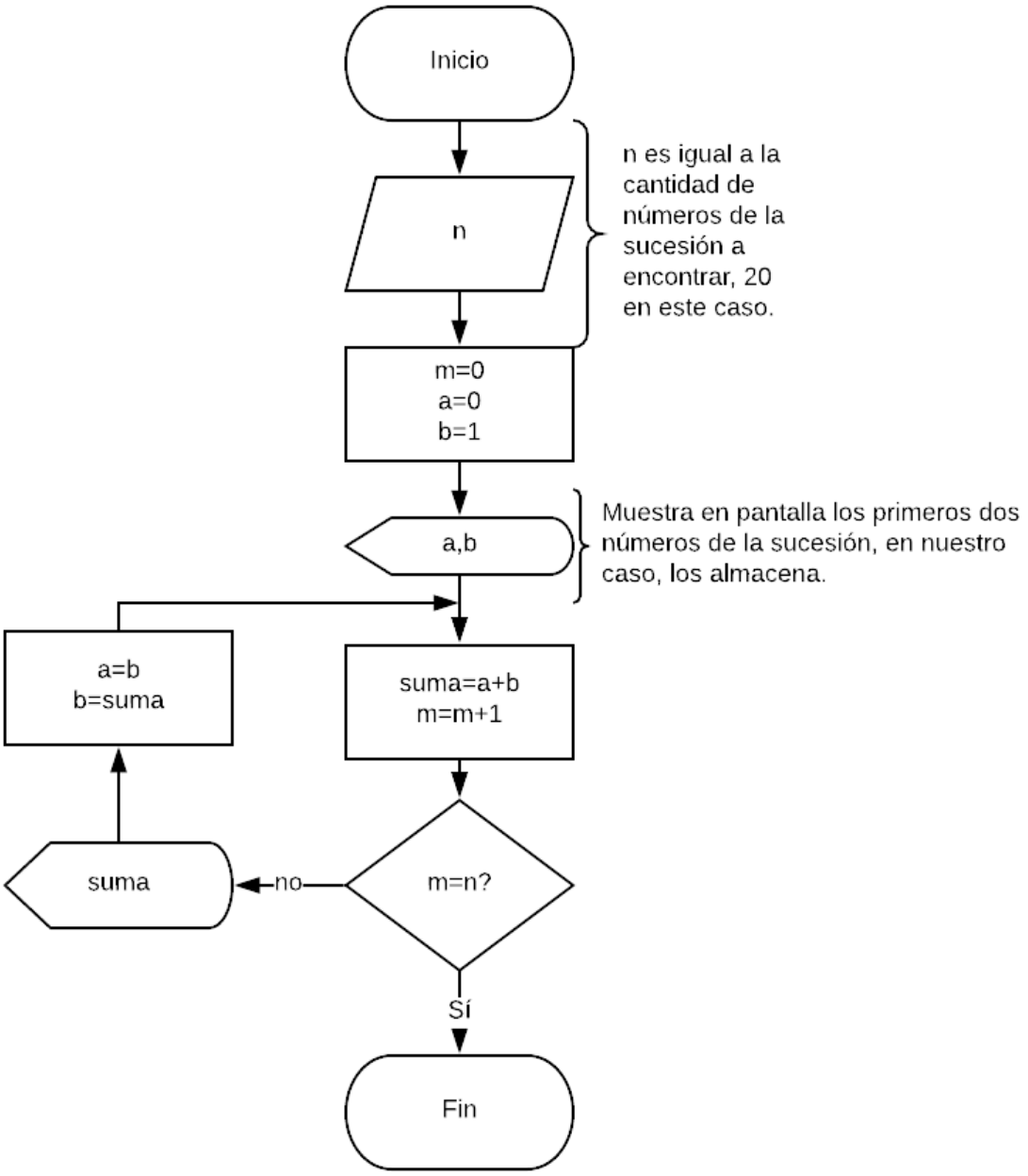


Figura 7: Diagrama de flujo sucesión de Fibonacci

Código en assembler:

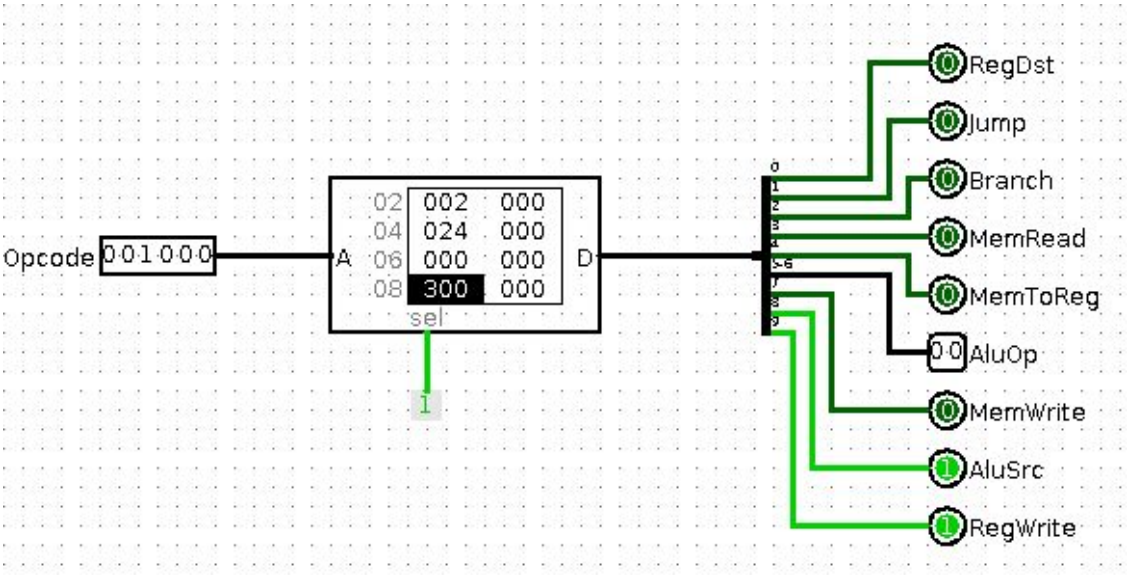
```
#Guardar en memoria los 20 primeros términos de la sucesión de fibonacci
addi $s0,$zero,0        #Se inicializa s0 en 0 y la cantidad de elementos
addi $s1,$zero,20       #se inicializa s1 en el valor final de valores a buscar.
addi $t0,$zero,0        #guarda en t0 fibonacci de 0 f(0).
addi $t1,$zero,1        #guarda en t1 fibonacci de 1 f(1).
addi $s2,$zero,0        #apunta a la dirección 0
sw $t0,0($s2)           #guarda en el primer valor de fibonacci on 0
addi $s0,$s0,1          #Aumenta en 1 el término
add $s2,$s2,4           #aumenta el apuntador de memoria
sw $t1,0($s2)           #guarda en memoria el segundo término de fibonacci
addi $s0,$s0,1          #Aumenta en 1 el término y la pos de memoria
add $s2,$s2,4           #aumenta el apuntador de memoria

loop:
beq $s0,$s1,finLoop     #si la cantidad de números es igual a 20 se detiene.
lw $t0,-8($s2)          # obtiene f(n-2) y guarda en t0
lw $t1,-4($s2)          # obtiene f(n-1) y guarda en t1
add $t3,$t0,$t1         # guarda en t3 f(n)
sw $t3,0($s2)           #guarda en memoria el término calculado
addi $s0,$s0,1          #pasa al siguiente número
add $s2,$s2,4           #aumenta el apuntador de memoria
j loop
finLoop:                #Fin del código
```

Problemas que se presentaron y soluciones planteadas:

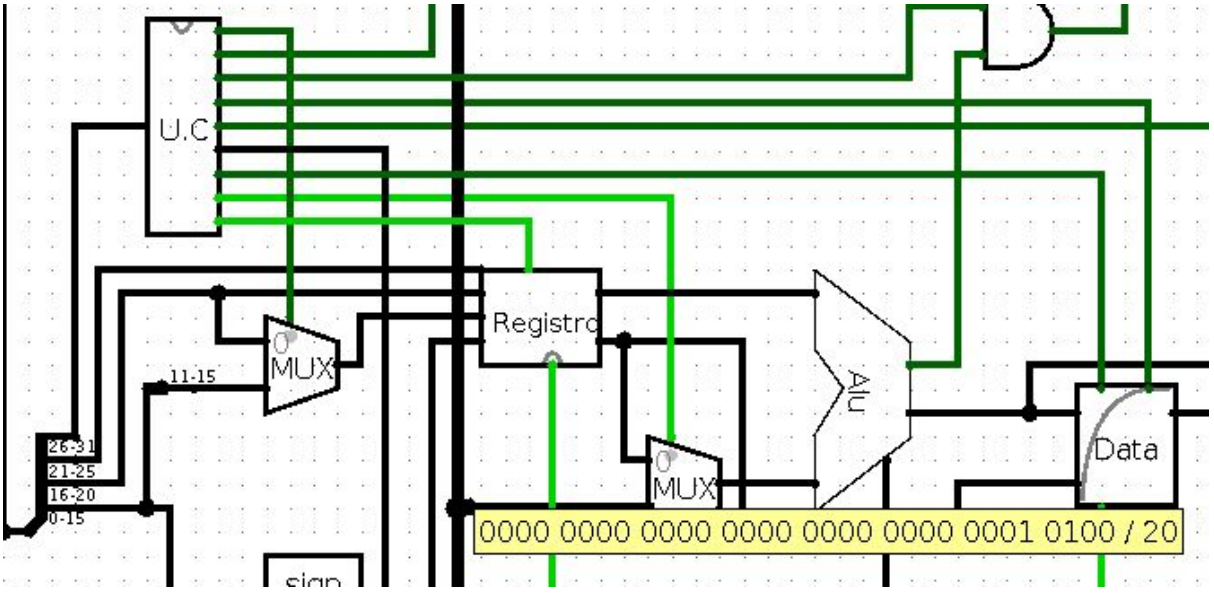
- El diseño del procesador monociclo plantea que a la posición de memoria de la instrucción actual (PC) se le sume 4 para acceder a la siguiente instrucción, pero en el logisim las memorias tienen un aumento de a 1 entre cambios de posición,

entonces había dos soluciones posibles: Hacer aumentos de a 1, o hacerle un SlR de 2, que es equivalente a dividir por 4, optamos por esta opción.

- Para resolver nuestro problema asignado era necesario que algunos registros iniciaran con unos valores predeterminados, para esto encontramos dos soluciones: que dicho valor se encuentre cargado en la memoria de datos(Debe restaurarse cada vez que se reinicie el circuito porque es una memoria RAM) o realizar los cambios pertinentes para que soporte la instrucción ADDI y que dichos valores le sean cargados en el código mediante esta instrucción.
  Esto se hizo posible haciendo que cuando le llegue a la unidad de control un opcode de (001000) solo se habiliten las salidas de RegWrite y AluSrc



La salida del AluSrc selecciona la entrada 1 de un multiplexor que escoge entre otra salida del banco de registros y el valor inmediato(este es el necesario para esta operación que se realiza en el Alu)



- Las instrucciones de Jump no se estaban realizando correctamente y los saltos los hacía a posiciones de memoria desbordadas o que no correspondian, llegamos a pensar en usar una mala técnica de programación y realizar los saltos con la

instrucción “beq $t0,$t0,label” , pero le hicimos seguimiento a estas instrucciones creando nosotros mismos el código máquina y comprobamos que funcionaba bien, entonces analizamos qué era lo que sucedía en mars y vimos que las posiciones de memoria de las instrucciones comenzaban en un valor diferente de 0 (Como lo hace el procesador monociclo), solo fue cuestión de configurar correctamente el MARSMIPS para que comience desde la posición de memoria 0.

Conclusiones:

- Al poder ver cómo funciona internamente un procesador y además conseguir los insumos teóricos para la implementación de uno, se consigue hacer que no se vea la máquina y la manera en que se ejecutan las instrucciones como una caja negra con la cual no podemos tener comunicación más allá que con lenguajes de programación de alto nivel, ignorando lo que ocurre por debajo, sino que empezamos a comprender la manera en que grandes genios no hace muchas décadas concibieron la computación y las cuales siguen siendo hoy las bases necesarias para comprender la computación por lo menos en los próximos años venideros.
- Podemos evidenciar la gran utilidad que presta la ALU(Unidad lógica aritmética) en el desarrollo de un procesador monociclo mips y cómo interactúa con los demás componentes para que puedan realizar las instrucciones correctamente.
- Este tipo de prácticas nos ayudan a corroborar los conocimientos atendidos en clase y verlos en funcionamiento, comprender la transformación de una instrucción en código máquina y ver cómo es interpretada por el procesador para realizar su labor correctamente.