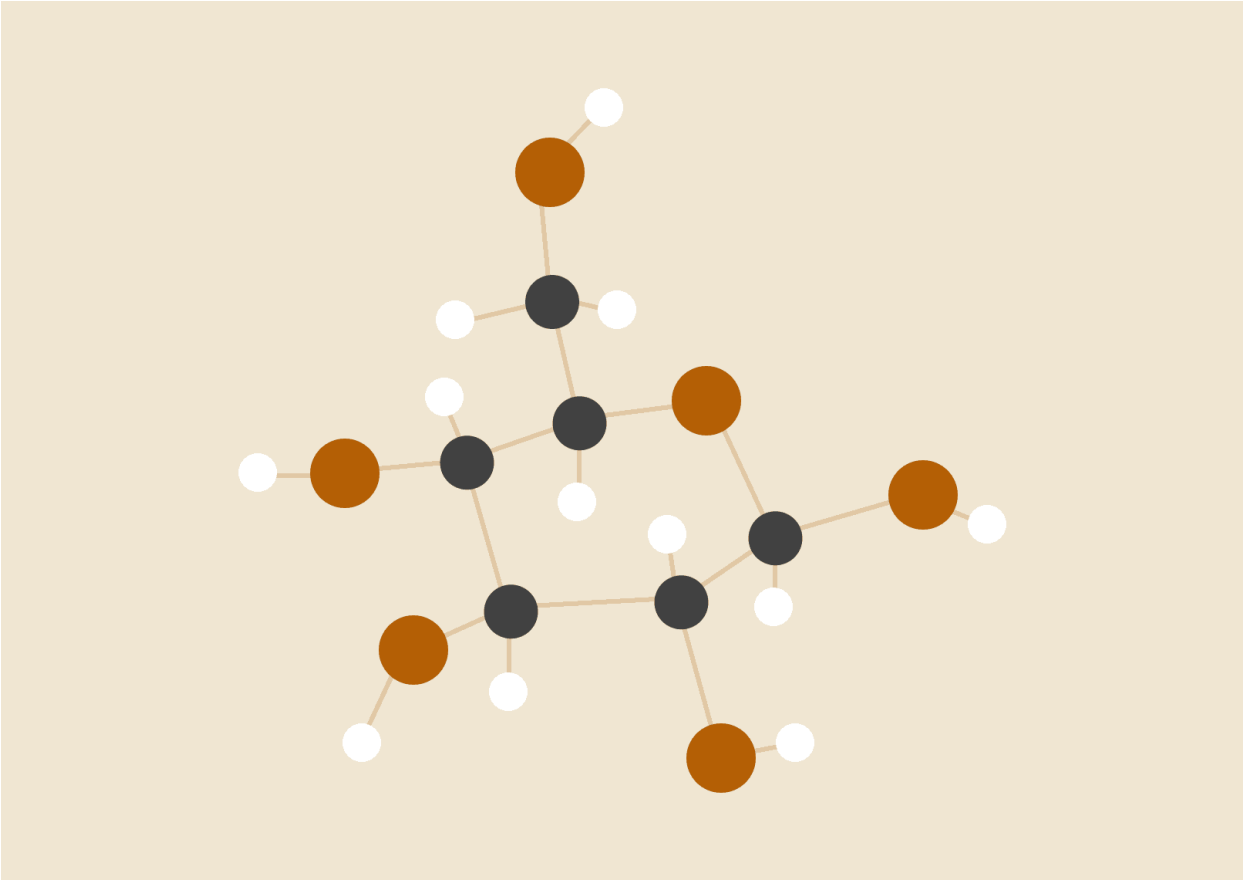
PRÁCTICA 34 ARQUITECTURA DE COMPUTADORES

FREDY ALEXANDER RIVERA VÉLEZ



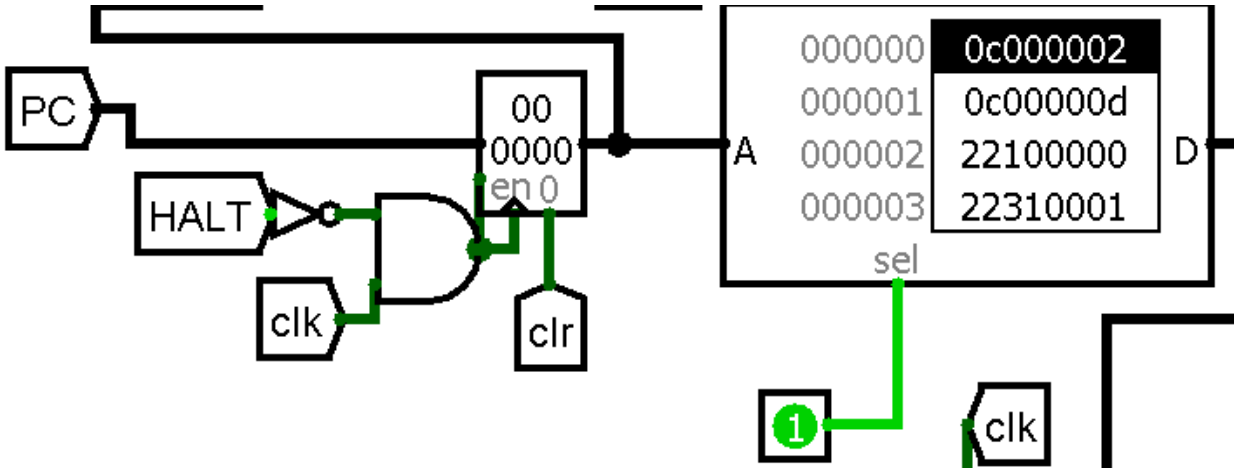
Emiro Moreno Soto

Daniel Sierra Mejia

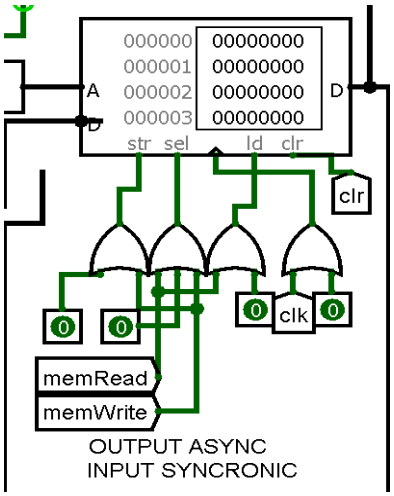
27/07/2020
UNIVERSIDAD DE ANTIOQUIA

LA MEMORIA

La memoria fue una memoria ROM palabras de 32 bits para las instrucciones accedidas secuencialmente de 1 en 1.En esta implementación no fue necesario en varias instrucciones como el BEQ o el JUMP , multiplicar por 4 debido a que con PC+1 se podía acceder secuencialmente sin ningún problema .

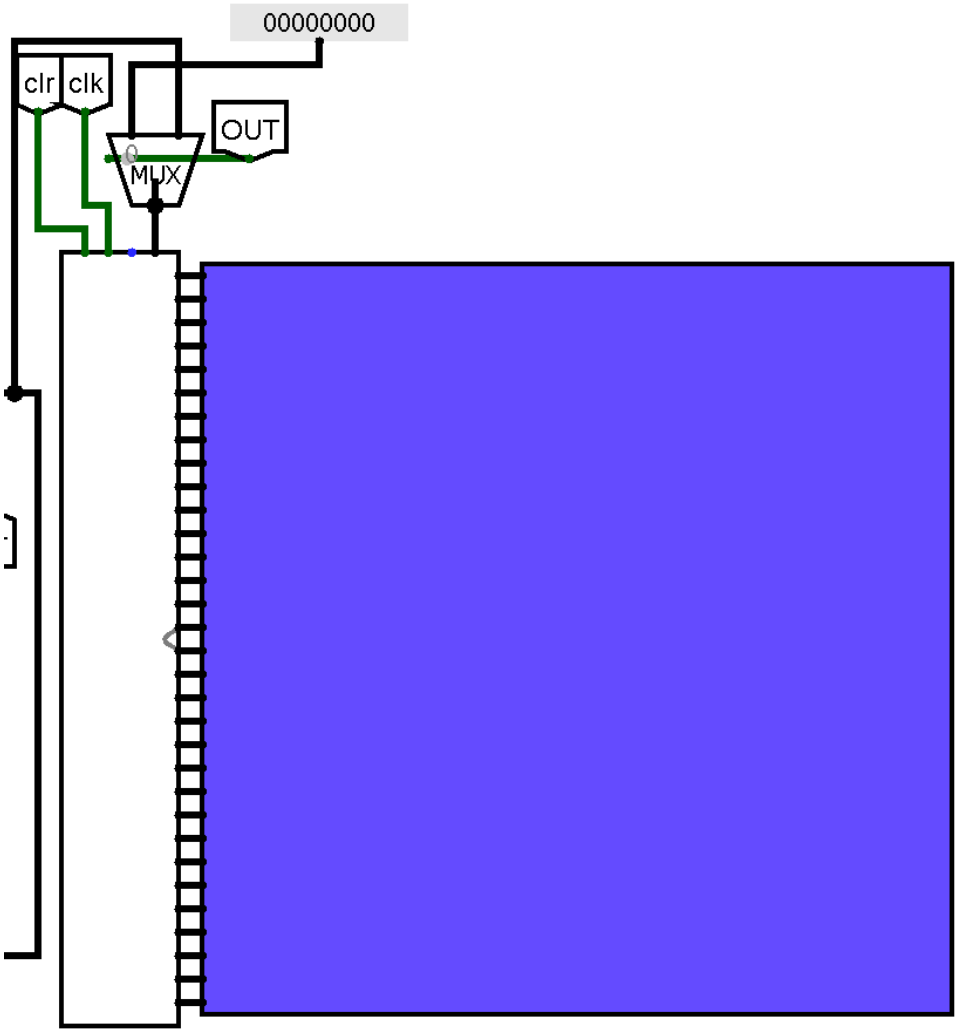


Para la memoria de datos también se utilizo la memoria de logisim con unas entradas extra por si se debía escribir un dato manualmente para TESTING que no se incluyeron en el resultado final.



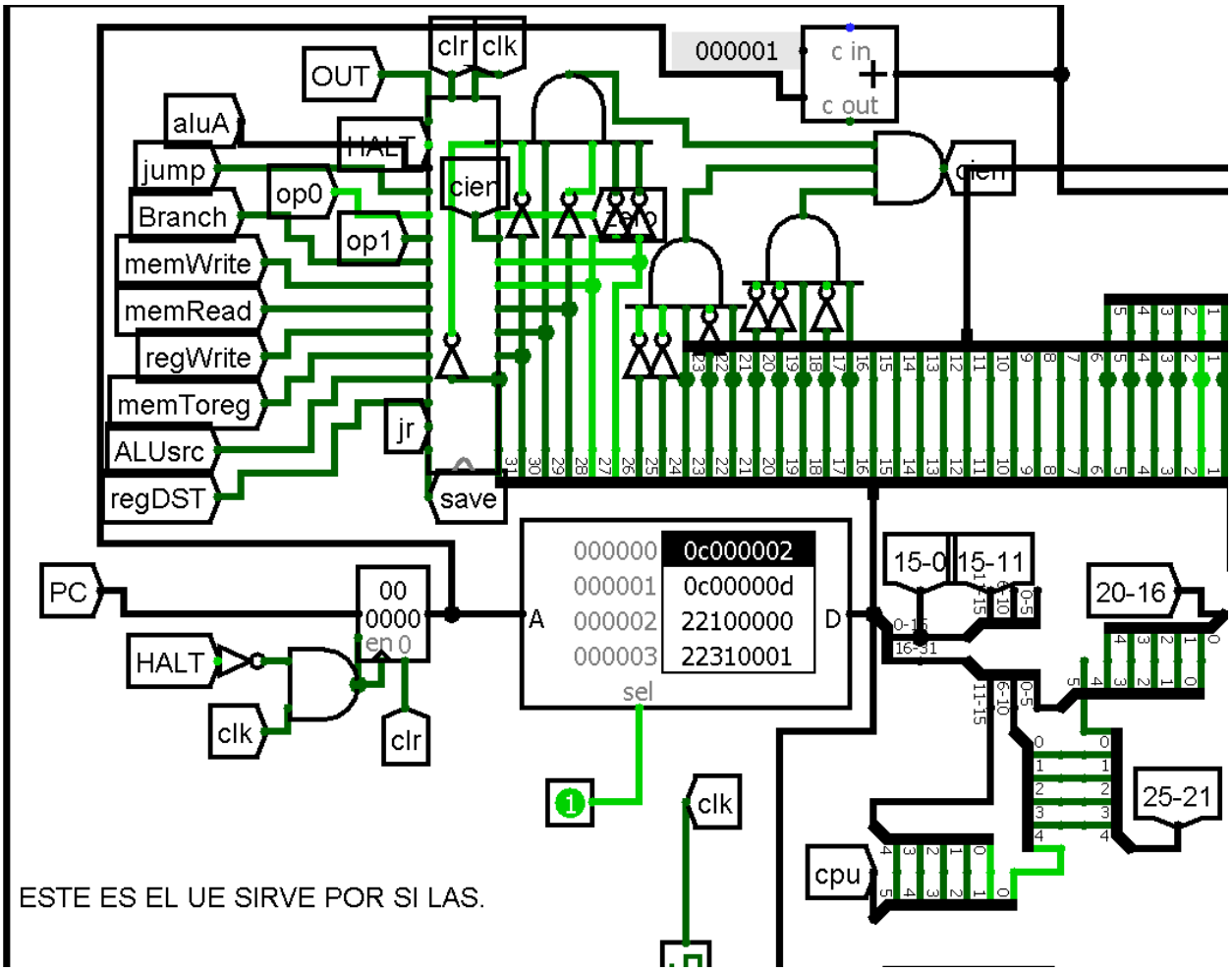
SALIDA

Para la salida se utilizo la pantalla de la practica numero 2 con los registros en serie para mostrar secuencialmente los valores en memoria .Este dispositivo es accionado por el OUT del procesador ,en caso de que sea cero un mux accionara el valor cero para mandarlo a la pantalla y que no muestre nada.

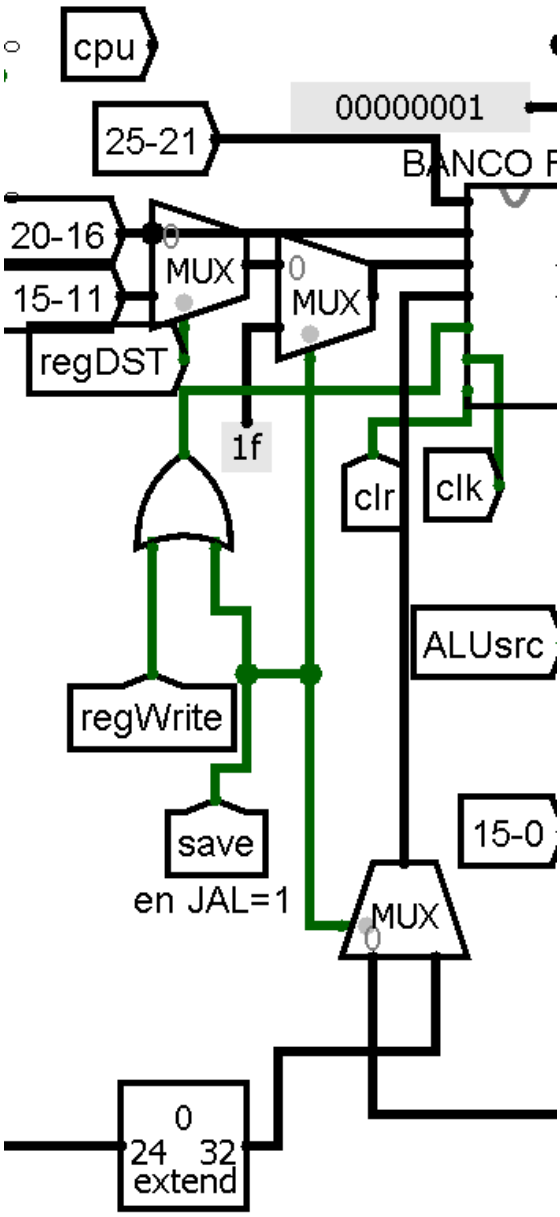


RUTA DE DATOS

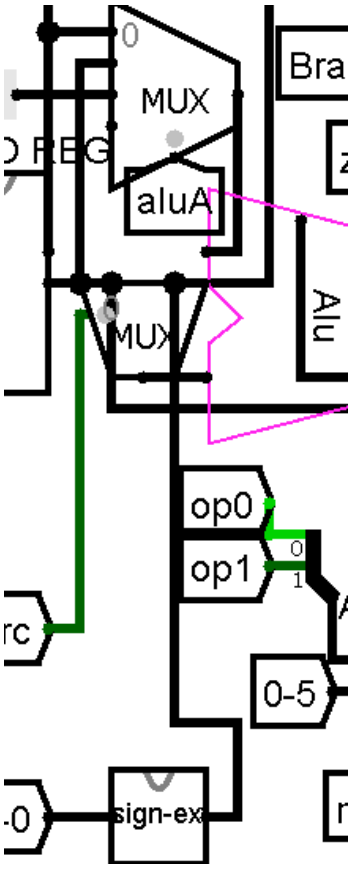
En el camino de datos la instrucción saldrá del ROM donde será desglosada por un árbol de splitters en donde 25-21 , 20-16,15-11 serán registros que irán al banco para ser seleccionados según sea la instrucción ,estos dos últimos se seleccionará un mux segun la instruccion que sea.Los 6 bits más significativos irán al procesador y existira otro spliter para que las instrucciones Jump puedan acceder directamente a la dirección efectiva.El registro PC simplemente sumará 1.



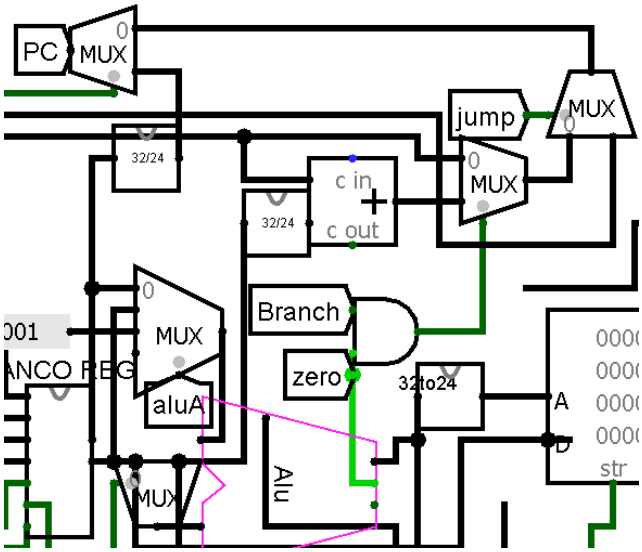
En la parte intermedia tendremos el banco de registros con sus accesos read 1 y 2 en donde habrá un mux según sea el registro que se quiere leer en la parte 2 además de otro mux adicional que pasara el número 31 para poder escribir en ese registro cuando se invoque un JAL junto con otro mux que controlara que el valor en el registro 31 sea PC+1 o sea cualquier otro proveniente de la memoria que se va a consolidar en el banco de registros en cualquier otro caso.



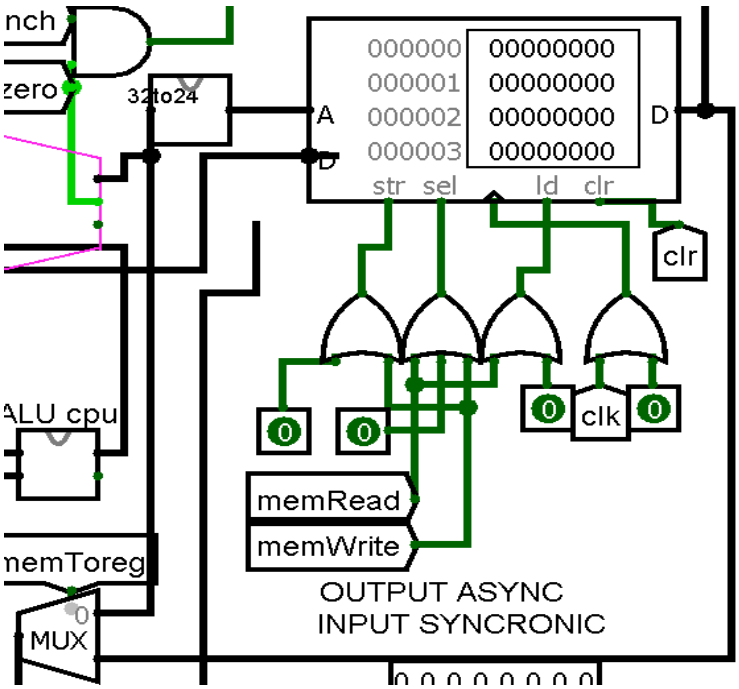
En la parte siguiente tendremos las salidas del banco y el inmediato que serán controlados por MUXES tanto en ALUB como ALUA .El nuevo mux implementado en ALU-A puede seleccionar según la instrucción Read data 1,2 o una constante de 1 .El mux de B es el normal que escoge entre Read data 2 y el número inmediato que posee el extendedor de signo para efectos de la ALU.



Saliendo de la ALU tenemos la salida normal que va a la memoria y la salida zero que va con el AND gate para efectuar un salto branch lo cual va a un sumador para PC+1 + los saltos hacia adelante .Este resultado va a un MUX controlador que escoge si el siguiente PC sera PC+1 o la dirección efectiva del branch .Luego ira este resultado a dos Muxes mas que evaluaran y filtraran el resultado dependiendo si la dirección es por un J o JR que finalmente irán al PC.



Como se puede observar en la imagen anterior , el resultado de la memoria se debe reducir a los 24 LSB debido a que los componentes de logisim solo dejan un direccionamiento de hasta 24 bits y de 1 en 1 , no mas 4.Por lo demás la memoria posee señales de write/read y clear ,su salida es al multiplexor que definirá si sale el resultado de la ALU o lo que la memoria encontro basado en ese resultado con la señal memtoreg.

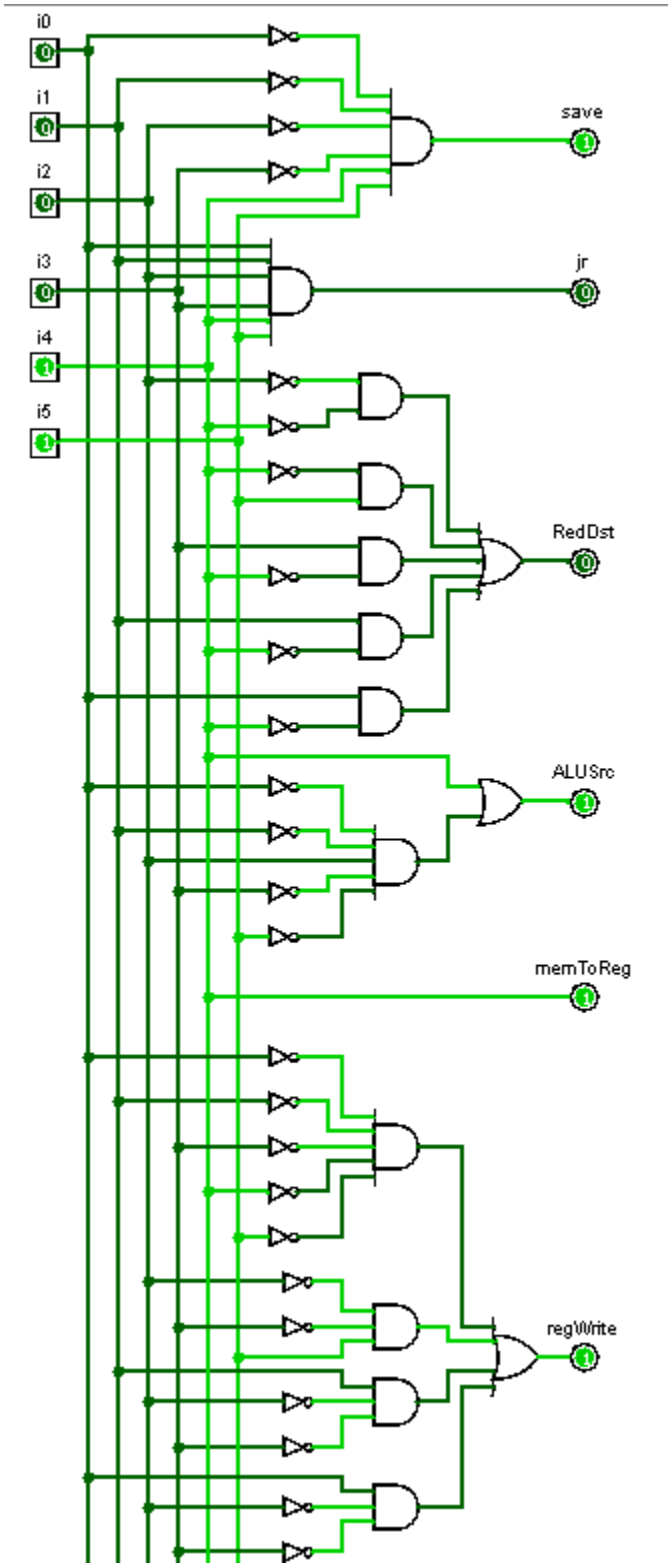


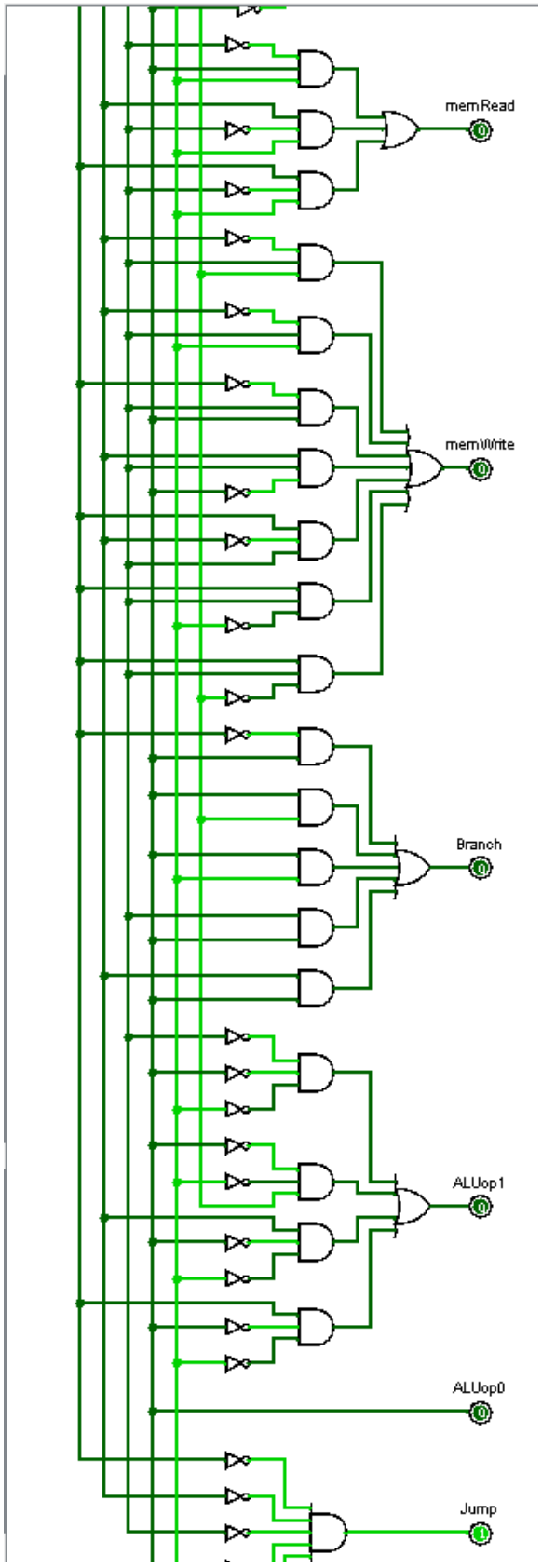
EL PROCESADOR

El procesador es un componente combinacional típico de la arquitectura mips ,soporta las instrucciones load word, store word, add, sub, and, or, nor, set-on-less-than, branch if equal, jump, jump and link y jump register exigidas pro la práctica.Adicionalmente soporta sumas inmediatas con la operación Addi que está organizada exactamente como lo indica la tabla de referencia mips (no hay diferencia alguna).

Como se mencionó el procesador es un componente combinacional el cual ejecuta una instrucción a la vez durante 1 solo ciclo de reloj;IPC=1 en su forma más básica.En el logisim primero se creó un CPU llamado simplemente de esa manera que soportara todas las básicas , se testeo y luego se creó una versión mejorada llamada el CPU2 que soportaba JAL y JR siendo aun completamente combinacional .Las instrucciones son las siguientes para el CPU2 que luego se le adiciono la funcionalidad addi tambien:

	i0	i1	i2	i3	i4	i5	save	jr	RedDst	ALUSrc	memToReg	regWrite	memRead	memWrite	Branch	ALUop1	ALUop0	Jump
LW	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0
SW	1	0	1	0	1	1	0	0	0	1	1	0	0	1	0	0	0	0
TIPO R	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0
BEQ	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0
J	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1
JAL	0	0	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1
JR	1	1	1	1	1	1	0	1	0	1	1	0	0	0	1	0	1	0
ADDI	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0


i0
i1
i2
i3
i4
i5
save
jr
RedDst
ALUSrc
memToReg
regWrite


memRead
memWrite
Branch
ALUop1
ALUop0
Jump

Posteriormente se pensó en hacer una nueva funcionalidad llamada OUT ;tarea elaborada laboriosamente ya que para este fin se debió proceder a llevar las cosas un poco más allá. Para este fin se vio inicialmente la necesidad de que el mismo procesador hiciera unas subrutinas que fueran disparadas de alguna manera ; la idea era que el procesador mostrará por algún medio varios valores alojados en memoria secuencialmente por medio de algún dispositivo de salida conectado al computador .La palabra subrutinas supuso inmediatamente una máquina de estado finito pero habia un problema de que el procesador era monociclo además de combinacional .

Dándonos a la tarea hubo que tener varias consideraciones :

1.la subrutina sería un algoritmo o un algoritmo que también tendría que tomar decisiones en algún momento para cambiar de estado?

2.se debian aumentar el hardware pero tratando de mantenerlo simple y mientras el procesador ejecutaba sus subrutinas debía parar (HALT) la secuencialidad de las instrucciones y muchos componentes para solo dirigirse a los que necesitara en un momento en específico .

3.Esto debía suponer que el procesador sería un híbrido combinacional y secuencial que cuando leyera las “palabras ” correctas iniciará una secuencia y dejará de ser combinacional .

Se separó el desarrollo en 2 ,por un lado teníamos el cpu2 que podía hacer todo menos las subrutinas ; por otro teníamos que hacer una máquina de estados que hiciera las subrutinas y luego unirlos de alguna manera .

Para desarrollar la máquina de estados se pensó en un simple algoritmo tipo for-loop embebido en el hardware ; la maquina de estados estaría diseñada de tal forma que haría las veces de un algoritmo tipo for-loop para imprimir los valores .La lógica embebida en el hardware ,escrita en instrucciones MIPS , es la siguiente:

INIT:beq $s1 $s2 END

OUT ,$s1(base)

addi $s1,$s1,1

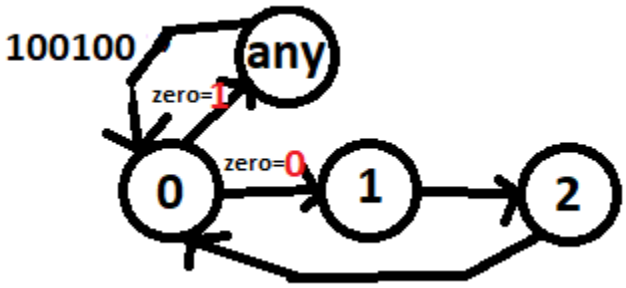
J INIT

END

Este pequeño algoritmo es lo que hace la máquina de estados finitos ;otra vez , esta lógica está embebida en el hardware .Se copió en mips solo a modo ilustrativo porque lo que importa es la lógica que conlleva y no tanto su representación aunque básicamente eso es lo que hace el hardware en cada paso de la rutina .

Lo que el algoritmo hace ,la subrutina , es chequear si el iterador es igual que el tope máximo del ciclo , si lo es finaliza y nuestro procesador vuelve a ser combinacional ; sino hace una salida desde la posición de memoria base +iterador finalmente le suma 1 al iterador y vuelve a la instrucción inicial. Estos datos los proveerá una instrucción como veremos más adelante.

Teniendo la lógica básica se aproximó al problema con un diagrama de burbujas en el que cada estado activaría las respectivas señales que necesitara para cumplir su función .Cada estado seria como una instrucción del codigo anterior ejecutando cosas diferentes .Inicialmente el estado ANY seria el estado por defecto en el que el procesador se mantendría ejecutando instrucciones combinacionales monociclo ;este seria el puente entre las dos partes del híbrido .Una vez llegado los parámetros correctos , detonaría la rutina .



Las primeras veces hubo un gran problema ya que el valor IN a la maquina de estados debia ser el propio opcode de la instrucción nueva (OUT) 100100 que sería el detonante para la rutina ,para que pasara de ANY a 0 , sin embargo esto creaba un gran problema como si falta un tiempo más de reloj para que funcionara ya que no se puede combinar el estado mismo de la máquina de estados para que este sea uno de sus inputs para cambiar de estado .El Estado 0 sería el mismo 100100 pero por estas dualidades se mantuvo separado . Se tomó otro camino.Ya que la rutina iba a ser un for-loop con n repeticiones se soluciono el problema de ingresar al arutina mediante una instrucción previa que prepara los datos necesarios para la instrucción OUT diciendo cuantas veces iba a estar en el loop (en la máquina de estados sería cuántas veces estaría repitiendo entre el estado 0 y 2).La instrucción es addi $5,$5,n en donde el hardware entendera esta

instrucción por defecto como el detonante de la siguiente instrucción OUT (si es que llega a seguir) .Esta fue una buena solución ya que igual se necesita primero saber cuantas veces va a dar salida de los valores de todas maneras por lo que anes de llamar a la instrucción out se debe usar esa addi para preparar al hardware para hacerlo ;igualmente se podría hacer la instrucción addi mencionada sin que siguiera la instruccion OUT.De esta manera para invocar la funcionalidad OUT se debe proceder así en código :

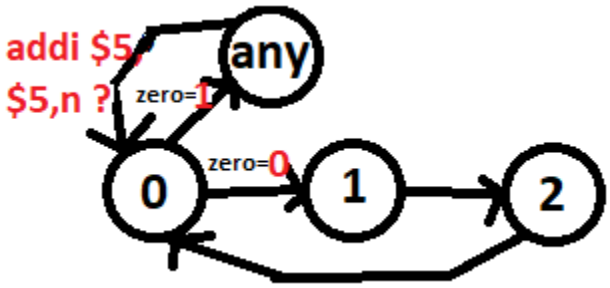
addi $5,$5,n-veces

OUT $5 ,$Riterador,base

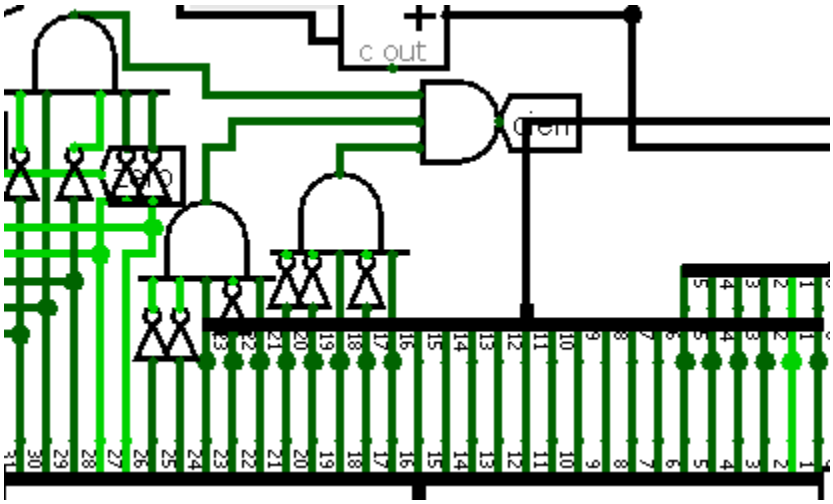
En donde el addi prepara los datos del tope en el registro 5 (por defecto se decidio asi).Luego OUT ejecuta ;esta tiene la siguiente forma:

El tope siempre será en OUT en el registro 5 , el iterador debe ser un registro que se sepa que en ese momento sea cero , la base es la posición de memoria desde donde se quiere empezar a contar.

Como se mencionó anteriormente , el problema se soluciono si antes del OUT la instrucción anterior es el addi para preparar los argumentos lo que dispararía la subrutina.Finalmente el algoritmo como se noto empieza con un BEQ .si son iguales el tope y el iterador , debe salir ,esto sucede cuando ejecuta la el beq en el estado cero y si la señal que recibe de la ALU ZERO es 0 significa que no son iguales y debe continuar la rutina ; si es 1 significa que son iguales y la rutina acabo ,retomando el estado ANY para tratar cualquier instrucción siguiente combinacional.

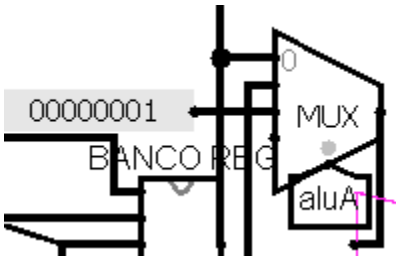


El hardware que se da cuenta de este trigger es simplemente un AND que reconoce el patrón de la instrucción anterior y le devuelve la señal al CPU llamada CIEN.



Cabe notar que una vez esta en el estado 1 ,ira automáticamente al estado 2 y se devolvera al 0 donde volvera a preguntar.La decision de si son iguales o no es monociclo y se da en un solo tic del reloj , en ese mismo tic salta al estado 1 o ANY según sea ZERO.

Para las salidas de los estados se tuvo que implementar un multiplexor extra en la entrada A de la ALU que le brindara varios valores diferentes , controlados por el CPU .



En el estado 0 de la máquina de estados finitos ,las salidas es un BEQ normal ; la única diferencia es que el procesador debe mandar dos bits 00 al MUX DE ALU-A para decirle que deje pasar el registro tope que sale de read data 1 para poderlo comparar con el registro iterador y determinar si sigue la rutina o no.Esta senal de dos bits se llama aluA.

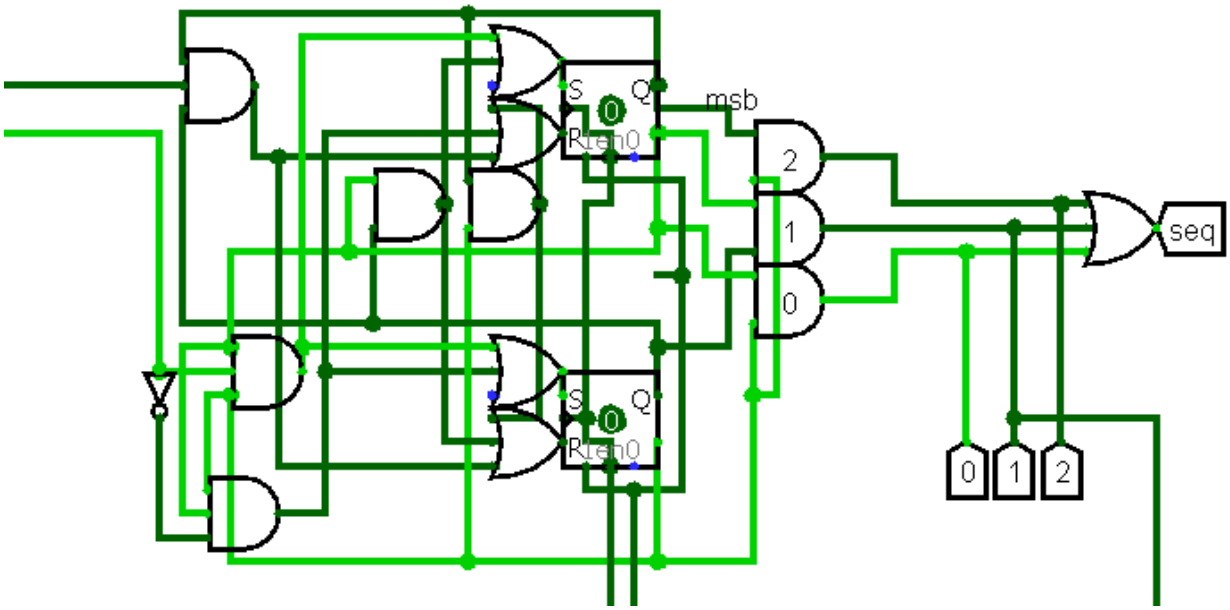
En el estado 1 las salidas del procesador son la señal típica para que entre el inmediato(base) a la ALU-B (ALUsrc=1) y senal 01 en el MUX de ALU-A para que entre el iterador que sale de Read data 2 y se sumen computando la dirección efectiva que luego es llevada a la memoria con la señal memRead=1 para que cuando salga el dato se lleve al dispositivo de salida y se le dé la señal del procesador OUT=1 para que muestre el

valor.

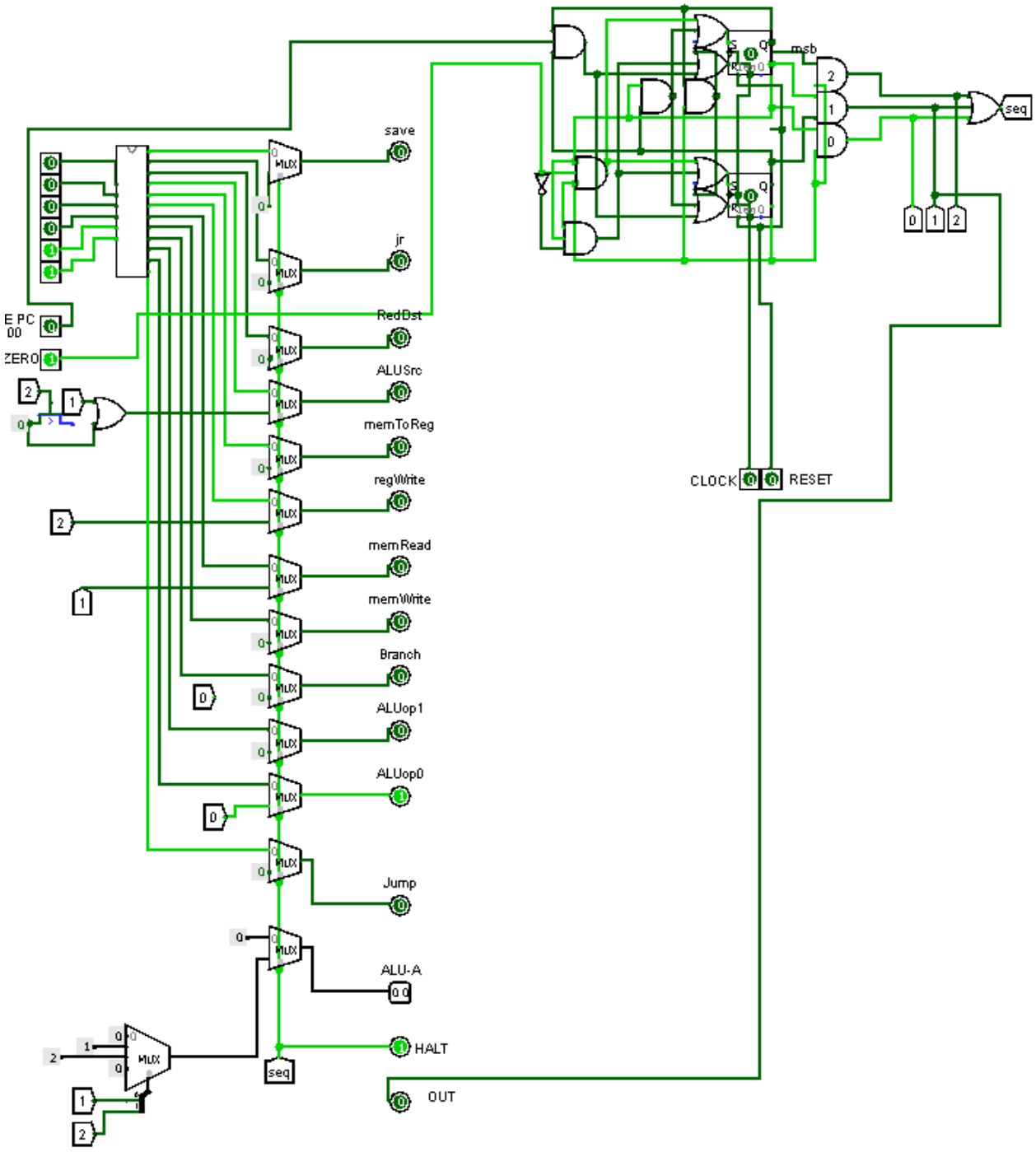
En el estado 3 las salidas son simplemente una suma inmediata para continuar con el próximo valor .En ALU-A la señal es 10 para que la constante 1 pase a la ALU , En la ALU -B se manda 0 al MUX para que el iterador pase normalmente.Posteriormente se suman y el valor es guardado en el registro iterador con regwrite=1.

Cabe aclarar que como se dijo antes ; durante la ejecución de las rutinas la lectura secuencial de instrucciones es parada de manera que los números donde consolidar los valores como el iterador están disponibles ahí mismo en los valores de la instrucción.

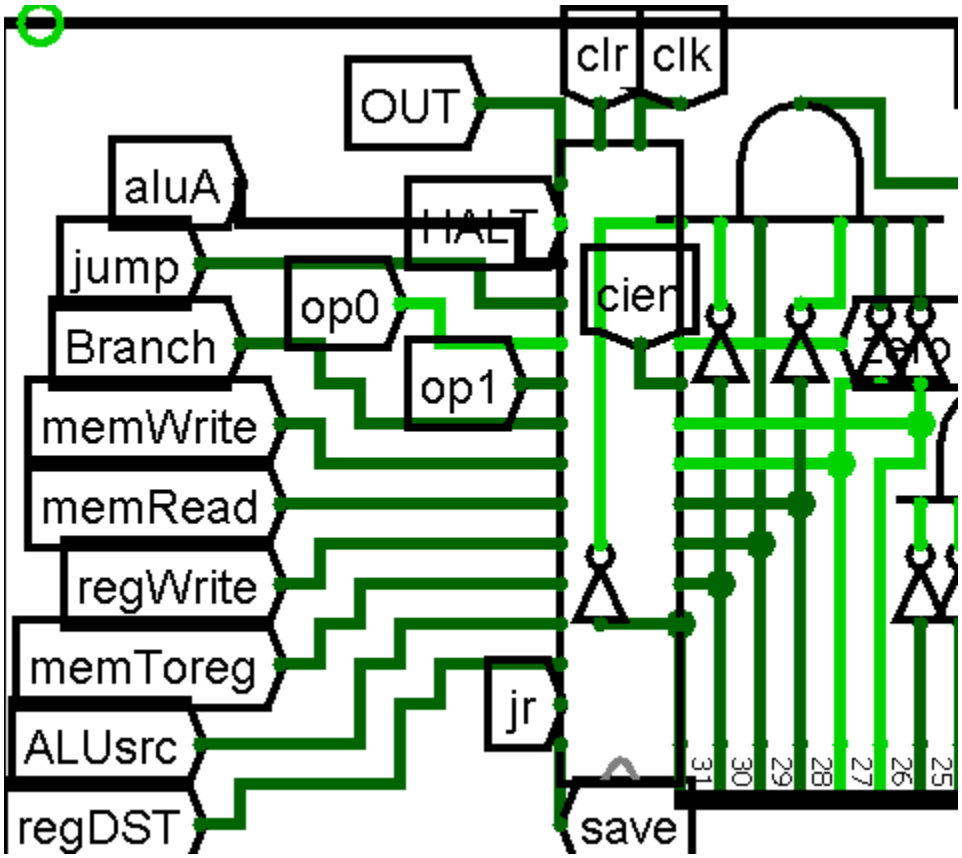
La máquina de estados finitos es :



Su estado ANY propiamente será el estado 3 .Ahora lo que falta es el puente entre estos dos tipos de circuito en un mismo procesador.Para este fin se utilizó el CPU 2 combinacional regido por la máquina de estados que lo inhabilitaba dadas las condiciones necesarias por medio de las señales de entrada al procesador CIEN y ZERO anteriormente explicadas.Dadas las condiciones el FSM saltaría al estado 0 inhabilitando el procesador combinacional (CPU2) por medio de MUXes en las salidas para el brindar sus salidas propias anteriormente descritas .Una vez la rutina esta completa ya que se llego al tope la máquina retorna al estado 3 y habilita al procesador combinacional para que siga tomando instrucciones normalmente .Esto constaría el CPU3 y quedaria asi:

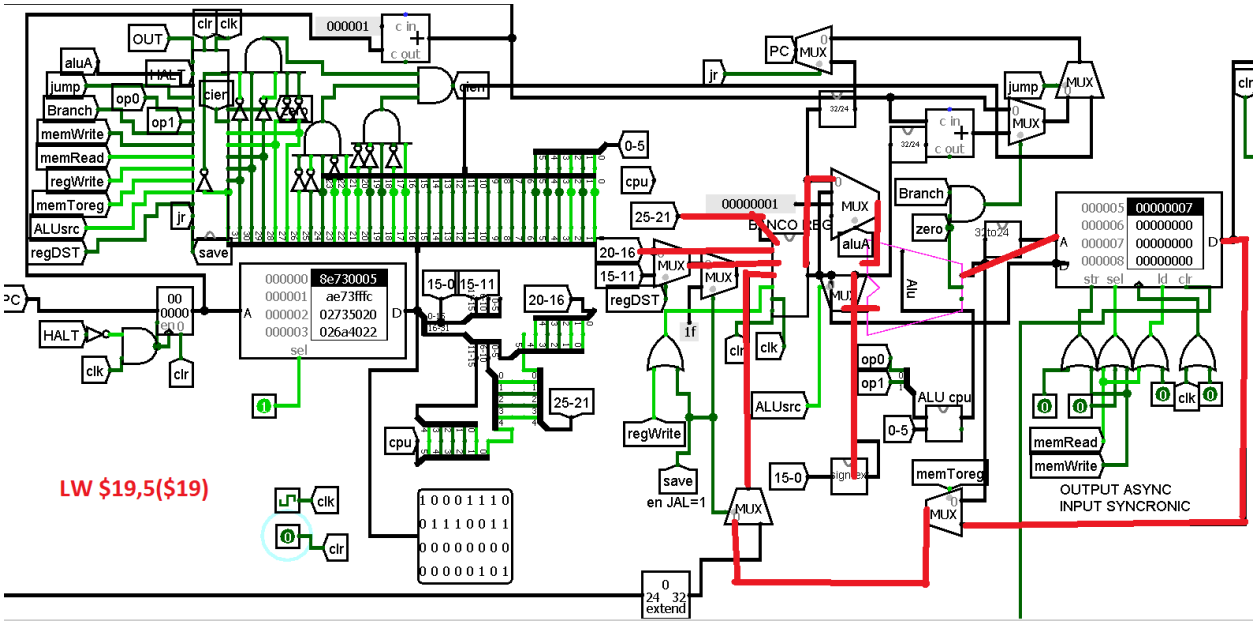


Conectado a la maquina se ve asi:

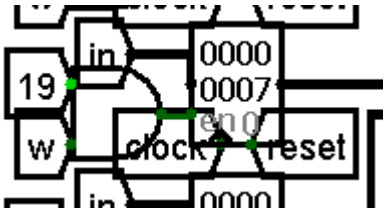

clr
clk
OUT
aluA
HALT
jump
op0
cier
Branch
op1
memWrite
memRead
regWrite
memToreg
jr
ALUsrc
regDST
save
31
30
29
28
27
26
25

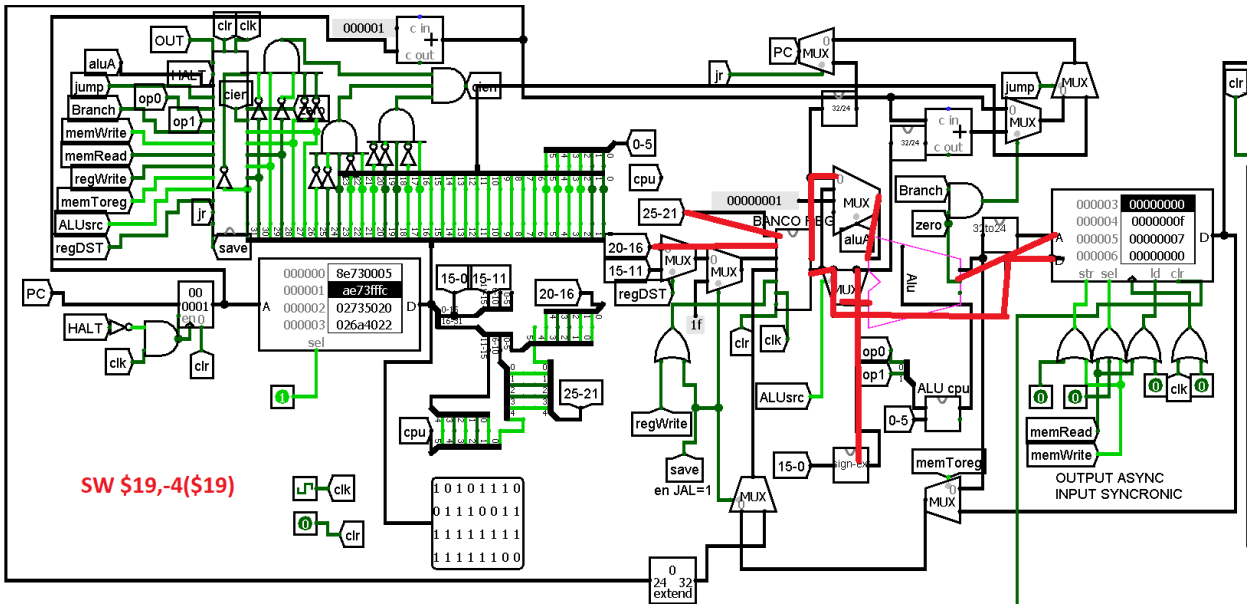
EJECUCIÓN DE INSTRUCCIONES

A Continuación se muestran los caminos de los datos según la instrucción , y el resultado que generan.Este codigo se ejecuto teniendo en la memoria valores F en 0x4 y 7 EN 0x5 y todos los registros inicialmente en 0.

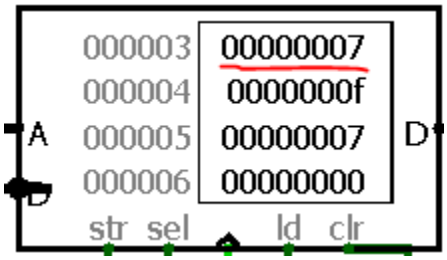


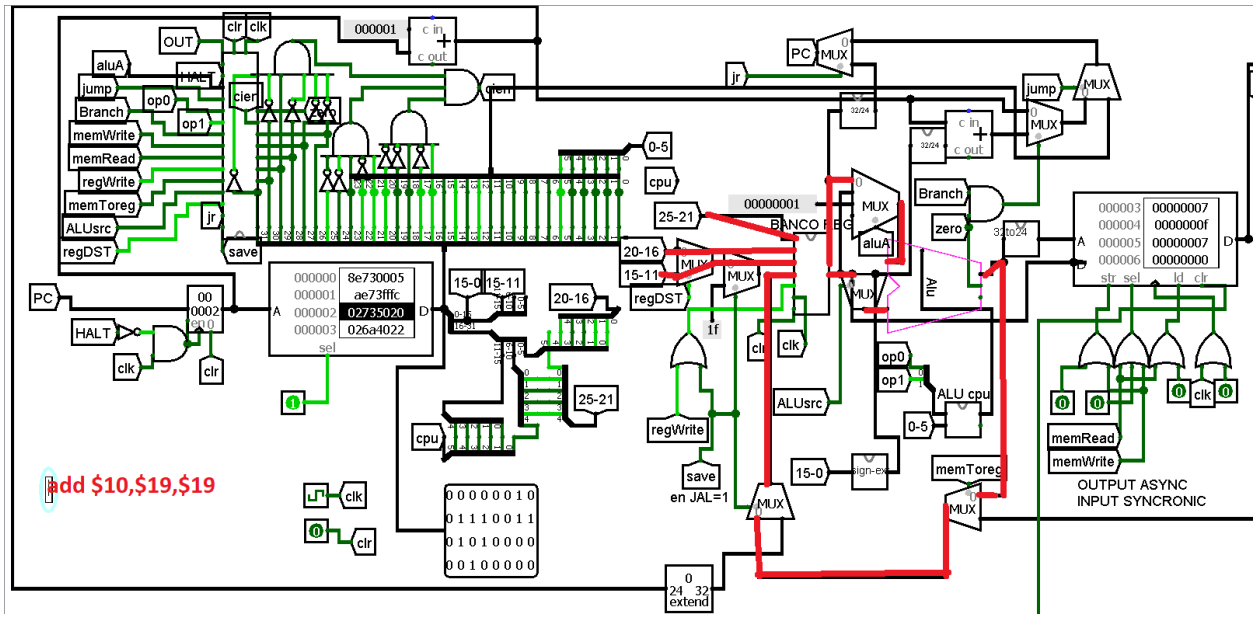
Esta ejecución produce :



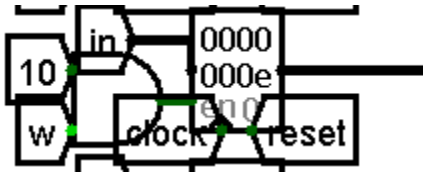


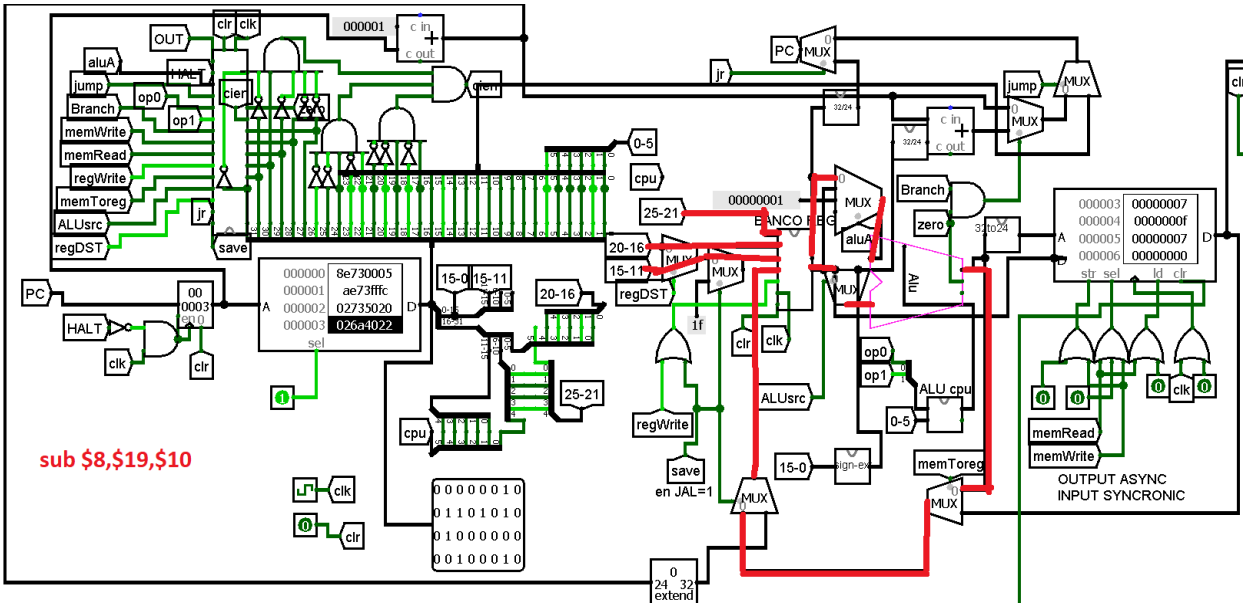
Esta ejecución produce:



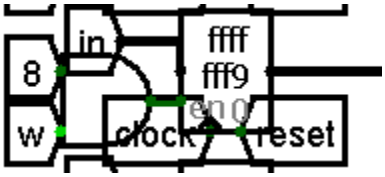


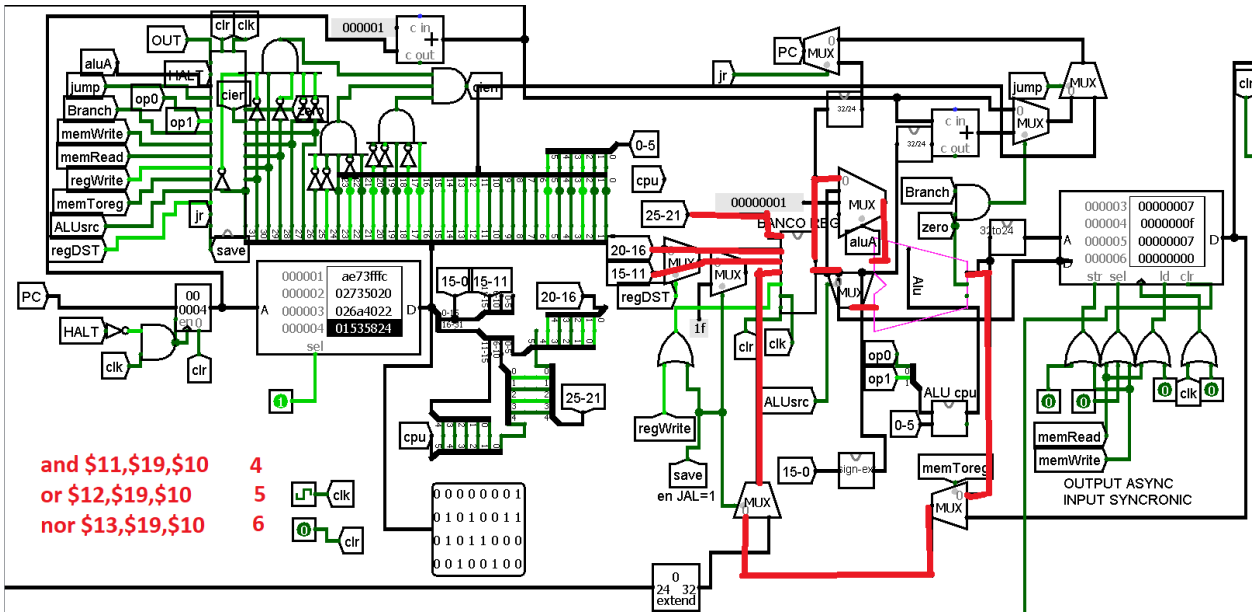
Este ejecucion produce:



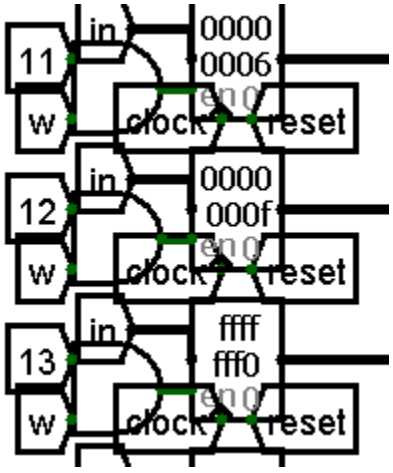


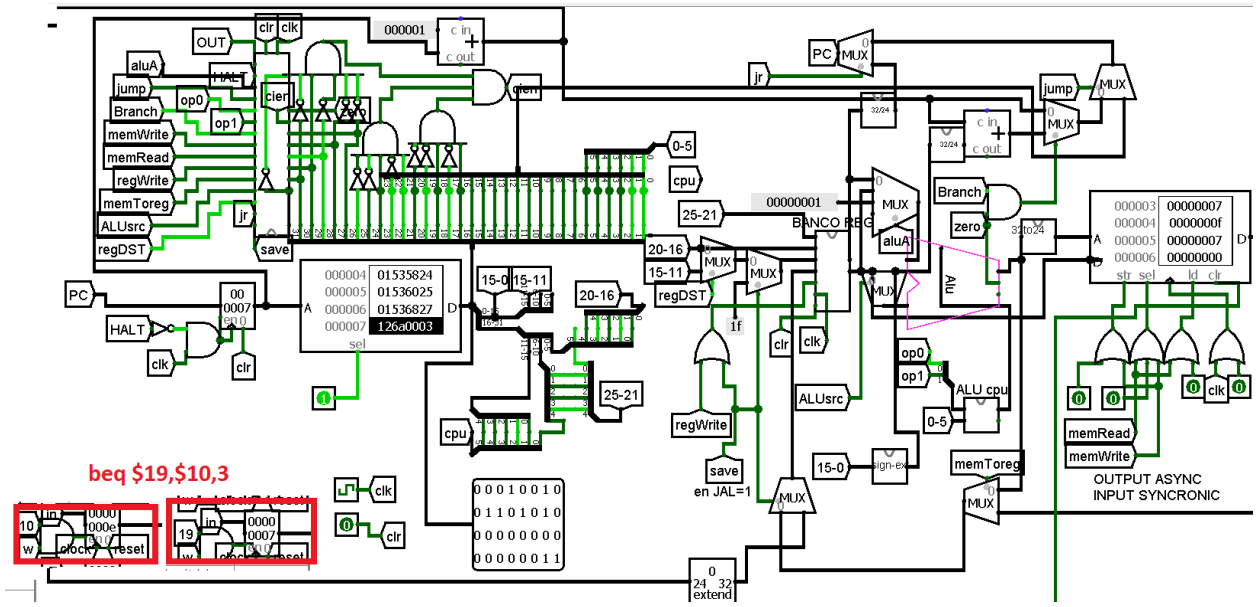
Esta ejecución produce:



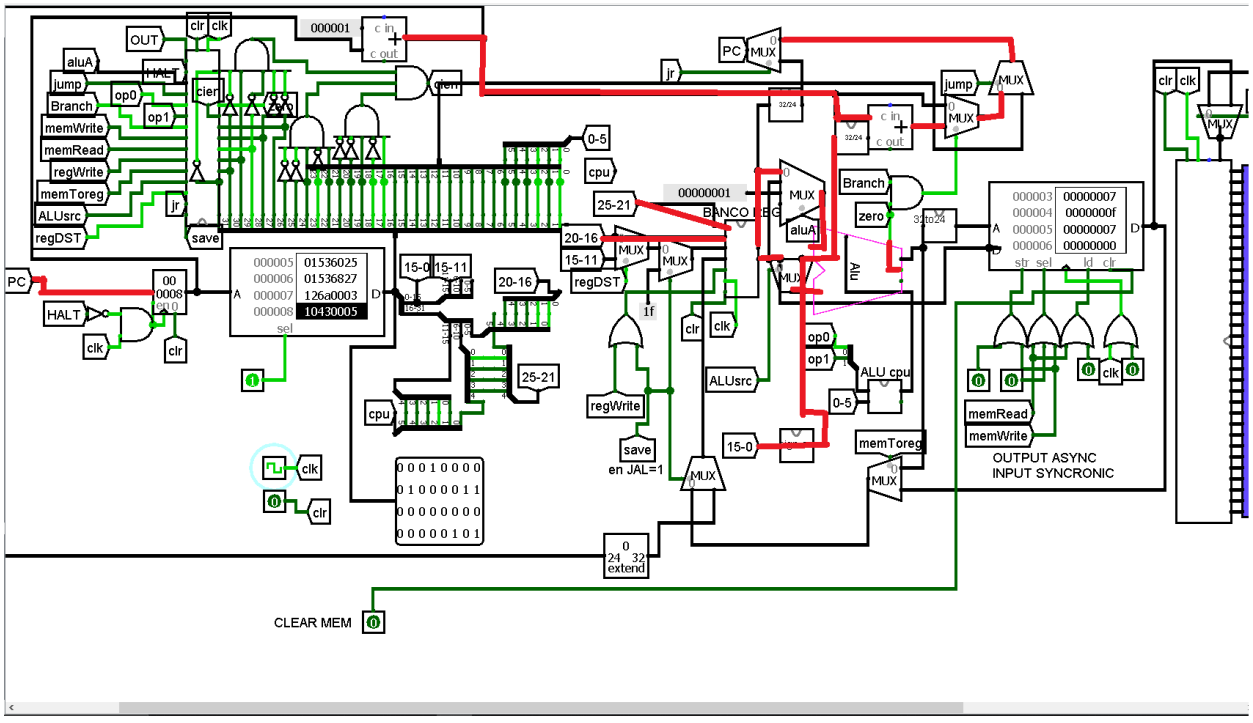


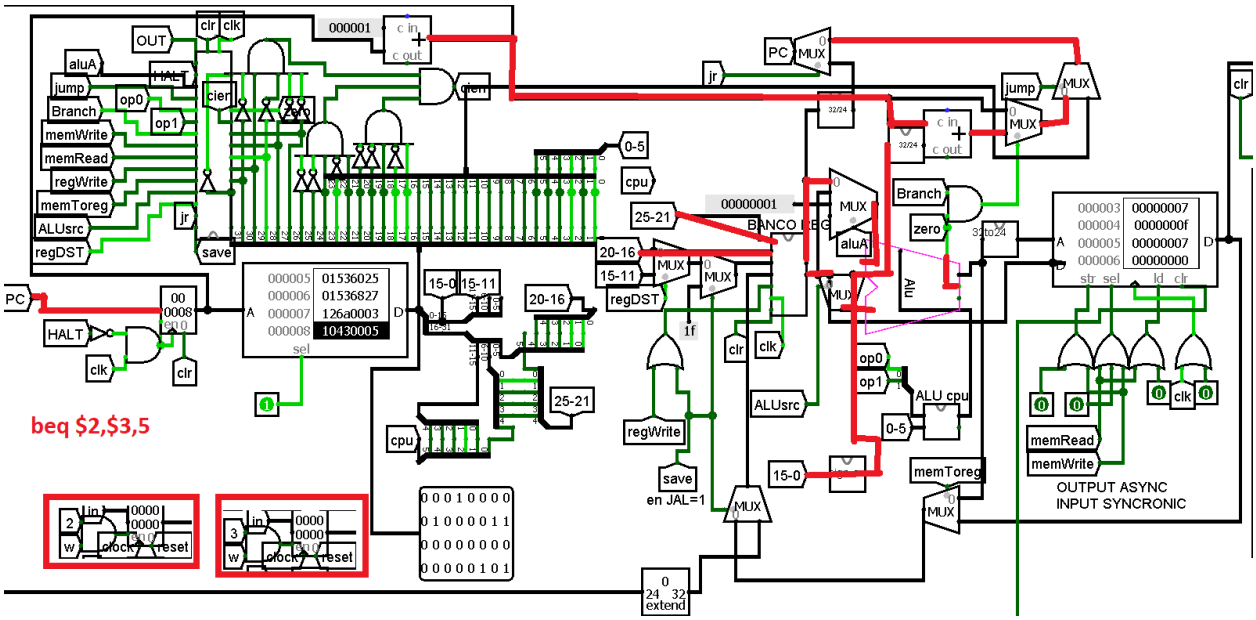
Estas tres ejecuciones producen:



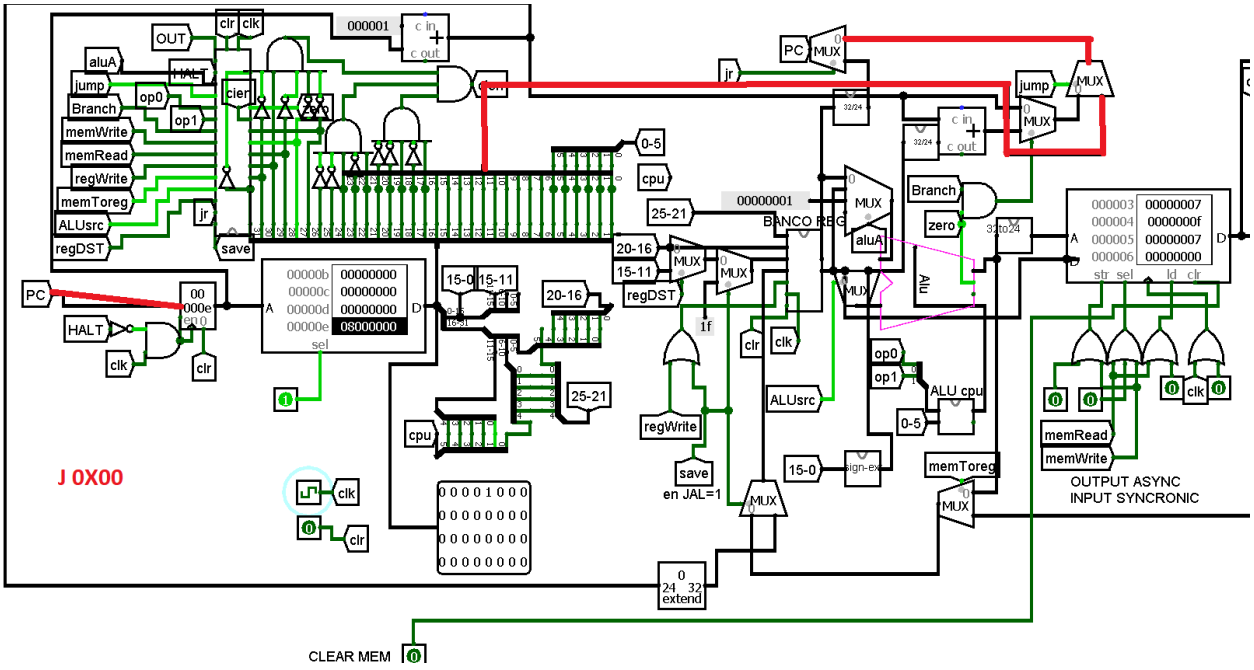


Este resultado es secuencial pues los registros son diferentes:

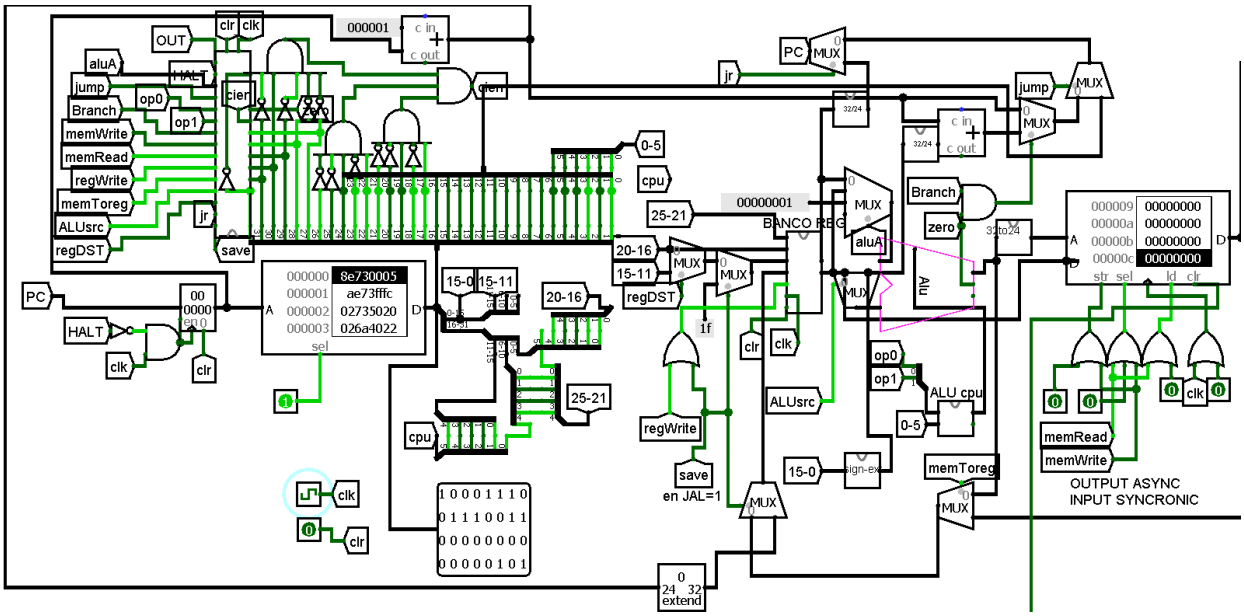




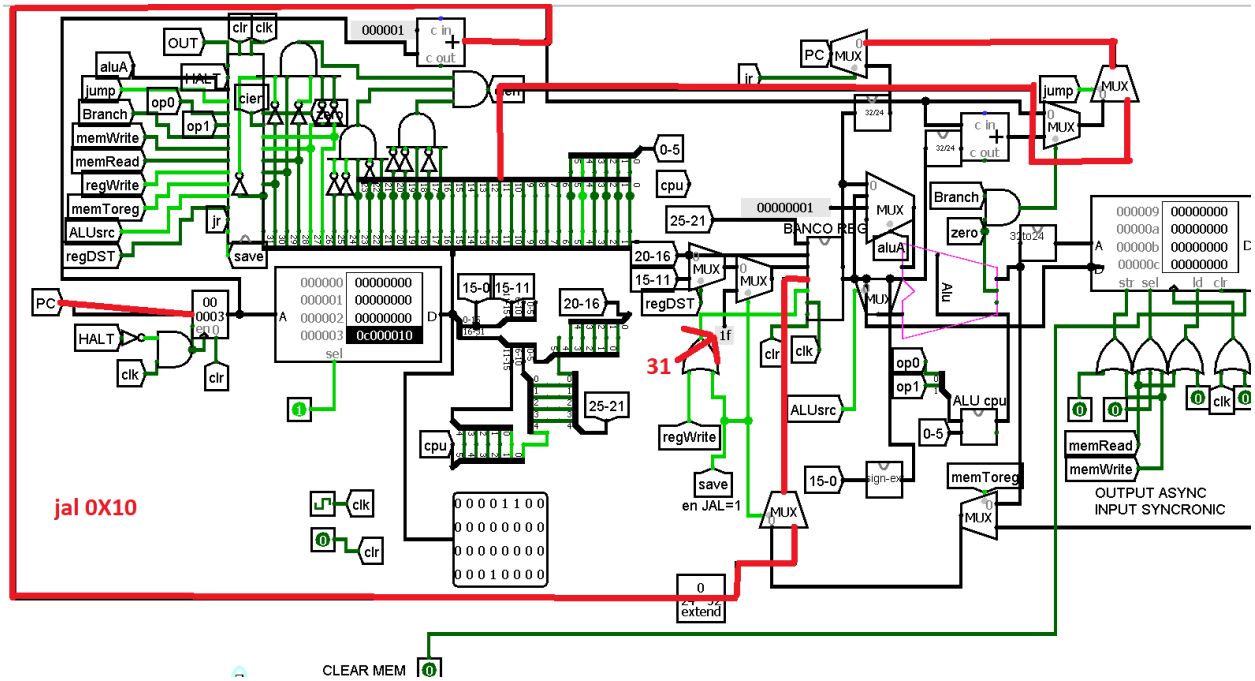
Este resulta en un salto a (8+9)+5=E. Luego en E tenemos:



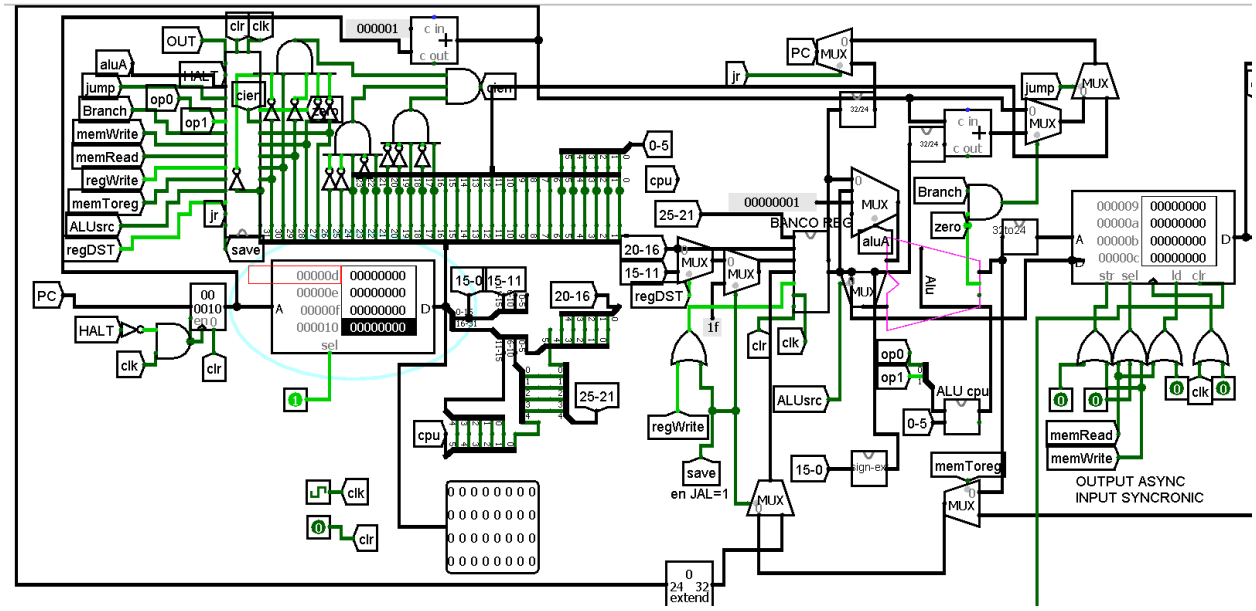
Que nos devuelve a 0:



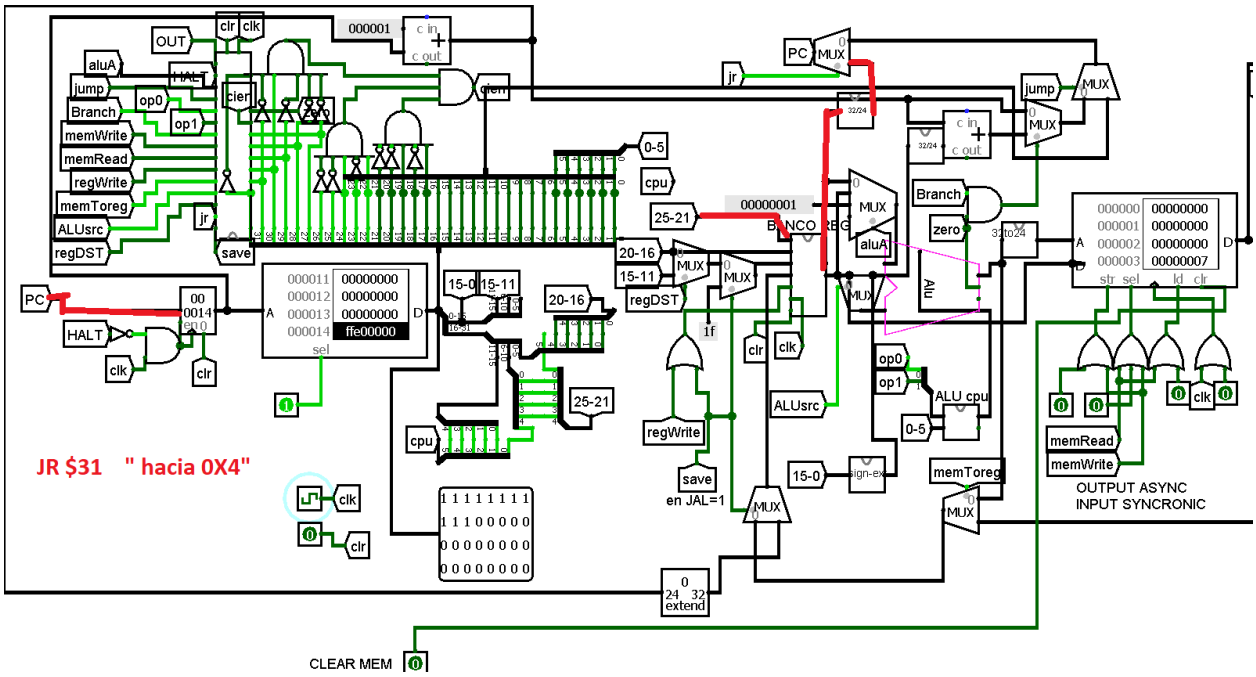
Ahora cargamos otras instrucciones para evaluar el JAL y JR:



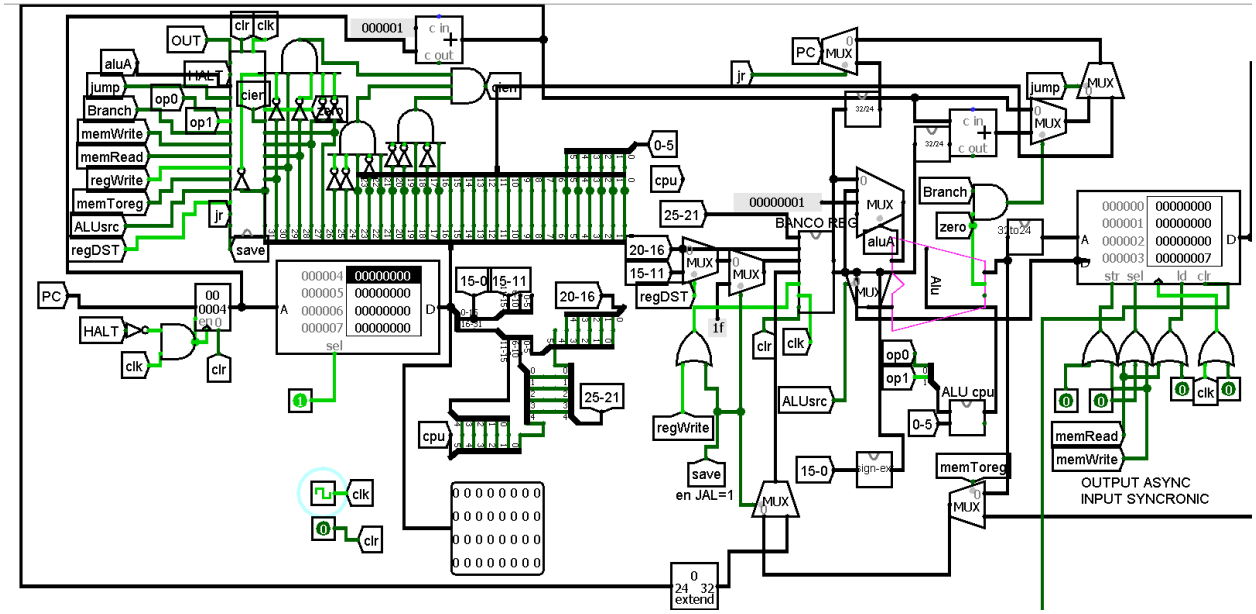
Saltando a 0x10 :



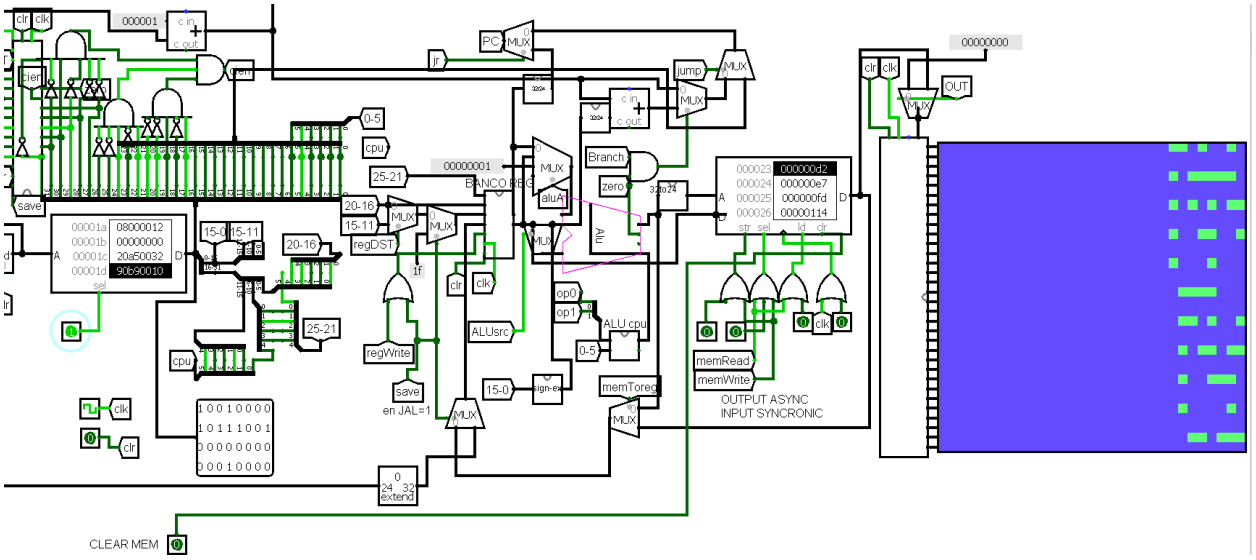
Avanzamos hasta 0x14 en donde esta la siguiente instrucción:

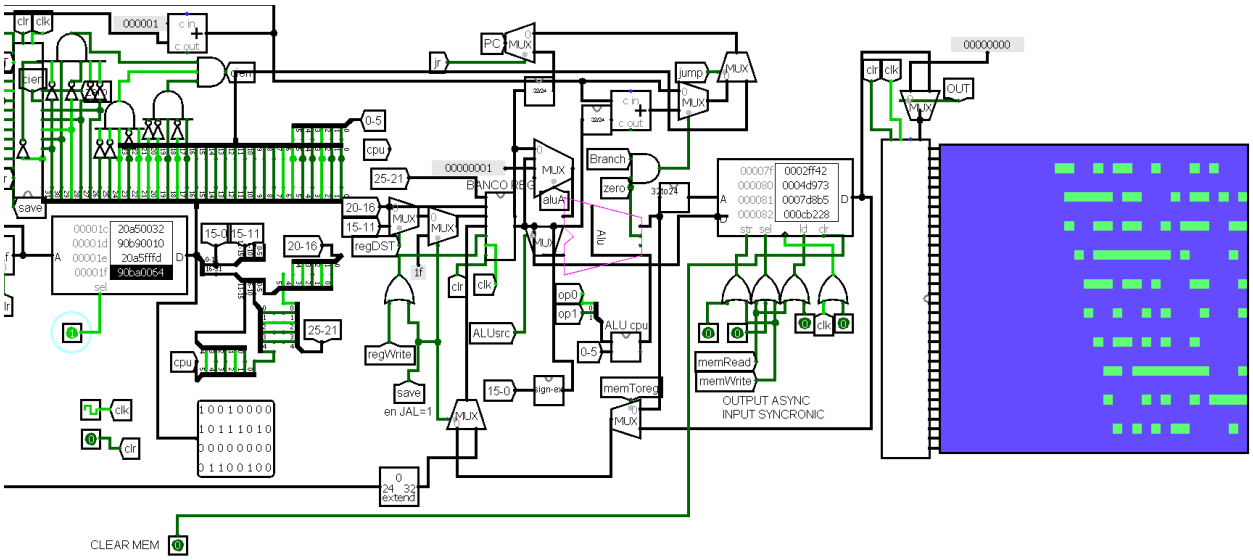


y nos devolvemos a 0x3:



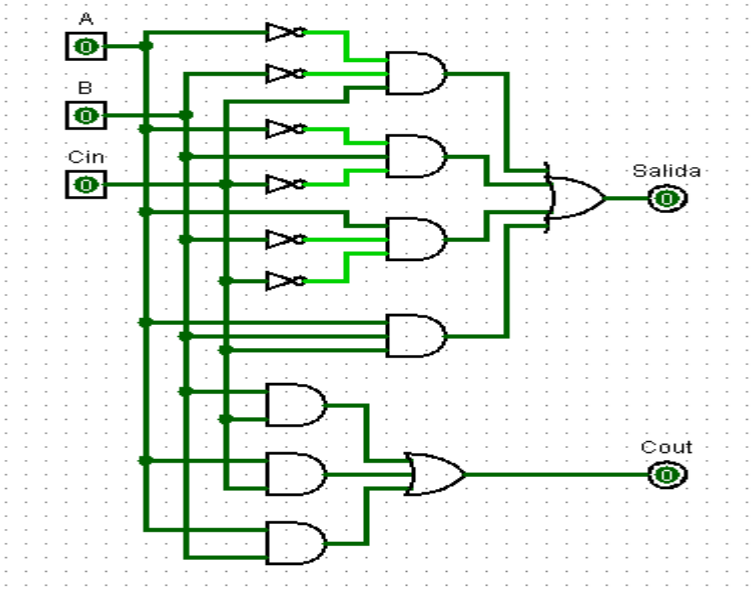
Finalmente se muestra la ejecución de la función OUT mostrando números triangulares que calculo en la pantalla y luego mostrando los fibonacci:



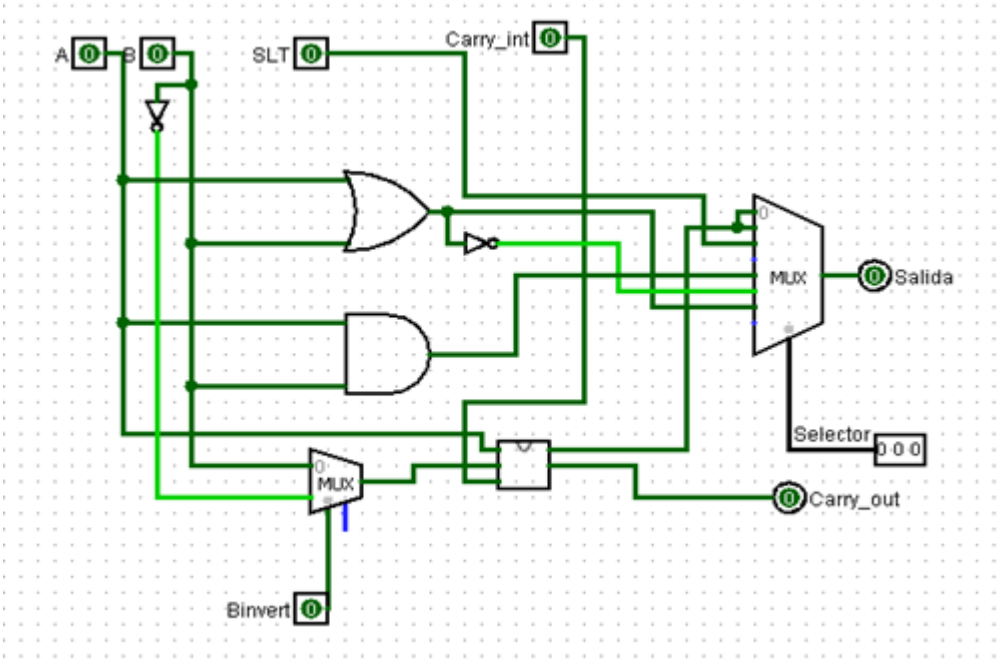


ALU

para comenzar con este componente lo primero que tuvimos que hacer fue un sumador, este lo necesitábamos para poder construir un segundo componente muy importante que es la ALU de un bit, escogimos uno que ya habíamos trabajado el laboratorio pasado.



Ya teniendo este componente continuamos hacer la ALU de un bit, este lo construimos de acuerdo con la codificación que nos tocó, pusimos unas entradas, una OR y AND con sus negadas, y lo fuimos conectando al sumador y un multiplexor de acuerdo con la codificación, una vez conectado este, nos tenía que dar en el resultado nuestra codificación

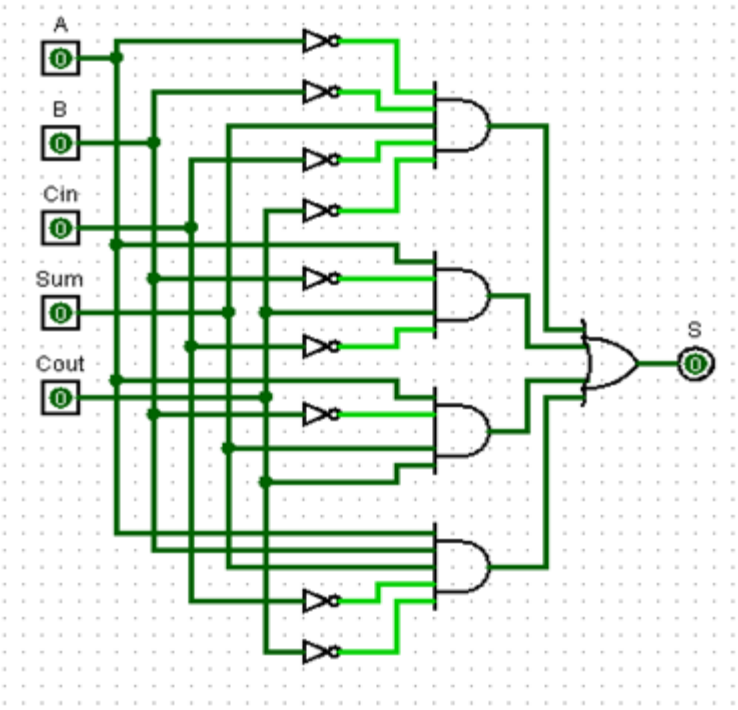


Una vez terminado la ALU de un bit, hicimos otro circuito de lo mismo componente anteriores, pero estas con un componente adicional, otra salidas llamada OverFlow y a la Alu de un bit le agregamos esa salida y también un componente llamado signo, el circuito resultante lo llamamos, sumador y Alu con overflow, para crear este signo tuvimos que sacar una tabla y sus respectivos mapa Karnaugh.

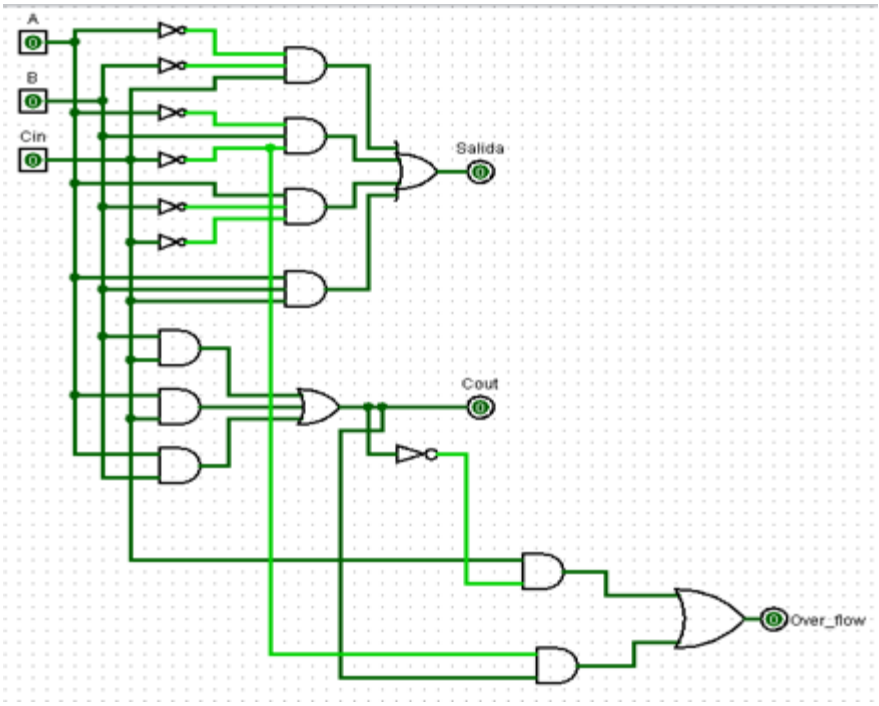
A	B	Cin	Sum	Cout	S
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	1
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

	Sum/Cout		~A				A	
B/cin				1		1	1	
							1	
								1

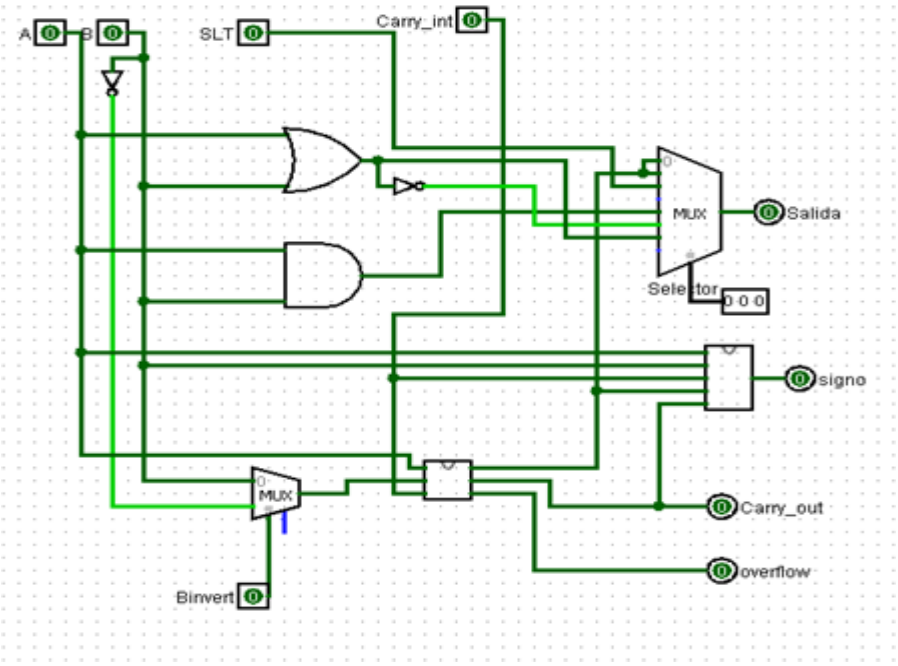
~A B sum ~cin ~cout + A ~B cout ~cin + A ~B sum cout + A B sum ~cin ~cout



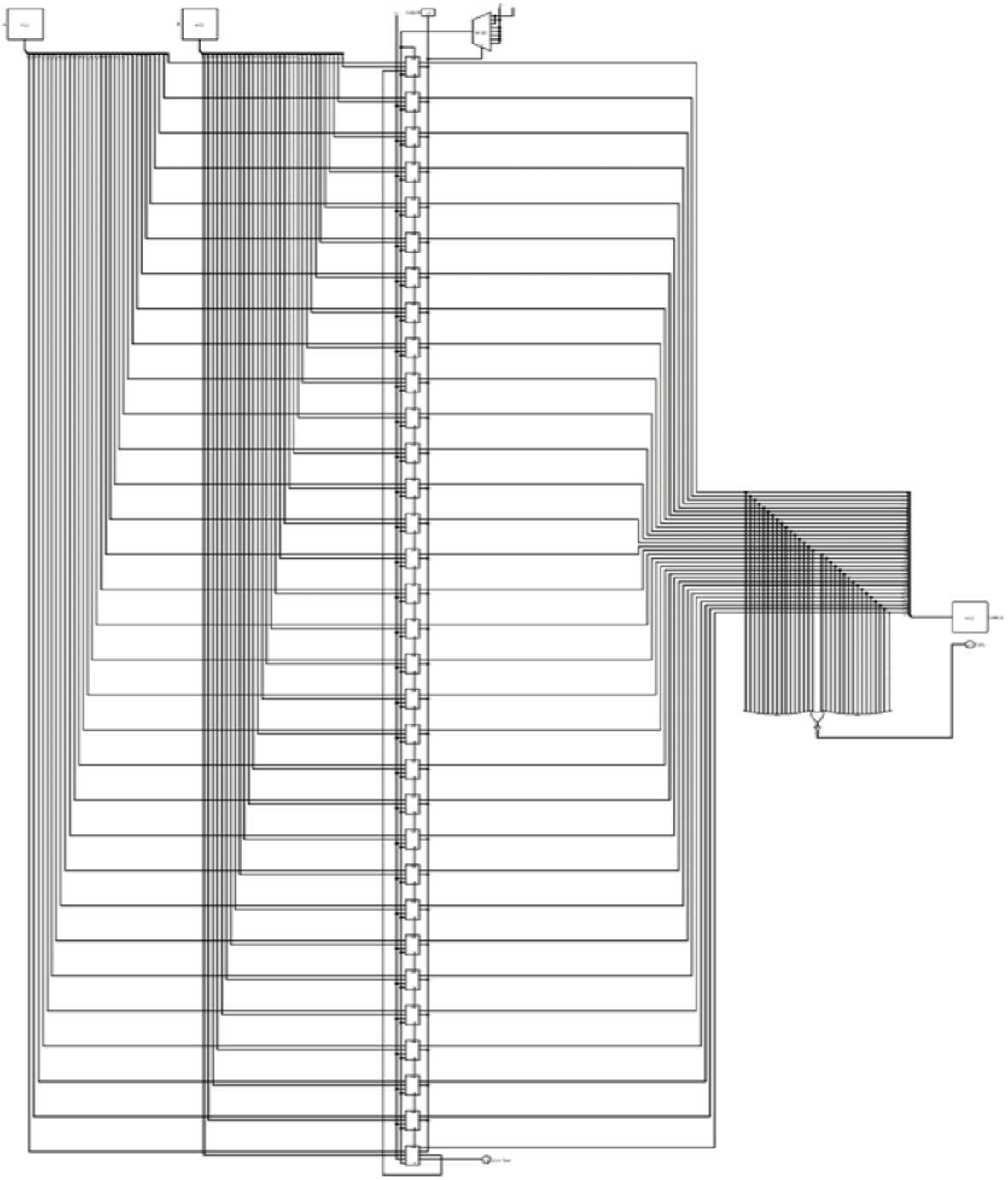
Sumador overflow



Alu de un bit con overflow



Una vez hecho todos estos componentes, continuamos con la Alu de 32 bits, para esto unimos 31 veces la Alu de un bit sucesivamente y la última Alu de un bit con overflow. Esta Alu de 32 bits, tiene dos entradas de 32 bits, una de 3 bits, una constante, una salida de 32 bits, la salida zero, un multiplexor, la imagen aparece blanco porque la exportamos directamente de logisim para que se viera completa.



ALU CONTROL

Para crear la Alu control, nos basamos en la siguiente tabla, que habíamos visto en clase.

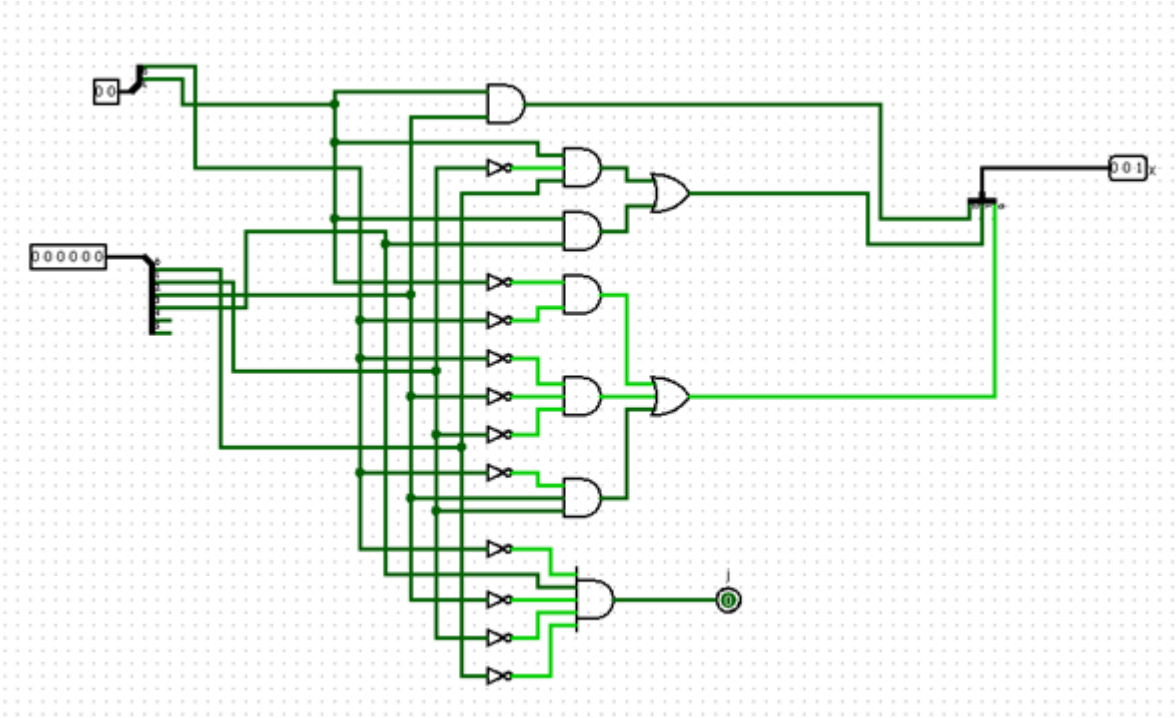
funck						alu function	alu control		
x	x	x	x	x	x	add	0	0	1
x	x	x	x	x	x	add	0	0	1
x	x	x	x	x	x	subtract	0	0	0
1	0	0	0	0	0	add	0	0	1
1	0	0	0	1	0	subtract	0	0	0
1	0	0	1	0	0	AND	1	0	0
1	0	0	1	0	1	OR	1	1	0
1	0	1	0	1	0	SLT	0	1	0
0	0	1	0	0	0	JUMP			
1	0	0	1	1	1	NOR	1	0	1

Teniendo en cuenta nuestra codificación y la tabla anterior, sacamos esta tabla gigantesca

a	b	c	d	E	F	x	y	z	j
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	1	0
0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	1	0	0	1	0
0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	1	0	0	1	0
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	1	1
0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	0	0	1	0
0	0	1	0	1	1	0	0	1	0
0	0	1	1	0	0	0	0	1	0
0	0	1	1	0	1	0	0	1	0
0	0	1	1	1	0	0	0	1	0
0	0	1	1	1	1	0	0	1	0
0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0	0
0	1	0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0
0	1	0	1	0	1	0	0	0	0
0	1	0	1	1	0	0	0	0	0
0	1	0	1	1	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0
0	1	1	0	1	0	0	0	0	0
0	1	1	0	1	1	0	0	0	0
0	1	1	1	0	0	0	0	0	0
0	1	1	1	0	1	0	0	0	0
0	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	1	0

1	0	0	0	0	1	X	X	X	0
1	0	0	0	1	0	0	0	0	0
1	0	0	0	1	1	X	X	X	0
1	0	0	1	0	0	1	0	0	0
1	0	0	1	0	1	1	1	0	0
1	0	0	1	1	0	X	X	X	0
1	0	0	1	1	1	1	0	1	0
1	0	1	0	0	0	X	X	X	0
1	0	1	0	0	1	X	X	X	0
1	0	1	0	1	0	0	1	0	0
1	0	1	0	1	1	X	X	X	0
1	0	1	1	0	0	X	X	X	0
1	0	1	1	0	1	X	X	X	0
1	0	1	1	1	0	X	X	X	0
1	0	1	1	1	1	X	X	X	0
1	1	0	0	0	0	X	X	X	0
1	1	0	0	0	1	X	X	X	0
1	1	0	0	1	0	X	X	X	0
1	1	0	0	1	1	X	X	X	0
1	1	0	1	0	0	X	X	X	0
1	1	0	1	0	1	X	X	X	0
1	1	0	1	1	0	X	X	X	0
1	1	0	1	1	1	X	X	X	0
1	1	1	0	0	0	X	X	X	0
1	1	1	0	0	1	X	X	X	0
1	1	1	0	1	0	X	X	X	0
1	1	1	0	1	1	X	X	X	0
1	1	1	1	0	0	X	X	X	0
1	1	1	1	0	1	X	X	X	0
1	1	1	1	1	0	X	X	X	0
1	1	1	1	1	1	x	x	x	0

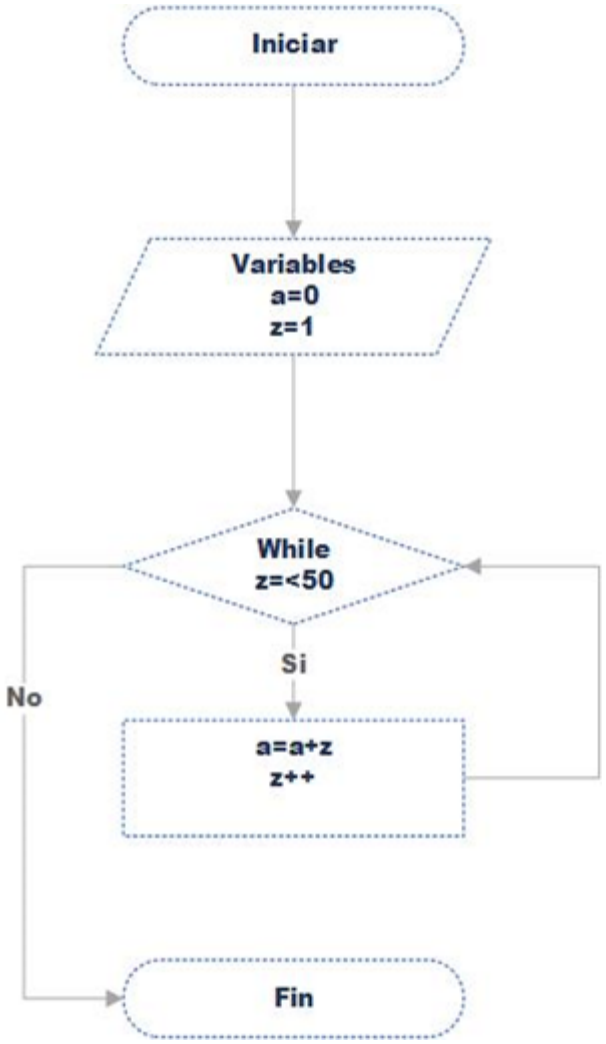
Como nos dio una tabla tan grande, y en los laboratorios pasado ya hemos demostrado que sabemos sacar los más de Karnaugh, decidimos usar la herramienta de logisim para crear el circuito directamente, porque eran demasiados mapas y el tiempo lo podíamos usar en otro componente que estábamos quedado, la siguiente foto ya es el circuito como tal implementado.



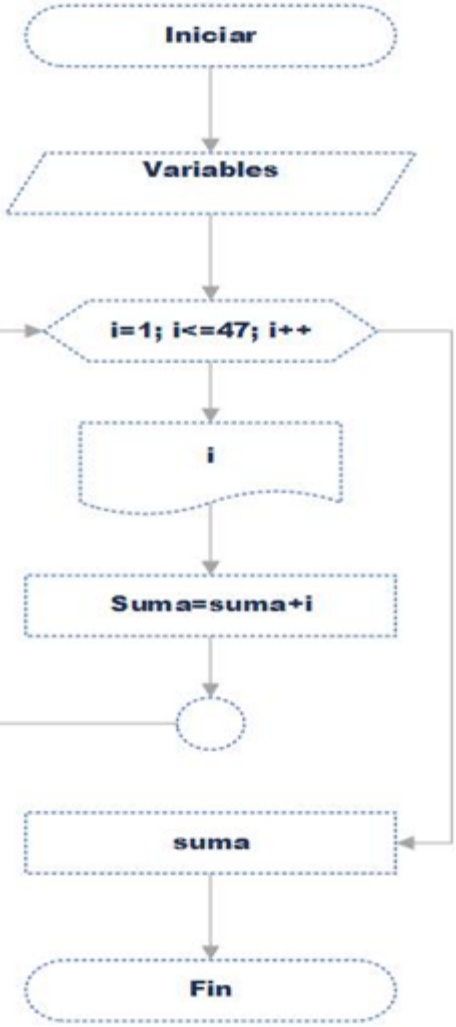
ALGORITMOS

Los algoritmos que nos tocaron fueron el “Fibonacci” y “Los números triangulares” para comenzarlo hacer primero hicimos los diagrama de flujo de cada uno para poder tener una base cuando comencemos la codificación.

Números Triangulares



Fibonacci



Una vez esto hecho, continuamos a hacer los código en el Mars, el cual hicimos en un principio cada uno individual, pero una vez teníamos una base de los dos, decidimos hacer un solo proyecto los dos juntos, para poder mostrar los dos códigos en nuestra máquina.

```
.data
vector: .space 20
.text

        main:
        jal NTriangulares
        jal fib

        NTriangulares:
                addi    $s0,$s0,0
                addi    $s1,$s1,1
                addi    $s3,$s3,0

        while:
                li      $t0,50
                beq     $s1,$t0, end
                add     $s0,$s0,$s1
                sw      $s0,vector($s3)
                addi    $s3,$s3,4
                addi    $s1,$s1,1

                j       while
        end:

        jr      $ra



    fib:
            addi     $t0, $zero, 47
            addi     $t1, $zero, 1
            addi     $t2, $zero, 0
            addi    $t3, $zero, 1
            addi    $s0, $zero, 0
    for:
            beq     $t3, $t0, salir
            addi     $a0, $t2, 0
            sw      $a0, vector($s0)
            add     $t4, $t2, 0
            add     $t2, $t2, $t1
            addi    $t1, $t4, 0
            addi    $t3, $t3, 1
            addi    $s0, $s0, 4
            j for
    salir:
```

Una vez listo el código, continuamos a desglosarlo para pasarlo a binario y después el binario a hexadecimal.

este primero es el código que usamos para ir probando la máquina

Tipo I	Opcode	rs	rt	immediate			Binario	Hexa
lw $19,5($19)	100011	10011	10011	0000000000000101			10001110011100110000000000000101	8E730005
sw $19,-4,($19)	101011	10011	10011	1111111111111100			10101110011100111111111111111100	AE73FFFC
Tipo R	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**		
add $10,$19,$19	000000	10011	10011	01010	00000	100000	00000010011100110101000000100000	2735020
sub $8,$19,$10	000000	10011	01010	01000	00000	100010	00000010011010100100000000100010	26A4022
and $19,$10	000000	01010	10011	01011	00000	100100	00000001010100110101100000100100	1535824
or $12,$19,$10	000000	01010	10011	01100	00000	100101	00000001010100110110000000100101	1536025
Nor $13,$19,$10	000000	01010	10011	01101	00000	100111	00000001010100110110100000100111	1536827
Tipo I	**Opcode**	**rs**	**rt**	**immediate**				
beq $19,$10,3	000100	10011	01010	0000000000000011			00010010011010100000000000000011	126A0003
beq $2,$3,5	000100	00010	00011	0000000000000101			00010000100001100000000000000101	10430005
beq $2,$3,-5	000100	00010	00011	1111111111111011			00010000100001111111111111111011	1043FFFB
Tipo j	**Opcode**	**address**						
j	000010	00000000000000000000000000					00001000000000000000000000000000	8000000

Y este si son los algoritmos que nos tocó y mostramos anteriormente.

Tipo j	Opcode	address						Binario	Hexa
jal NTriangulares	000011	00000000000000000000000010					0	00001100000000000000000000000010	C000002
jal fib	000011	00000000000000000000001101					1	00001100000000000000000000001101	C00000D
Tipo I	**Opcode**	**rs**	**rt**	**immediate**					
addi $s0,$s0,0	001000	10000	10000	0000000000000000			2	00100010000100000000000000000000	22100000
addi $s1,$s1,1	001000	10001	10001	0000000000000001			3	00100010001100010000000000000001	22310001
addi $s3,$s3,0	001000	10011	10011	0000000000000000			4	00100010011100110000000000000000	22730000
addi $t0,$t0,50	001000	01000	01000	0000000000110010			5	00100001000010000000000000110010	2108 0032
beq $s1,$t0, end	000100	10001	01000	0000000000000101			6	00010010000101000000000000000101	12280005
Tipo R	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**			
add $s0,$s0,$s1	000000	10000	10001	10000	00000	100000	7	00000010000100011000000000100000	2118020
Tipo I	**Opcode**	**rs**	**rt**	**immediate**					
sw $s0,vector($s3)	101011	10011	10000	0000000000010000			8	10101110011100000000000000010000	AE700010
addi $s3,$s3,1	001000	10011	10011	0000000000000001			9	00100010011100110000000000000001	22730001
addi $s1,$s1,1	001000	10001	10001	0000000000000001			10	00100010001100010000000000000001	22310001
Tipo j	**Opcode**	**address**							
j while	000010	00000000000000000000000110					11	00001000000000000000000000000110	8000006
Tipo R	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**			
jr $ra	111111	11111	00000	00000	00000	000000	12	11111111111000000000000000000000	FFE00000

Tipo I	Opcode	rs	rt	immediate					
addi $t0, $zero, 47	001000	00000	01000	0000000000101111			13	00100000000010000000000000101111	2008002F
addi $t1, $zero, 1	001000	00000	01001	0000000000000001			14	00100000000010010000000000000001	20090001
addi $t2, $zero, 0	001000	00000	01010	0000000000000000			15	00100000000010100000000000000000	200A0000
addi $t3, $zero, 1	001000	00000	01011	0000000000000001			16	00100000000010110000000000000001	200B0001
addi $s0, $zero, 0	001000	00000	10000	0000000000000000			17	00100000000100000000000000000000	20100000
Tipo I	Opcode	rs	rt	immediate					
beq $t3, $t0, salir	000100	01011	01000	0000000000001000			18	00010001011010000000000000001000	11680008
Tipo I	Opcode	rs	rt	immediate					
addi $a0, $t2, 0	001000	01010	00100	0000000000000000			19	00100001010001000000000000000000	21440000
Tipo I	Opcode	rs	rt	immediate					
sw $a0, vector($s0)	101011	10000	00100	0000000001100100			20	10101110000001000000000001100100	AE040064
Tipo I	Opcode	rs	rt	immediate					
addi $t4, $t2, 0	001000	01010	01100	0000000000000000			21	00100001010 0110000000000000000	214C0000
Tipo R	Opcode	rs	rt	rd	shamt	funct			
add $t2, $t2, $t1	000000	01010	01001	01010	00000	100000	22	00000001010010010101000000100000	1495020
Tipo I	Opcode	rs	rt	immediate					
addi $t1, $t4, 0	001000	01100	01001	0000000000000000			23	00100001100010010000000000000000	21890000
addi $t3, $t3, 1	001000	01011	01011	0000000000000001			24	00100001011010110000000000000001	216B0001
addi $s0, $s0, 1	001000	10000	10000	0000000000000001			25	00100010000100000000000000000001	22100001
Tipo j	Opcode	address							
j for	000010	00000000000000000000010010					26	00001000000000000000000000010010	8000012

Componente extra

addi $5,$5,50	0 01000	0 0101	0 0101	0 0000000000110010			28		20A50032
	OPCODE	TOPE	ITERADOR	BASE(16BITS)					
OUT $5,$25,0x10	100100	0 0101	11001	0000000000 010000			29		90B90010
addi $5,$5,-3	0 01000	0 0101	0 0101	1,11111E+15			30		20A5FFFD
	OPCODE	TOPE	ITERADOR	BASE(16BITS)					
OUT $5,$26,0x64	100100	0 0101	11010	1100100			31		90BA0064

CONCLUSIONES:

El trabajo fue muy ilustrativo para poder conocer finalmente cómo funciona un computador terminado .Nos dimos cuenta de que el CPU es no más que un manejador de recursos del computador ; que ejecuta de manera ordenada y sistemática ,en este caso monociclo lo que supone un circuito combinacional,que da sentido a unos símbolos 1 o 0 que en otro caso no tendrían sentido ; los convierte en información .La función extra fue difícil de diseñar debido a que se tuvo que concebir un híbrido de procesador;pero es muy curioso pensar en un computador como una orquesta muy organizada donde todo en cada , cada músico sabe que tocar y cuando generando una armonía y haciendo que las cosas funciones .Finalmente se puede concebir la funcionalidad de las partes para generar un todo que en este caso si es el resultado final , output , aunque es igual de importante e impresionante el desarrollo de la canción o programa mientras se ejecuta.