Arquitectura de Computadores y Laboratorio – 2508716

Departamento de Ingeniería de Sistemas

Facultad de Ingeniería

Universidad de Antioquia

Antonio González Restrepo

Vanessa Tocasuche Ochoa

Laboratorio 4: Diseño de Procesador Monociclo

Planteamiento

En el siguiente trabajo se encontrará el diseño detallado de un procesador monociclo en una versión reducida de la arquitectura MIPS32 en la herramienta Logisim Evolution. Este procesador soportará las instrucciones **load word (lw)**, **store word (sw)**, **add**, **sub**, **and**, **or**, **nor**, **set-on-less-than (SLT)**, **branch if equal**, **jump, jump-and-link** y **jump-register**. Los formatos de instrucción con sus códigos de operación, identificadores de registros y valores del campo función se deben ajustar a los definidos en el documento *MIPS Reference Data*.

El diseño de la ALU debe seguir un estilo estructural y jerárquico, en el que se diseñan bloques básicos que luego son instanciados para crear otros más complejos y de mayor nivel en la jerarquía de diseño. Para construir la ALU, de la biblioteca de componentes de Logisim Evolution sólo se utilizaron los siguientes componentes: compuertas AND, OR, NOT y multiplexores.

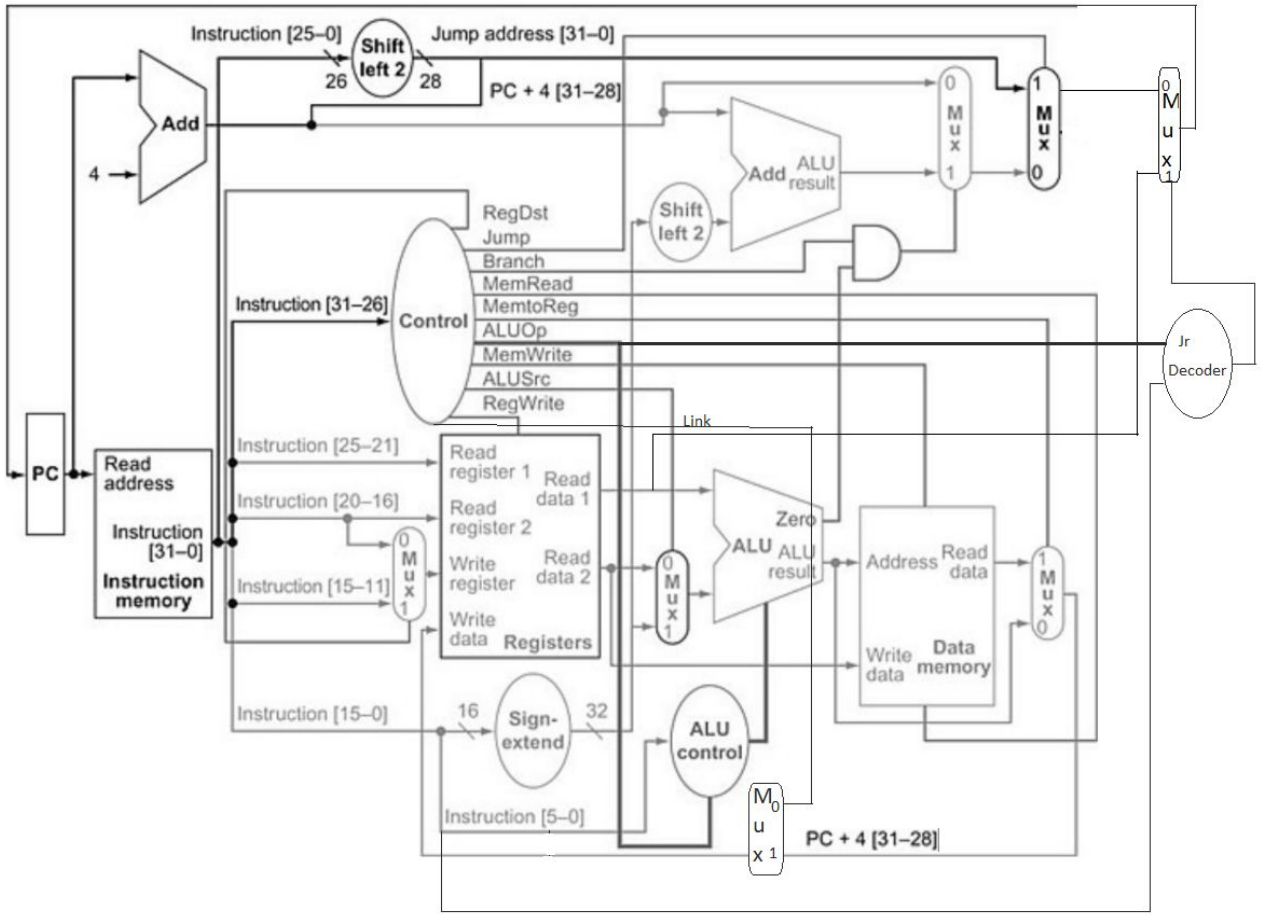
Finalmente, el procesador será empleado para ejecutar un programa, cargado en la memoria de instrucciones en código de máquina.

Objetivos

- Comprender los requerimientos de una versión reducida de la arquitectura MIPS de 32 bits para realizar una implementación en forma de procesador monociclo (ruta de datos y unidad de control).
- Codificar, ensamblar y simular un programa de prueba para verificar el comportamiento correcto del procesador.
- Emplear herramientas de software para el diseño y la simulación de computadores digitales.

Desarrollo

DIAGRAMA DE PROCESADOR MONOCICLO



Para la construcción de un procesador monociclo se necesitan varios componentes tales como se muestra en la imagen anterior, como cada uno tiene una función específica dentro del circuito, a continuación se hará la descripción de cada uno.

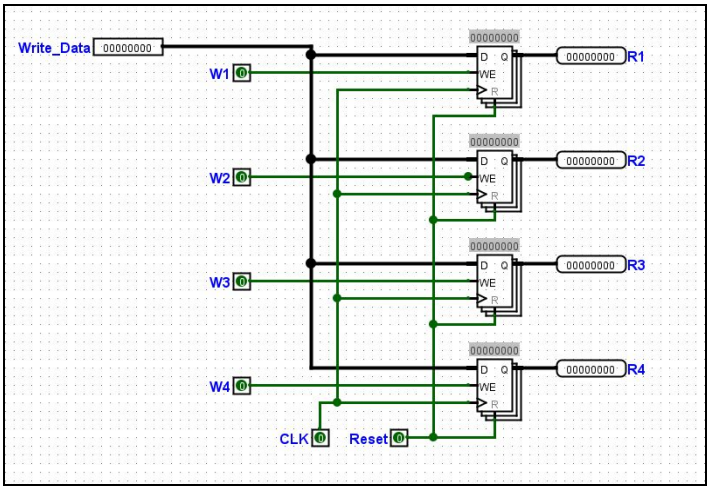
COMPONENTES de procesador monociclo

1. BANCO DE REGISTROS

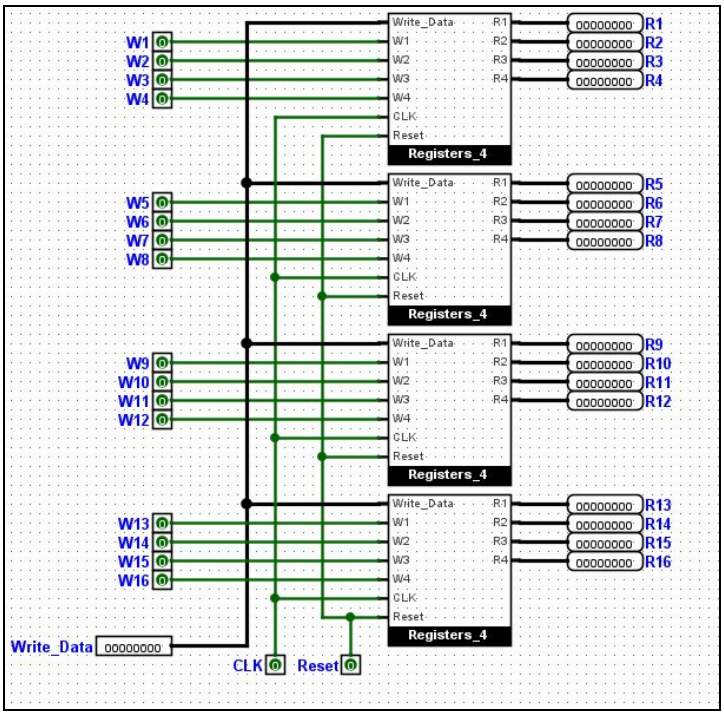
El banco de registro es el componente donde se almacenan los datos de acceso rápido, pero muy limitados, estos se llaman *registros*. Este banco cuenta con 32 registros como lo describe la arquitectura de MIPS.

Para este componente se utilizó el diseño jerárquico, por eso se inició con un banco de con 4 registros, luego este utilizó para uno de 16, y este de 16 registros se utilizó para construir el de 32 registros.

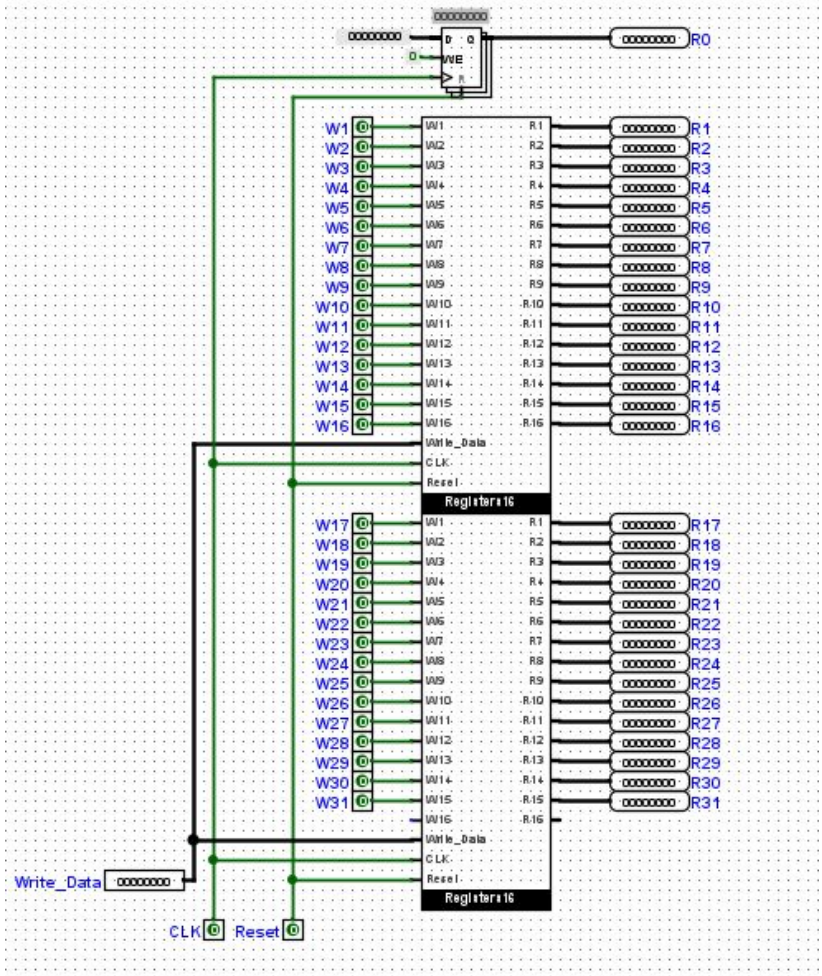
4Registros



16Registros

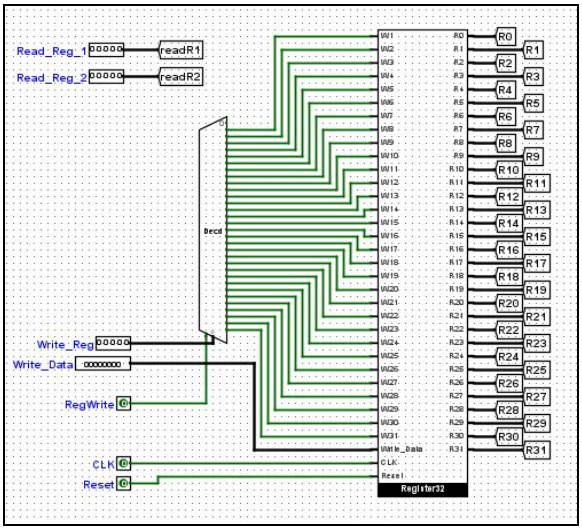


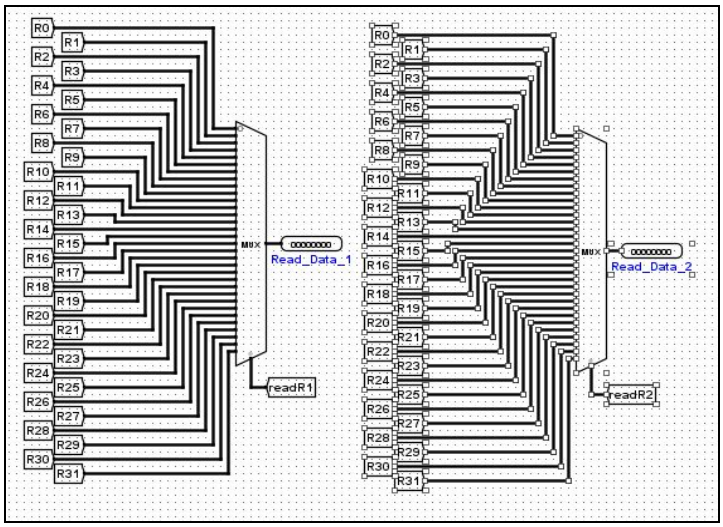
32Registros



Este componente en el registro **$zero** tiene como dato el número '0' como se requiere en la arquitectura

Para acceder de manera correcta a estos registros es necesario utilizar un decodificador y un multiplexor, donde estos recibirán el número de registro que se desea leer o copiar segun se le indique, a continuación está el banco de registros final.



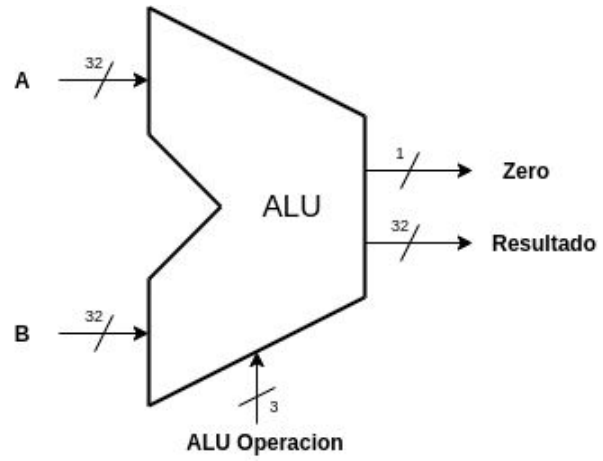


Donde:

- Read_reg_1, Read_reg_2: Son dos registros del banco de los cuales se quiere conocer su contenido.
- Read_Data_1, Read Data 2: Son las salidas.
- RegWrite: indica el registro sobre el cual se quiere escribir.
- Write_Data: contiene la información que se va a escribir sobre el registro indicado por la entrada regWrite.
- CLK: Señal de reloj necesaria debido a que la escritura en los registros se hace de manera **síncrona**, cabe anotar también que la lectura se hace de manera asíncrona.
- Regwrite indica si en el banco se va a hacer escritura de datos, por lo que actúa como condicionante de la señal de reloj, para que la escritura que se hace en un flanco de subida del reloj solo se realice si Regwrite está activa.
- Reset: limpia la totalidad del contenido de los registros del banco.

2. ALU

La ALU es de los componentes más esenciales del procesador, ya que se encarga de realizar operaciones aritmeticas y logicas segun la señal de control que le ingrese, la alu construida es la siguiente:

	- **A, B**: Operandos de entrada. - **ALU Operación:** Señal de control que indica la operación a realizar sobre los operando de entrada. Es una señal de tres bits que da la posibilidad de realizar hasta 8 operaciones en la ALU. - **Resultado:** Resultado de la operación realizada. - **Zero:** Salida que indica si el resultado arrojado es 0 (cero).

Las expresiones aritméticas que se van a soportar son 6, por lo cual, se utilizan 3 bits en la señal de control de ALUOperacion.

La siguiente es la codificación que nos tocó en el planteamiento del problema: (Codificación #2)

Función	ALU Operación
AND	010
OR	011
NOR	111
Suma	101
Resta	001
SLT	100

Para la construcción de la ALU de 32 bits se utilizó un diseño jerárquico en el que se inicia con una ALU_1bit luego, ALU_4bits, y de ahí la ALU_32bits.

La ALU se utilizan los siguientes componentes:

- FA_1bit_ov: Sumador completo con overflow
- ALU 1 bit: Alu de 1 bit
- ALU 4 bits: Alu de 4 bits
- ALU 1 bit _overFlow
- ALU 4bits_overFlow
- ALU 32bits
- **ALU (final)**

A continuación se hablará de cada uno:

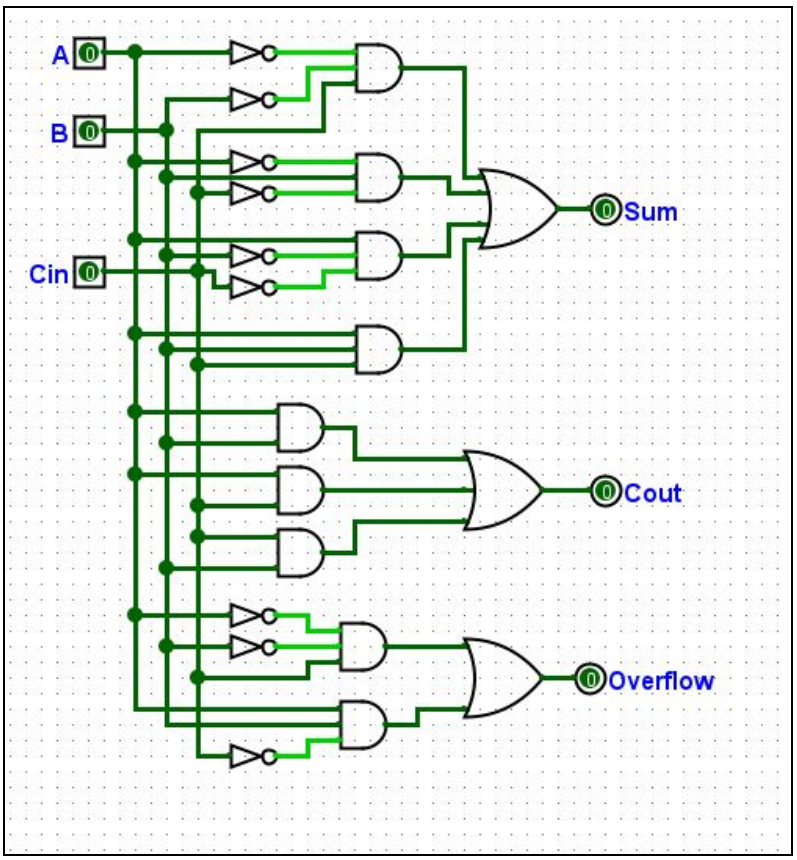
Sumador completo con overflow

El componente sumador completo tiene la capacidad de sumar bits, este está compuesto de las **entradas A, B** las cuales son los bits a sumar y **Cin** que es el bit de acarreo de entrada, en la **salida** tiene **R**(respuesta) y **Cout** que es el bit de acarreo de salida.

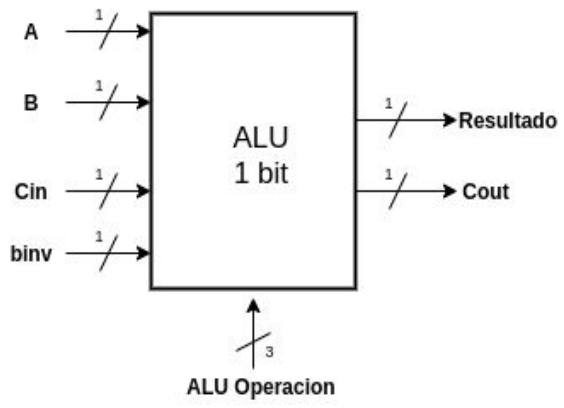
Se le añadió una nueva entrada y salida para saber en qué momento hay overflow en el circuito. Esta información es muy importante para el procesador porque algunas instrucciones se surten de ello para tomar decisiones.

Tabla de verdad del sumador:

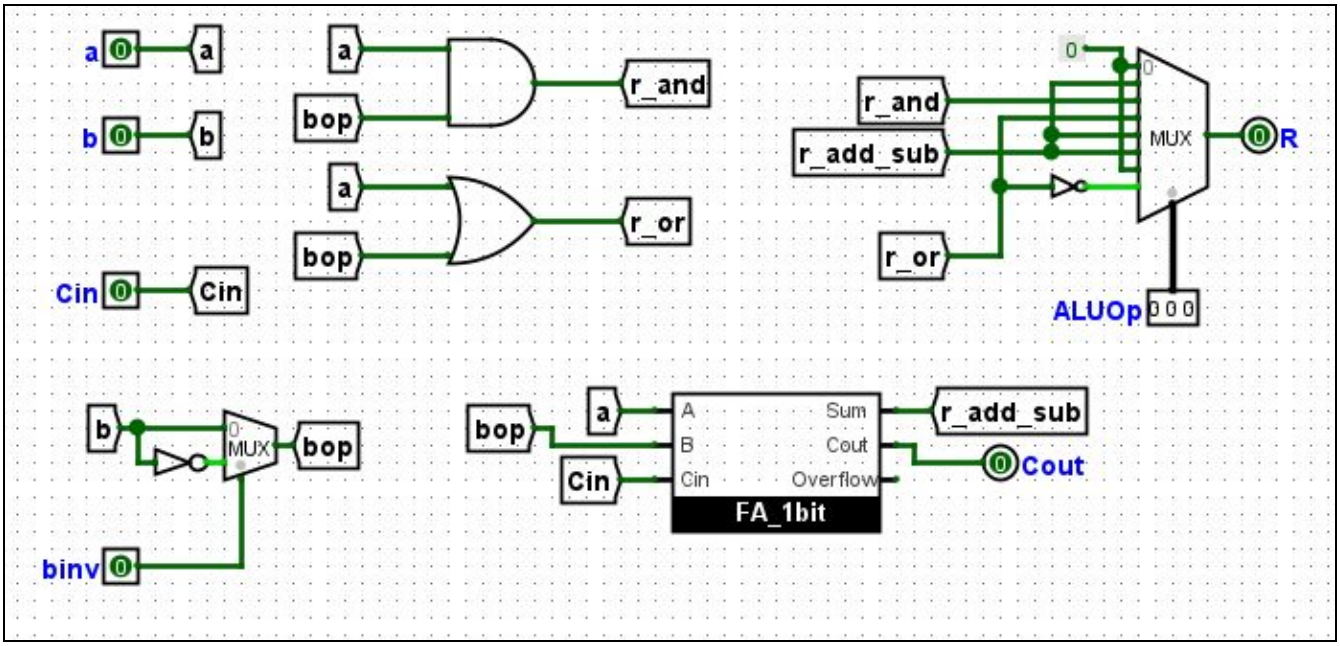
Entradas			Salidas		
A	B	Cin	Cout	Sum	overflow
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	0	0	1
1	1	1	1	1	0

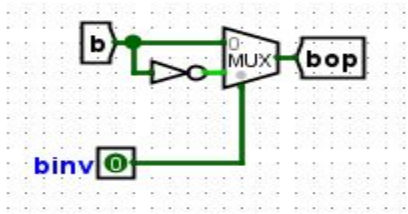


ALU de 1 bit

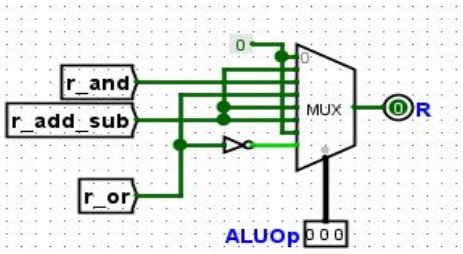
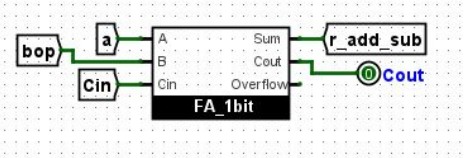


- **A, B:** Representan los bits sobre los cuales se va a realizar la operación.
- **Cin:** Bit de acarreo de entrada necesario en las operaciones suma y resta.
- **binv:** Bit que indica si se debe tomar el inverso de la entrada B.
- **ALU Operación:** Señal de control que indica la operación a realizar sobre los operando de entrada. Es una señal de tres bits que da la posibilidad de realizar hasta 8 operaciones en la ALU.
- **Resultado:** Resultado de la operación Realizada.
- **Cout:** Bit de acarreo de salida producido por las operaciones de suma y resta.





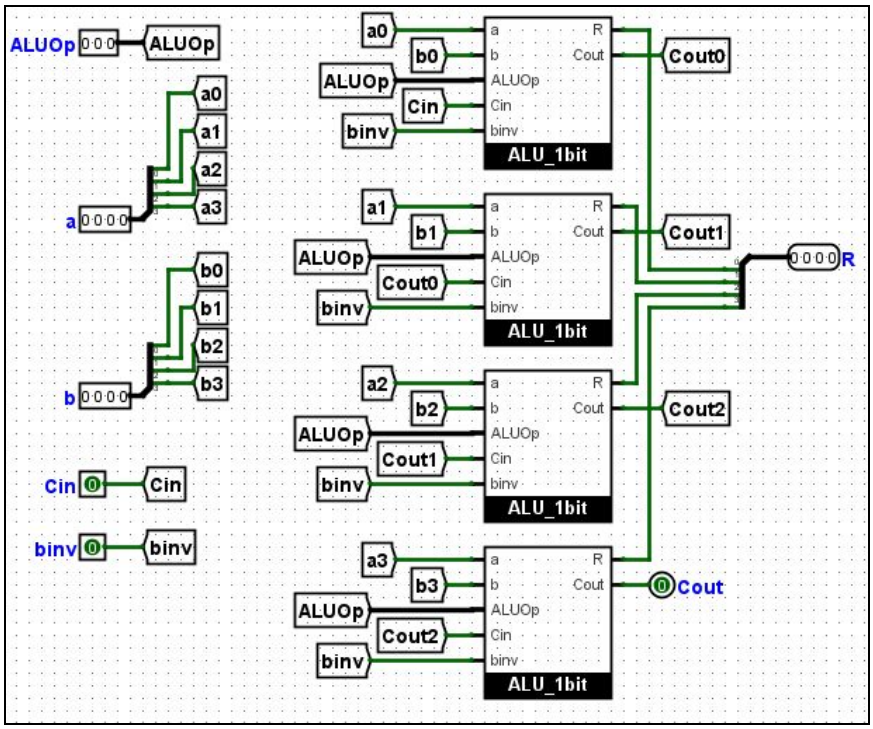
Cuando la operación es una resta **binv** toma el valor de 1, y pone negativo la entrada respectiva

	Multiplexor que habilita la entrada segun la señal que nos arroje el ALUOp
	Nos entrega el resultado de la operación requerida.

Resta:

Para la realización de la operación resta se decide hacer uso del sumador de 1 bit realizado anteriormente y transformando la expresión (A - B) a su expresión equivalente (A+ (B’ + 1)) en B’ es la negación de B.

ALU De 4 bits

	- **ALUOp:** Operación que se realizará - a, b: Entradas a las que se le aplicará la operación. - Cin: bit de acarreo. - binv: Entrada para saber cuando es una suma o resta.

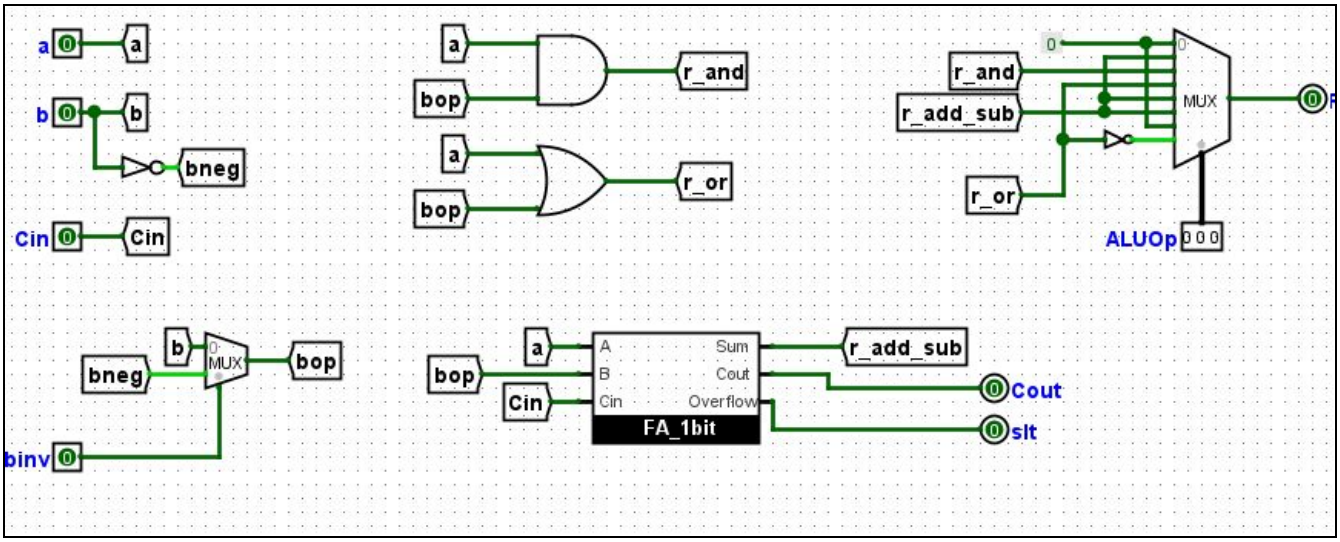
SLT:

Esta operación se traduce como Active si A es menor que B, por lo que es necesario saber si A es menor que B antes de tomar una decisión. Para saber esto basta con hacer A - B, si el resultado es negativo quiere decir que A efectivamente es menor que B. Pero solo se sabrá esto analizando el bit de signo que será el más significativo o su acarreo de salida en caso de haber overflow.

Debido a lo anterior, en la ALU de 1 bit sólo será necesario calcular la resta entre A y B. Esta ALU servirá para los 31 bits menos significativos de la ALU de 32 bits, en el caso del bit más significativo se necesitará construir una ALU de 1 bit en donde su sumador calculará además si existe overflow o no, este componente se diseña a continuación.

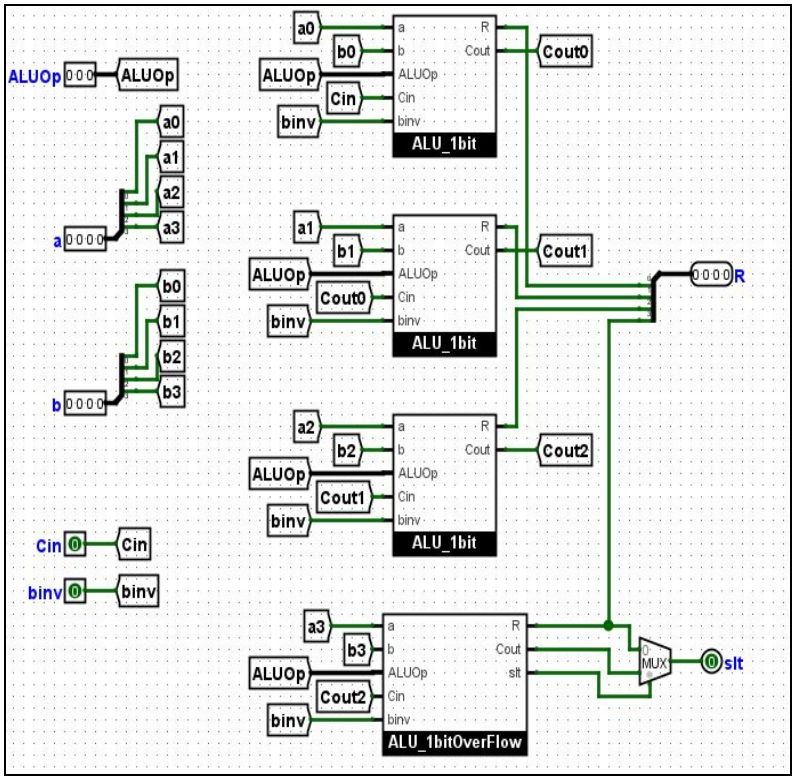
ALU de 1 bit con OverFlow

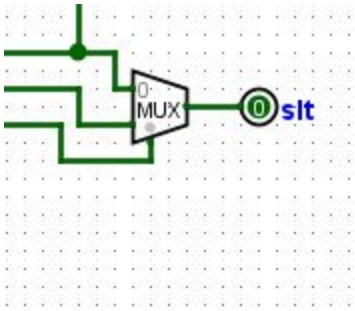
Es muy similar a la ALU de 1 bit sencilla, pero en este caso se le agrega una salida para saber cuando hay overflow segun lo que arroje el sumador.



ALU de 4 bit con OverFlow

Es muy similar a la ALU de 1 bit sencilla, pero en este caso se le agrega una salida para saber cuando hay overflow segun lo que arroje el sumador.



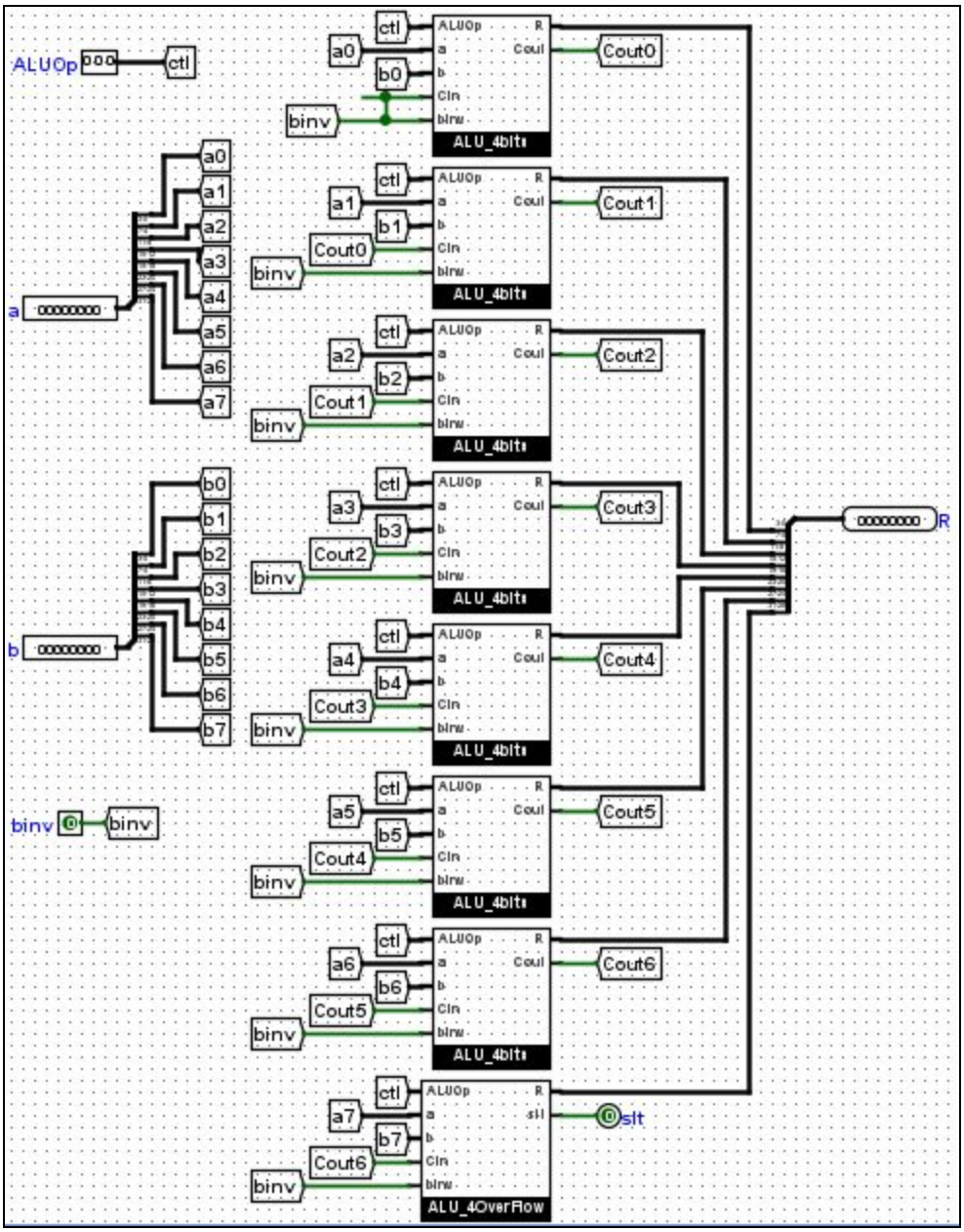


El resultado de la operación **SLT** solo es posible conocerlo con la salida de la ALU de 1 bit con overflow, en donde la salida overflow actúa de selector de un multiplexor para conocer el signo de la resta realizada (A - B), el cual la salida es resultado si no hay overflow y la salida Cout en caso de haber overflow. El signo elegido será el resultado de la operación SLT con 31 ceros en la parte más significativa.

ALU de 32 Bits

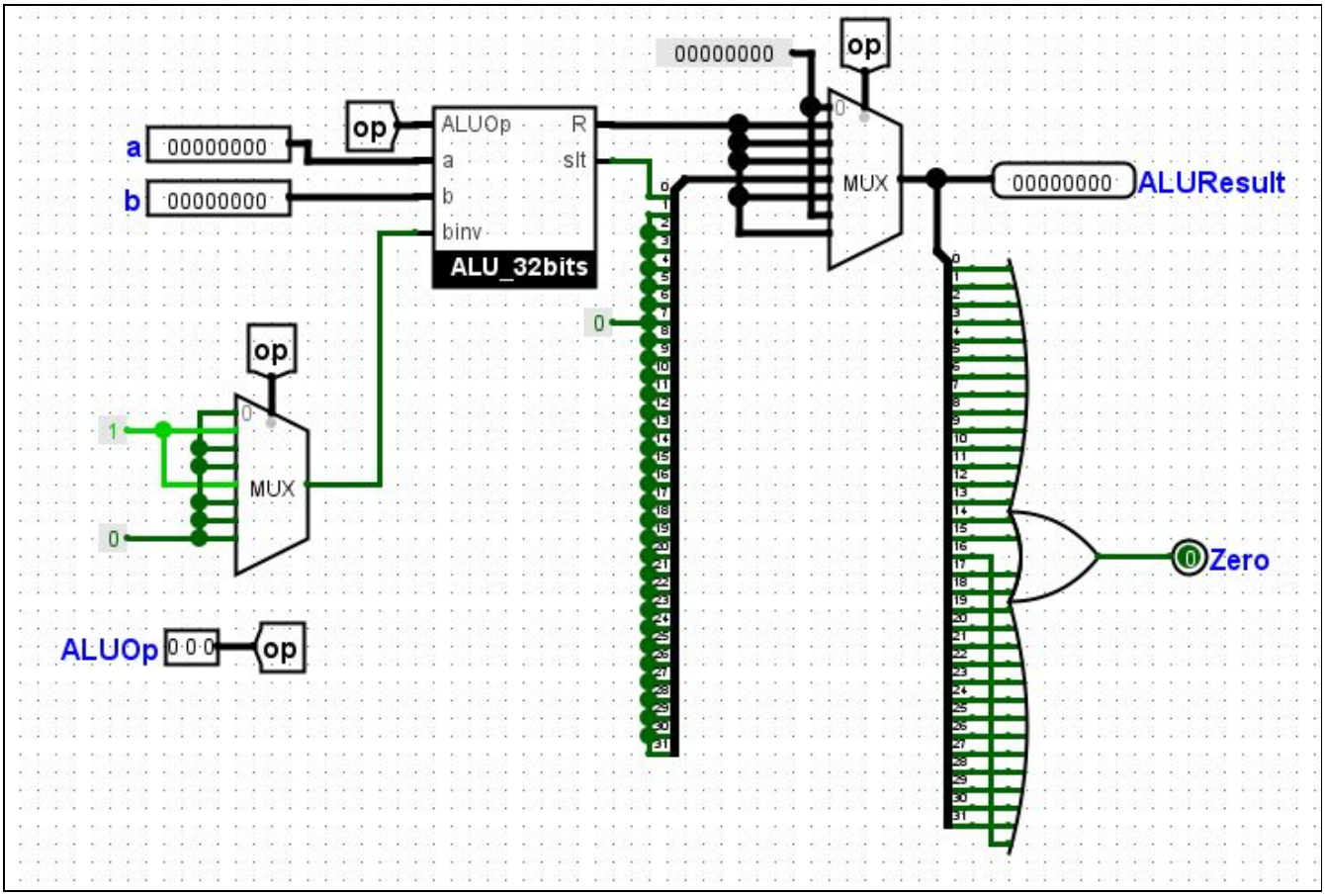
Esta es una conexión de varias ALU de 4 bits donde se arroja el resultado que se obtiene para mostrar.

Por manejar un orden se dispuso de esta ALU_de_32_bits en otro componente (ALU) para completar las conexiones requeridas segun las **señales de control**.



ALU (FINAL)

Este es el componente es la conexión de la ALU de 32 bits con las señales controladoras, segun el código de operación y las entradas, este nos entrega en la salida **ALUResult**, el resultado de la operación que se le solicitó a la ALU.



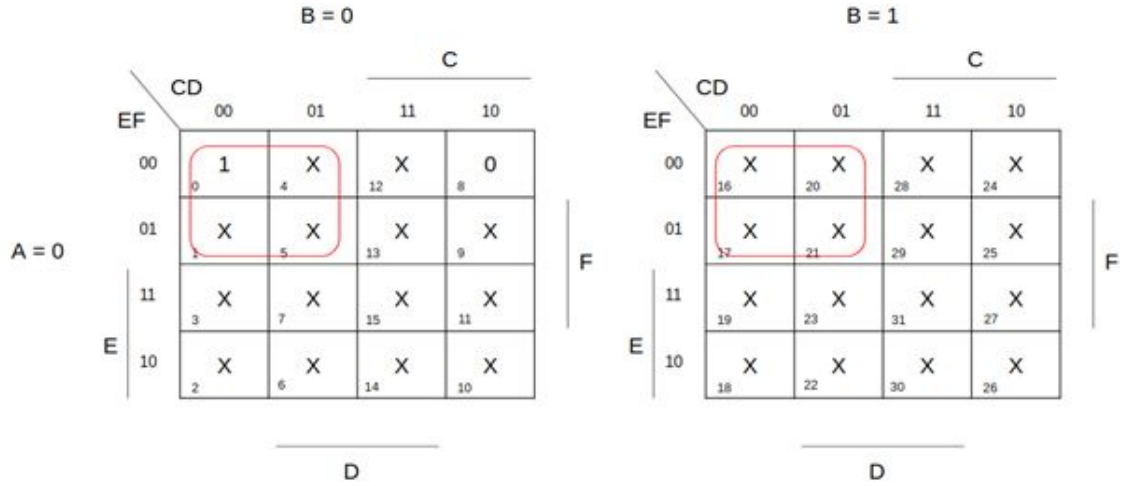
La salida **Zero** consiste en pasar cada uno de los bits del resultado por una compuerta NOR de tal manera que la salida se active sólo cuando todos los bits son cero.

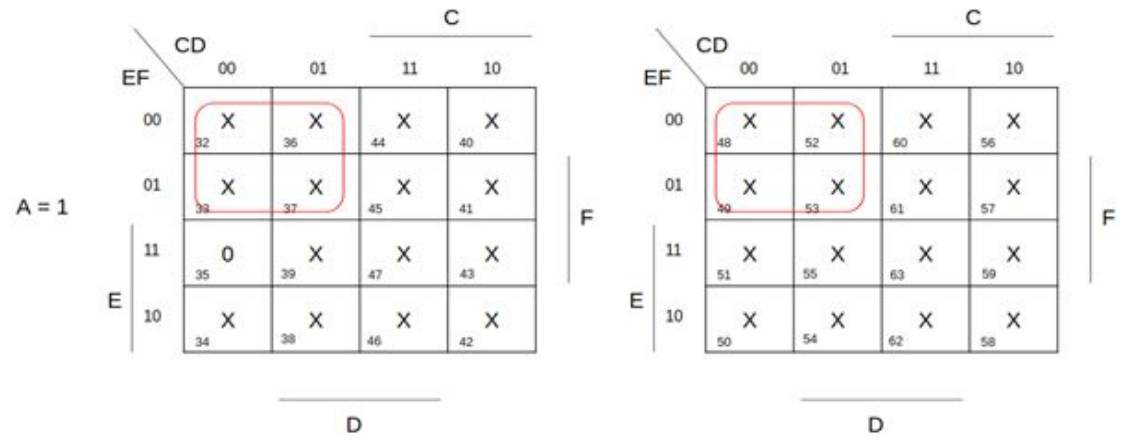
3. CONTROL

Este componente es de gran importancia en el diseño del procesador porque es el encargado de manejar cada una de las señales de control segun la instrucción que se esté realizando. Para esto se realiza una tabla de verdad donde en las **entradas** estará el **opcode** de la instrucción que se ejecute en este momento, y en las **salidas** cada uno de las **señales de control.**

Como el procesador que se está realizando recibe las instrucciones (load word (lw), store word (sw), add, sub, and, or, nor, set-on-less-than (SLT), branch if equal, jump, jump-and-link y jump-register) se utiliza la siguiente clasificación:

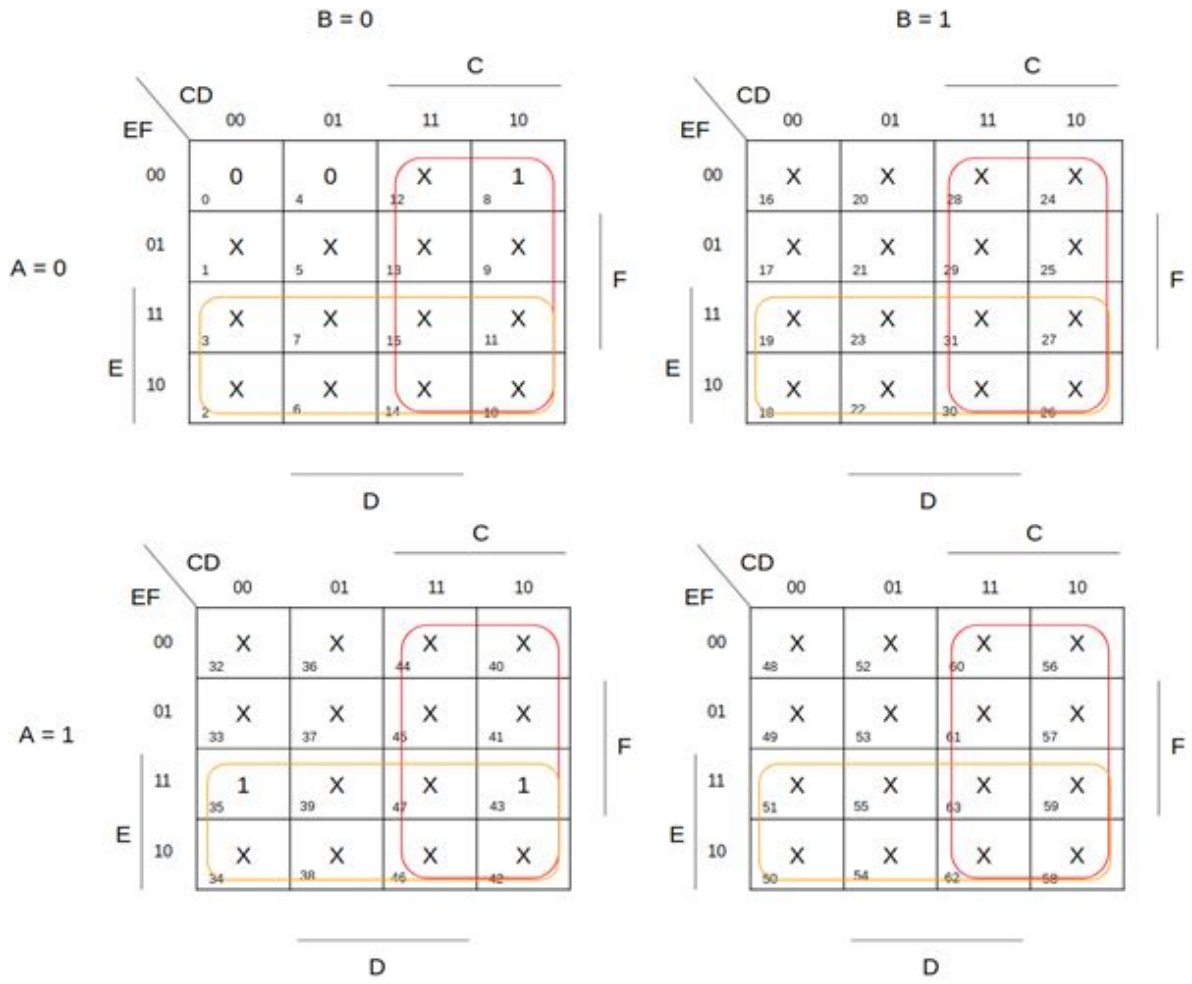
	Inst.	Cod. de Operación (OP)						Salidas										
		5 A	4 B	3 C	2 D	1 E	0 F	Reg Dst	ALU Src	Memto Reg	RegWr ite	MemR ead	Mem Write	Bran ch	ALU Op1	ALU Op2	Ju mp	Li nk
0	Tipo-R	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0
2	Jump	0	0	0	0	1	0	X	X	X	0	0	0	0	X	X	1	0
3	Jal	0	0	0	0	1	1	X	X	X	1	0	0	0	X	X	1	1
4	Beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1	0	0
35	LW	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0
43	SW	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0	0	0

Tabla de Karnaug	
Salida	Tabla
RegDst	

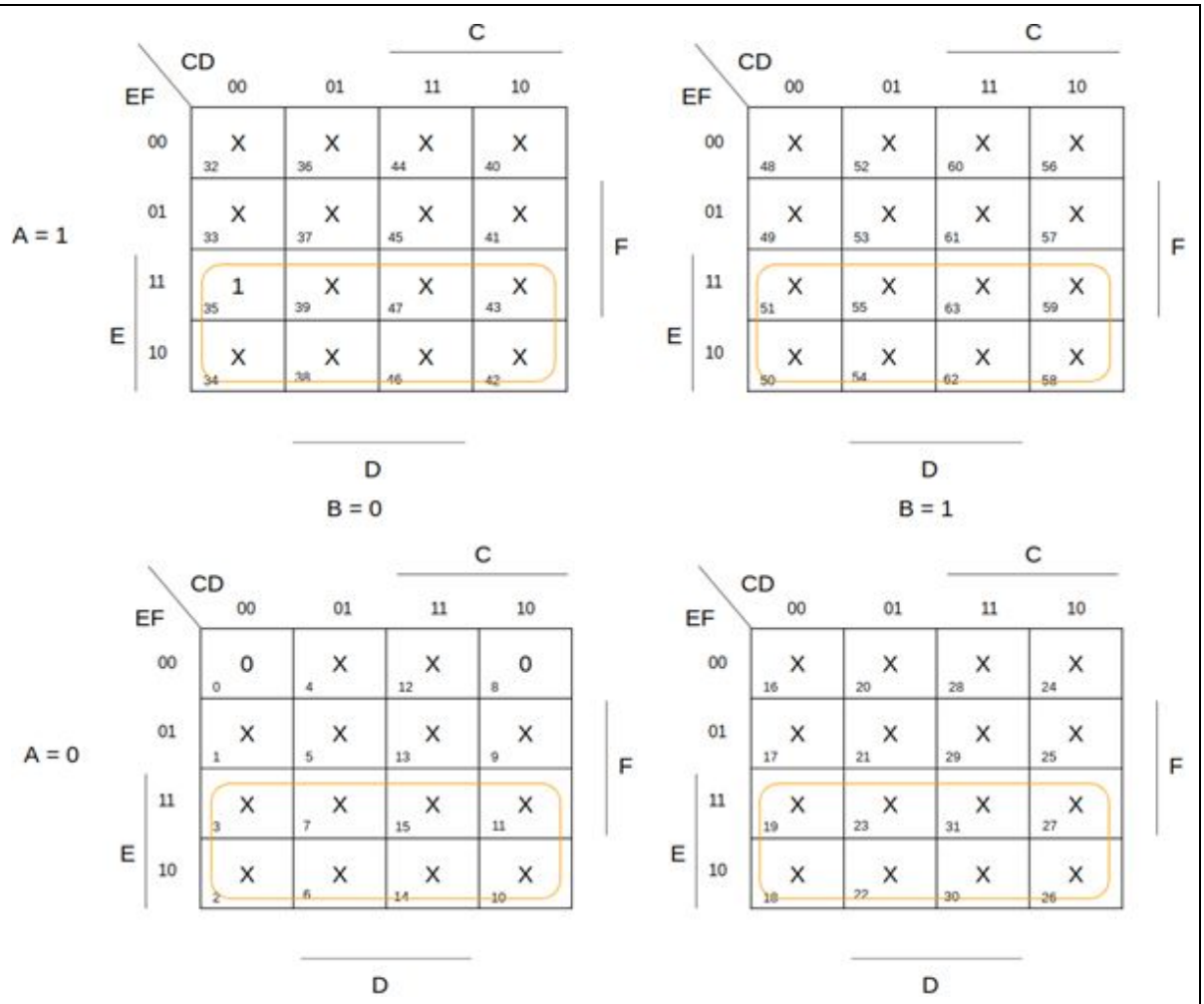
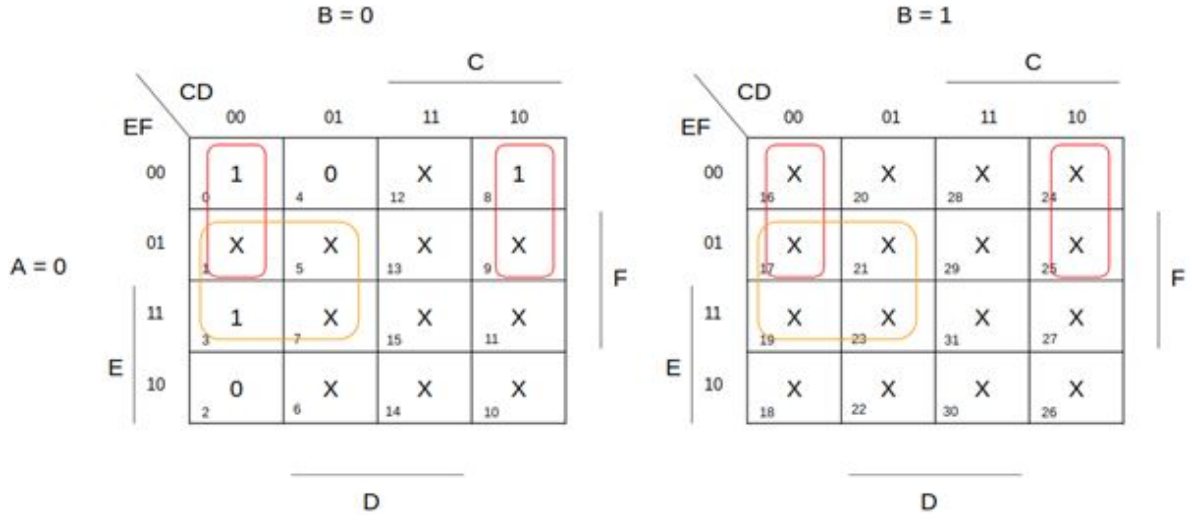


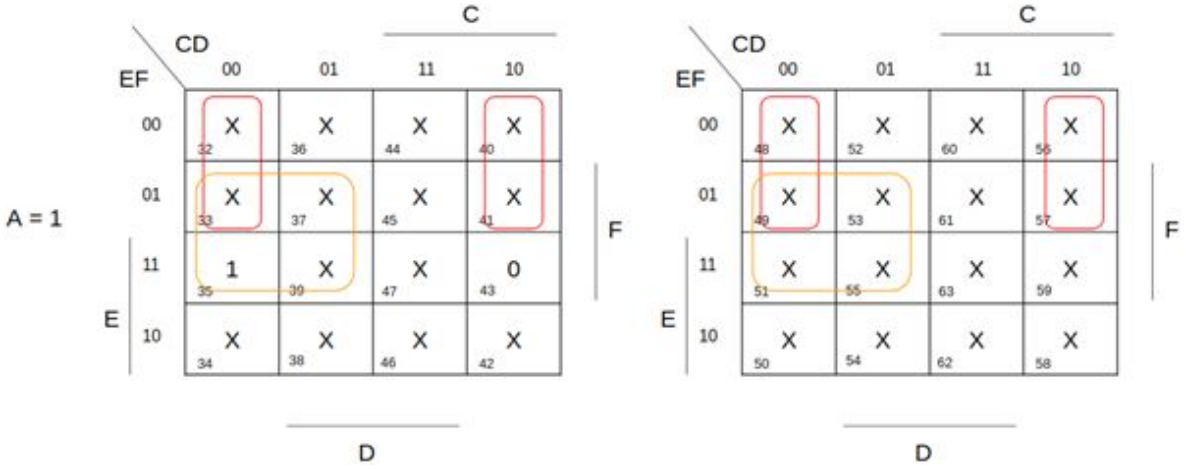
Expresión: E’+C’

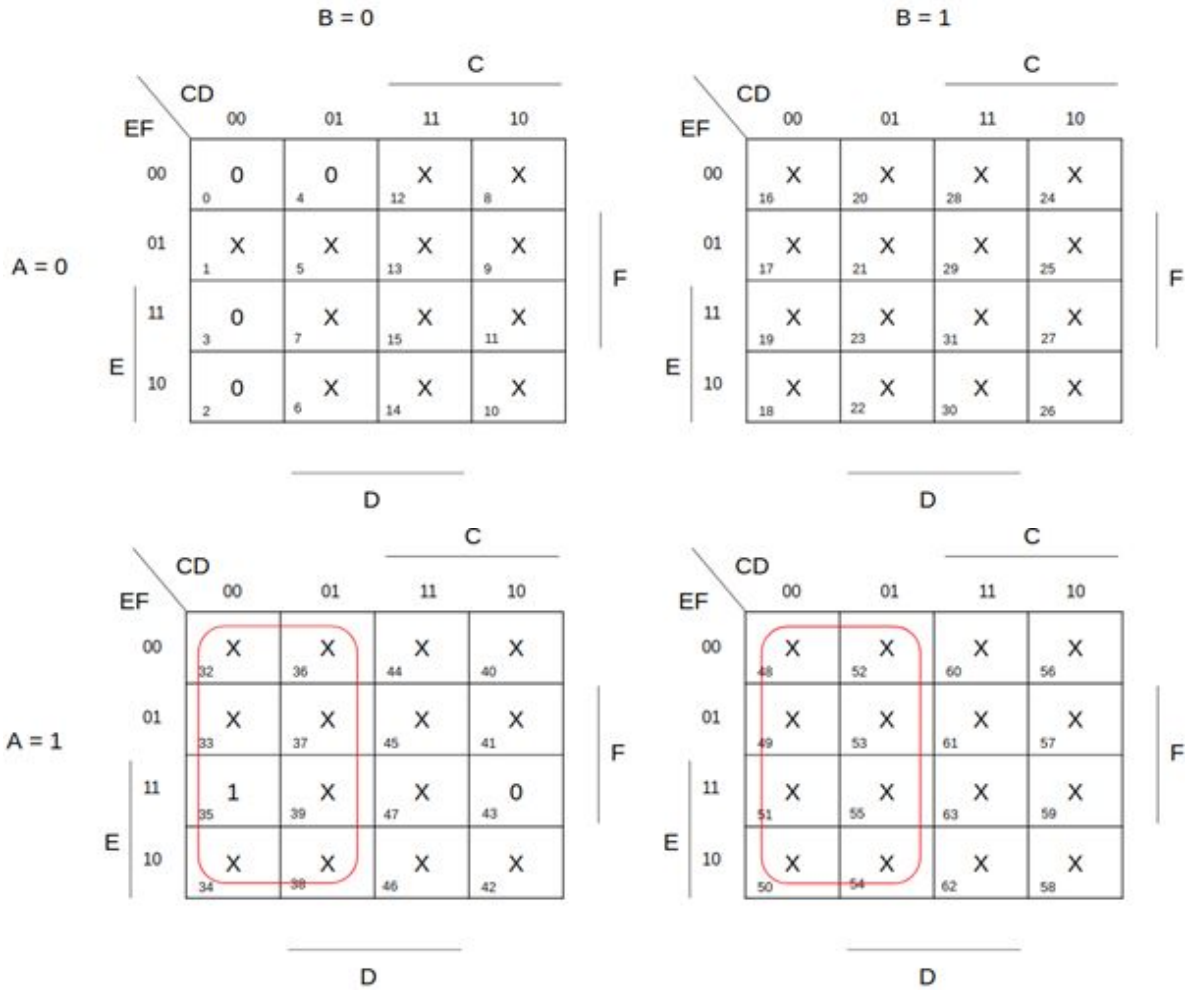
ALUSrc



Expresión: E+C

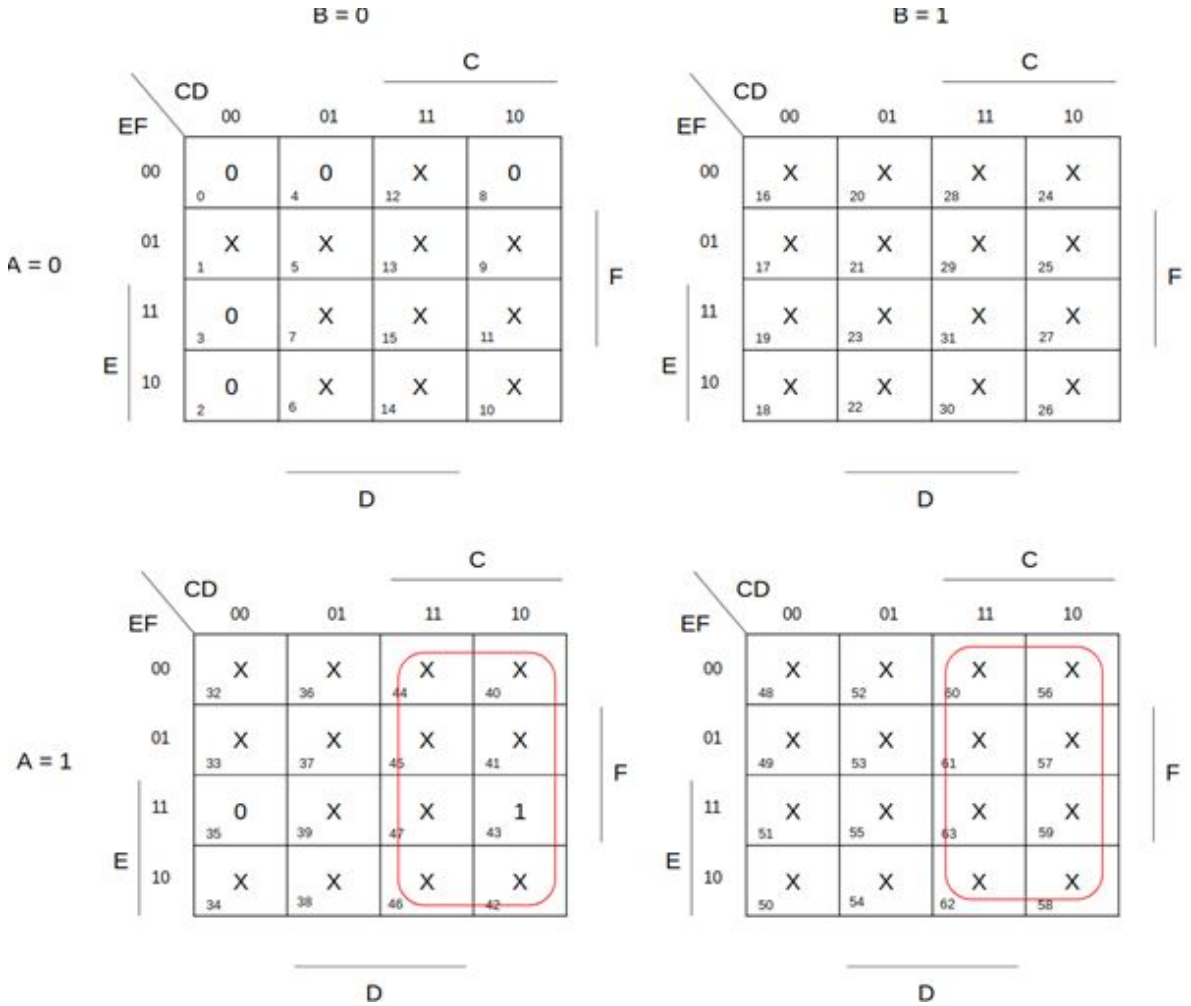
Memtoreg	 Expresión: E
RegWrite	

<table>
<tr>
<td></td>
<td>


Expresión: DE’ + C´F
</td>
</tr>
<tr>
<td>MemRead</td>
<td>


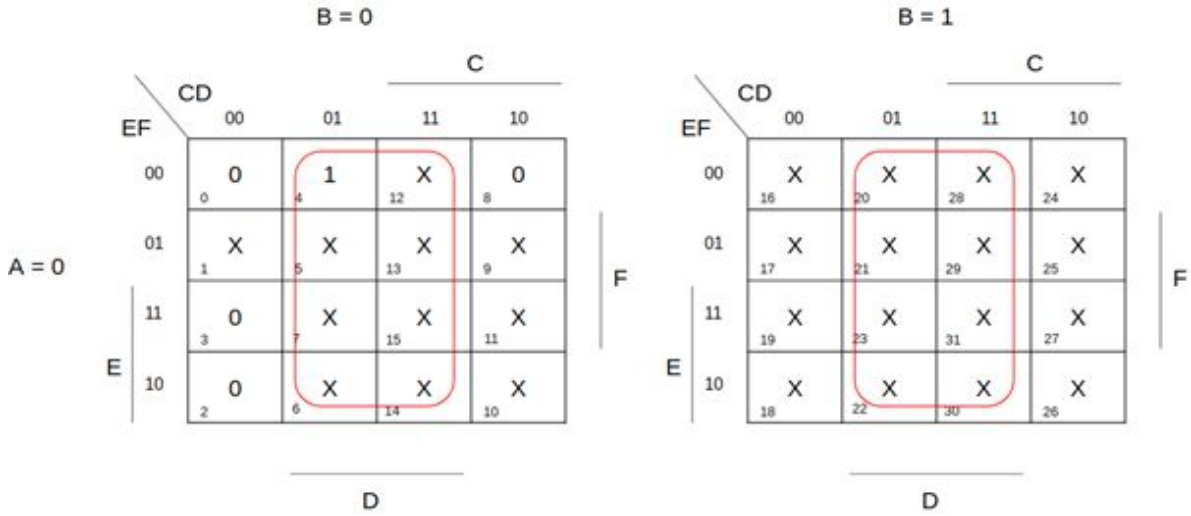
Expresión: A C’
</td>
</tr>
</table>

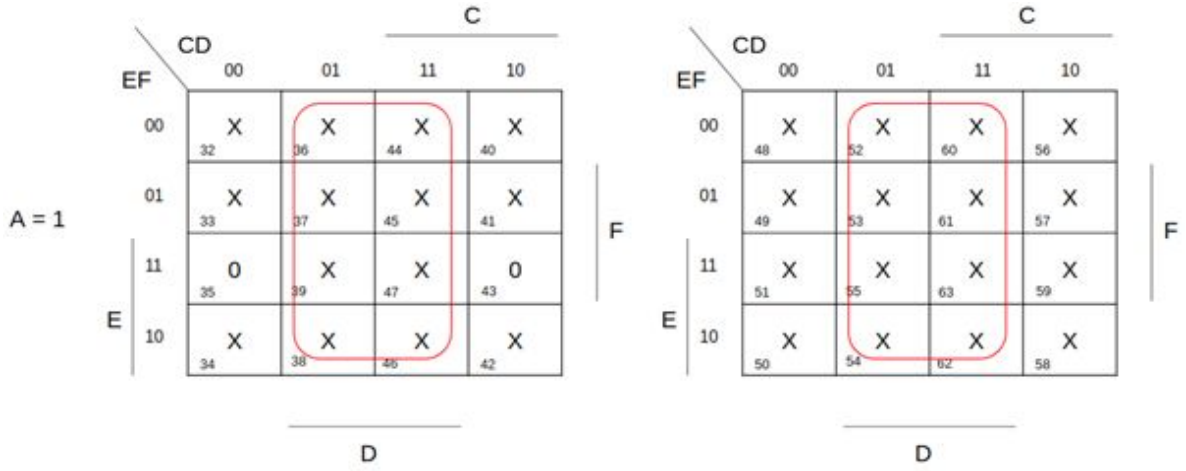
Mem Write



Expresión: A C

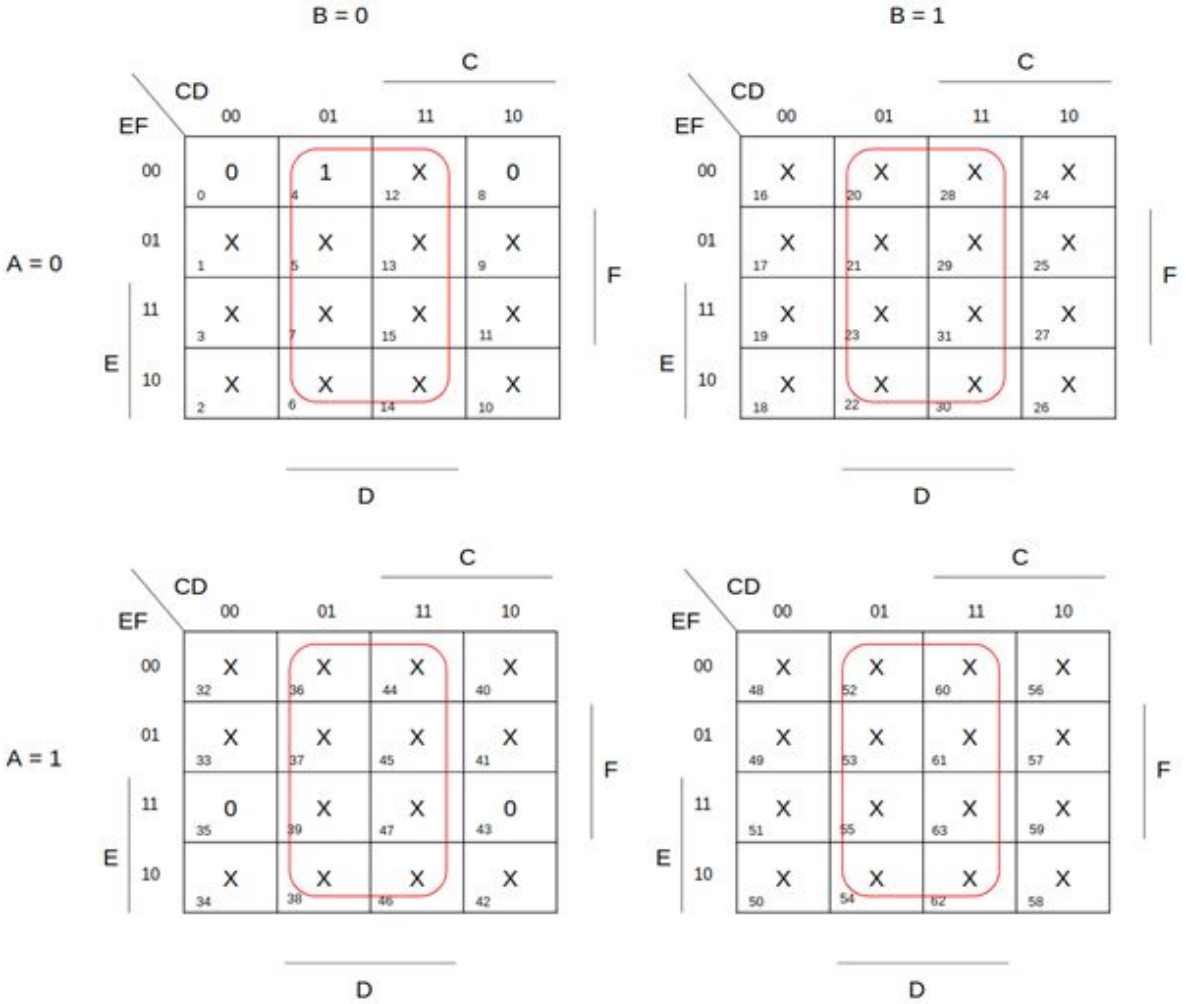
Branch





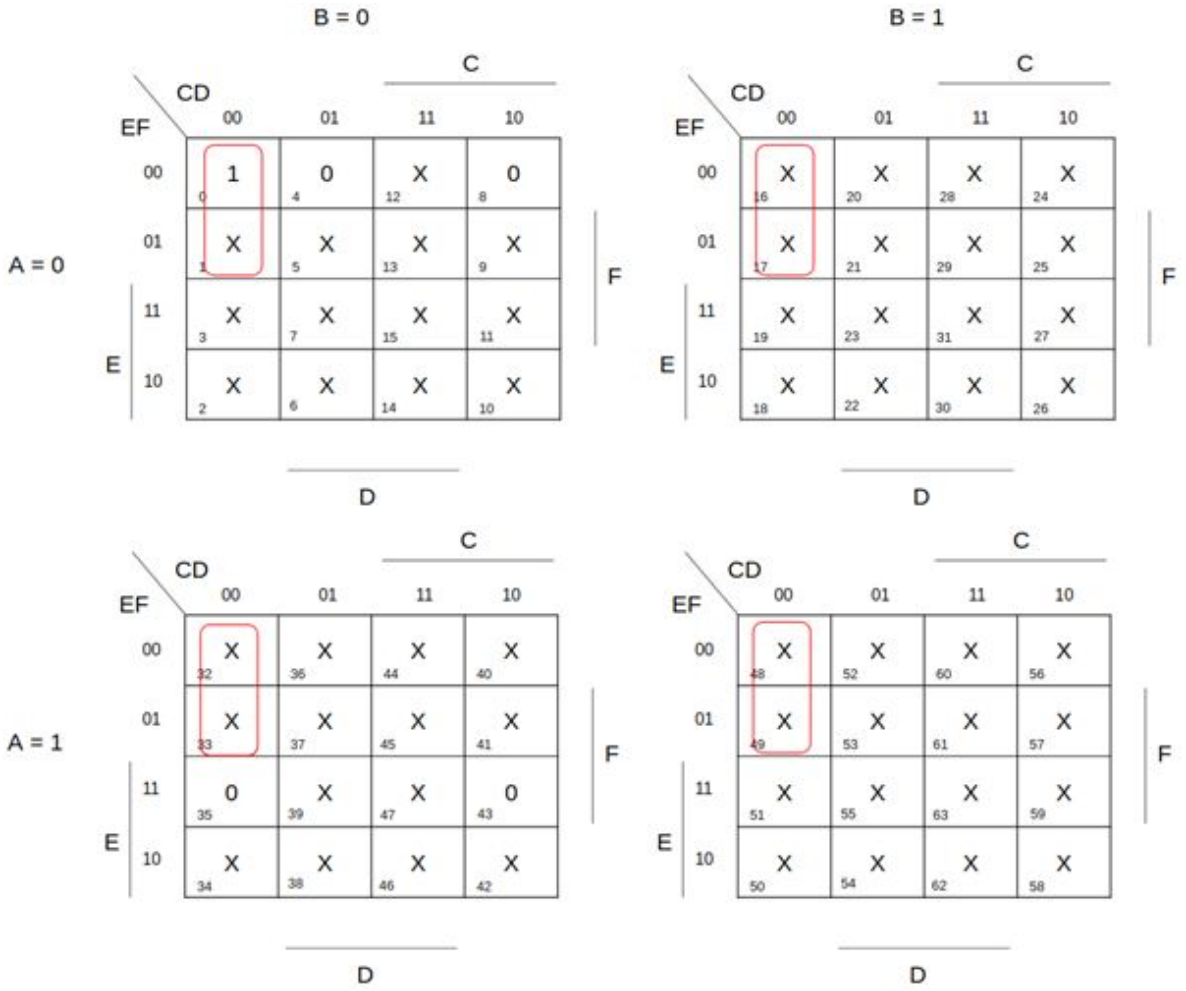
Expresión: D

Op0



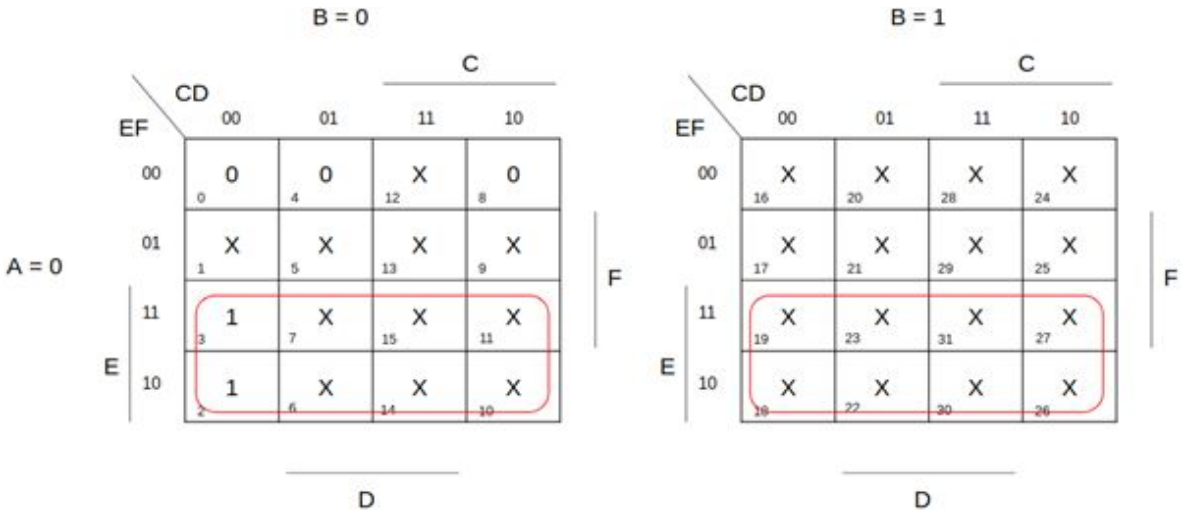
Expresión: D

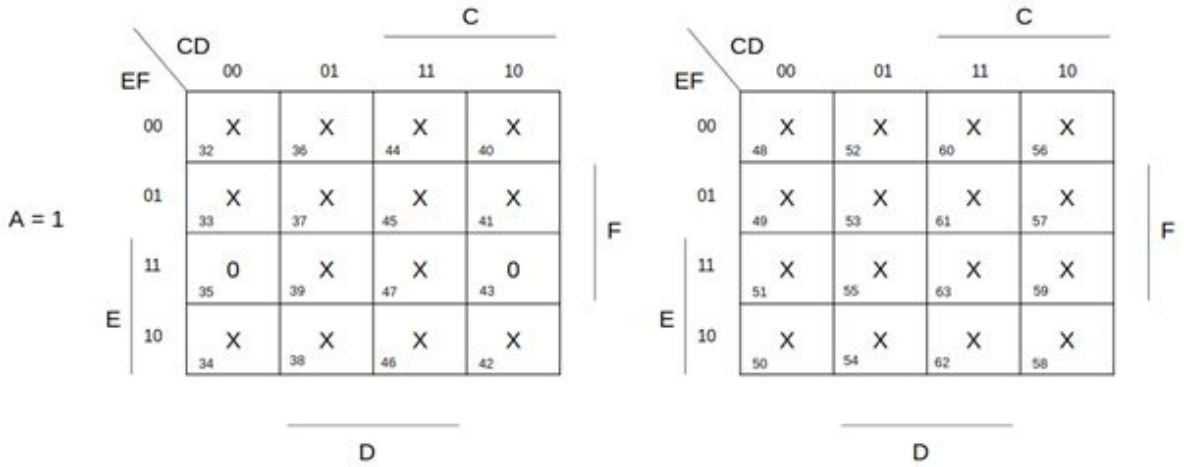
Op1



Expresión: C’ D’ E’

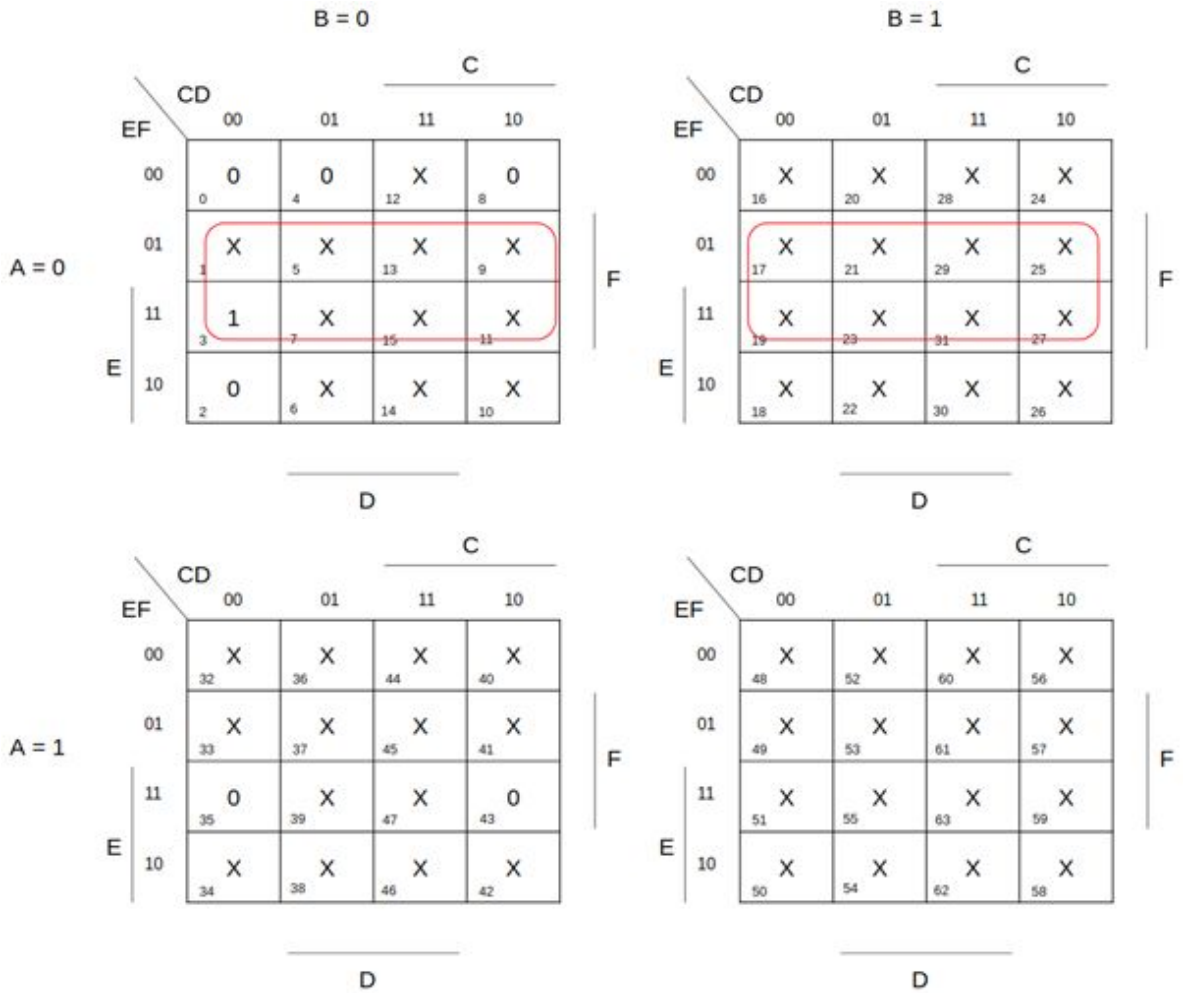
Jump





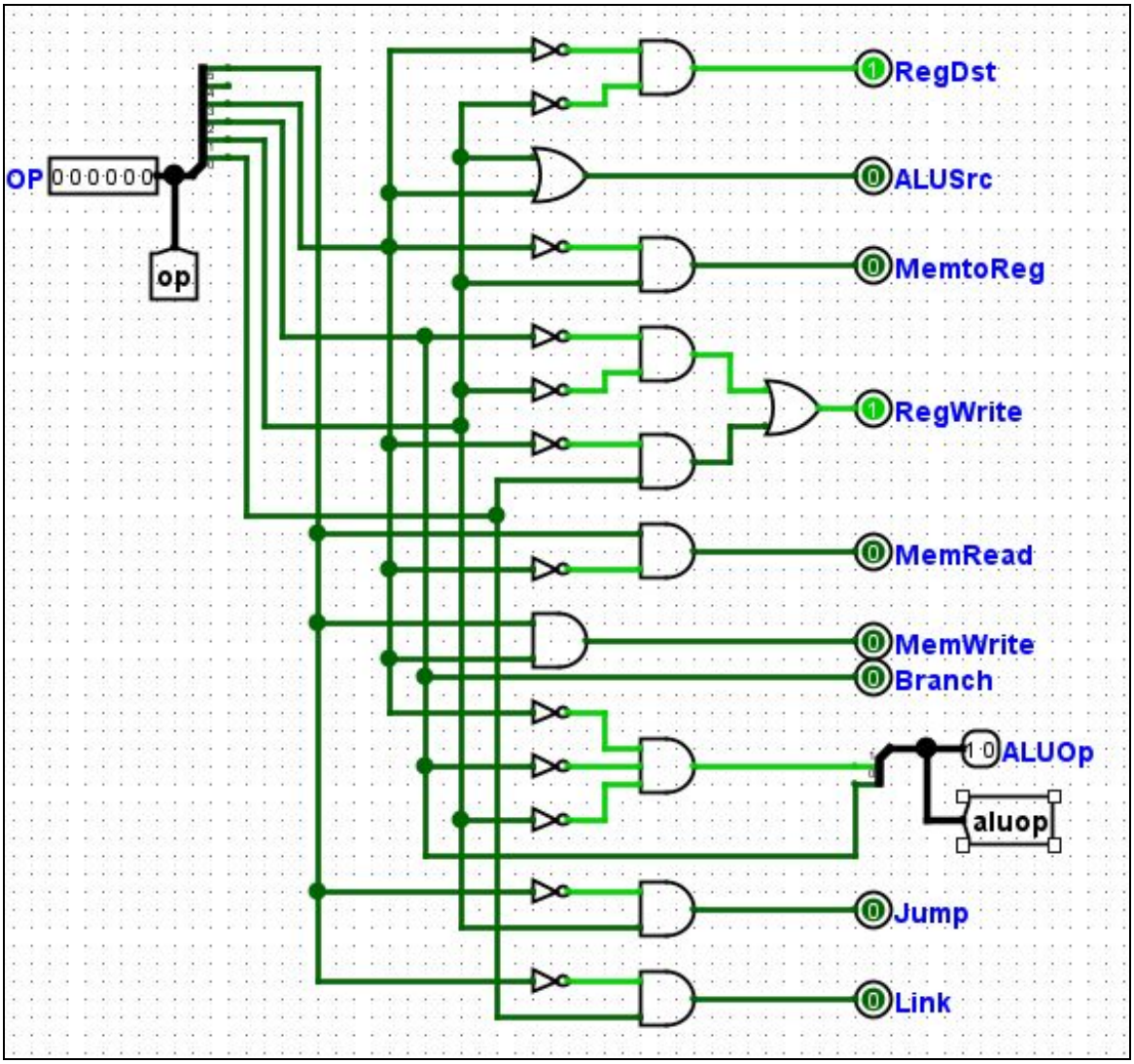
Expresión: A’ E

Link



Expresión: A’ F

De donde se obtuvo el siguiente circuito:



4. ALUControl

Este componente decodifica la señal de la ALUOp generada por el componente de control, de tal manera que genera un código de 3 bits sobre la operación que debe realizar la ALU.

La señal ALUOp es ‘00’ para indicar que se debe sumar, y es ‘01’, para indicar que se debe realizar una resta. Finalmente la señal de tipo R se definió como ‘10’, indicando que la operación a realizar la define el campo de función de 6 bits.

Alu Control									
	Función						Operación		
	5 a	4 b	3 c	2 d	1 e	0 f	2 A	1 B	0 C
add	1	0	0	0	0	0	1	0	1
sub	1	0	0	0	1	0	0	0	1
AND	1	0	0	1	0	0	0	1	0
OR	1	0	0	1	0	1	0	1	1
NOR	1	0	0	1	1	1	1	1	1
SLT	1	0	1	0	1	0	1	0	0

Los resultados de esta tabla son de acuerdo a la codificación que nos tocó en el planteamiento para cada una de las operaciones..

Tabla de Karnaug	
Salida	Tabla
Op2	

Map

	$\overline{D}.\overline{E}.\overline{F}$	$\overline{D}.\overline{E}.F$	$\overline{D}.E.F$	$\overline{D}.E.\overline{F}$	$D.\overline{E}.\overline{F}$	$D.\overline{E}.F$	$D.E.F$	$D.E.\overline{F}$
$\overline{A}.\overline{B}.\overline{C}$	x	x	x	x	x	x	x	x
$\overline{A}.\overline{B}.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.\overline{C}$	x	x	x	x	x	x	x	x
$A.B.C$	1	x	x	0	0	0	1	x
$A.\overline{B}.C$	x	x	x	1	x	x	x	x
$A.B.C$	x	x	x	x	x	x	x	x
$A.B.\overline{C}$	x	x	x	x	x	x	x	x

$$y = C + D'E' + EF$$

Op1

Map

	$\overline{D}.\overline{E}.\overline{F}$	$\overline{D}.\overline{E}.F$	$\overline{D}.E.F$	$\overline{D}.E.\overline{F}$	$D.\overline{E}.\overline{F}$	$D.\overline{E}.F$	$D.E.F$	$D.E.\overline{F}$
$\overline{A}.\overline{B}.\overline{C}$	x	x	x	x	x	x	x	x
$\overline{A}.\overline{B}.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.\overline{C}$	x	x	x	x	x	x	x	x
$A.B.C$	0	x	x	0	1	1	1	x
$A.\overline{B}.C$	x	x	x	0	x	x	x	x
$A.B.C$	x	x	x	x	x	x	x	x
$A.B.\overline{C}$	x	x	x	x	x	x	x	x

$y = D$

Op0

Map

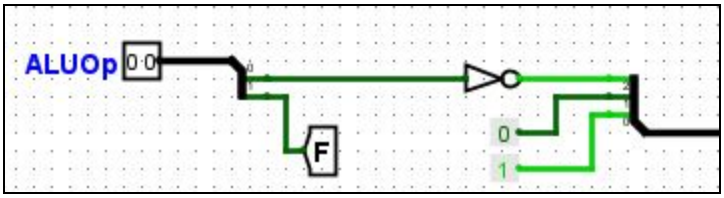
	$\overline{D}.\overline{E}.\overline{F}$	$\overline{D}.\overline{E}.F$	$\overline{D}.E.F$	$\overline{D}.E.\overline{F}$	$D.\overline{E}.\overline{F}$	$D.\overline{E}.F$	$D.E.F$	$D.E.\overline{F}$
$\overline{A}.\overline{B}.\overline{C}$	x	x	x	x	x	x	x	x
$\overline{A}.\overline{B}.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.C$	x	x	x	x	x	x	x	x
$\overline{A}.B.\overline{C}$	x	x	x	x	x	x	x	x
$A.B.C$	1	x	x	0	0	1	1	x
$A.\overline{B}.C$	x	x	x	0	x	x	x	x
$A.B.C$	x	x	x	x	x	x	x	x
$A.B.\overline{C}$	x	x	x	x	x	x	x	x

$y = F + D'E'$

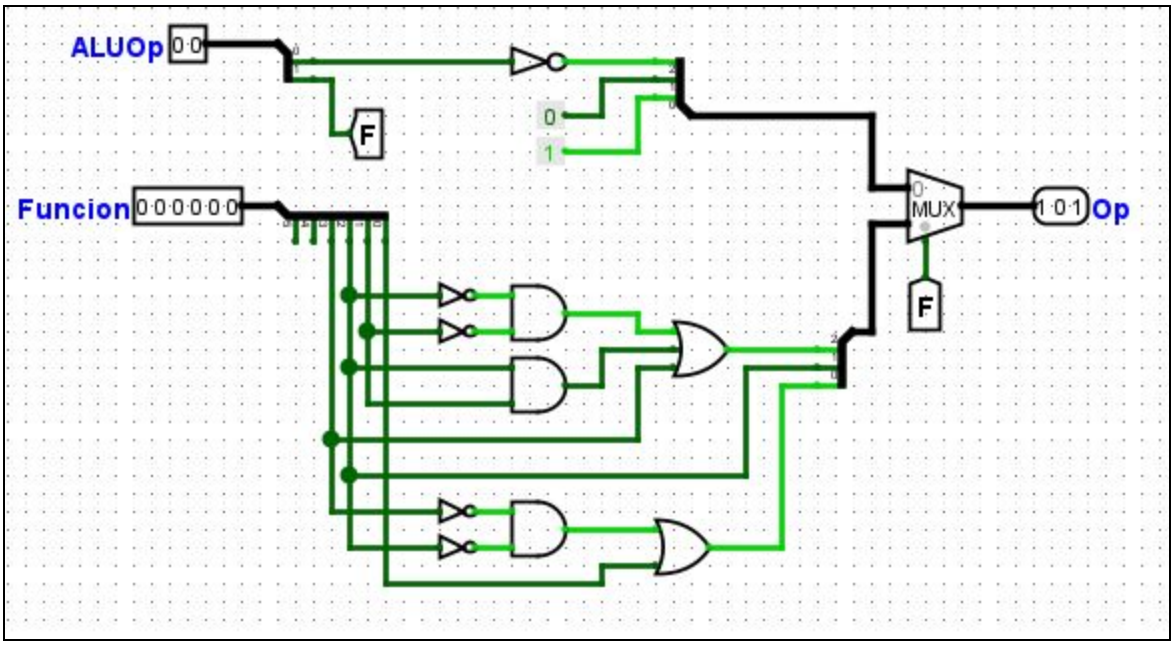
Finalmente se requiere una salida para indicar que explícitamente se está realizando una operación de tipo R. Para esto la tabla de verdad es la siguiente:

	ALUOp		Operación			Función
	1	2	2	1	0	
add	0	0	1	0	1	0
sub	0	1	0	0	1	0
tipo-R	1	0	X	X	X	1
	1	1	X	X	X	X

Que genera el circuito:



Finalmente el circuito de los resultados de las 2 tablas de verdad es:

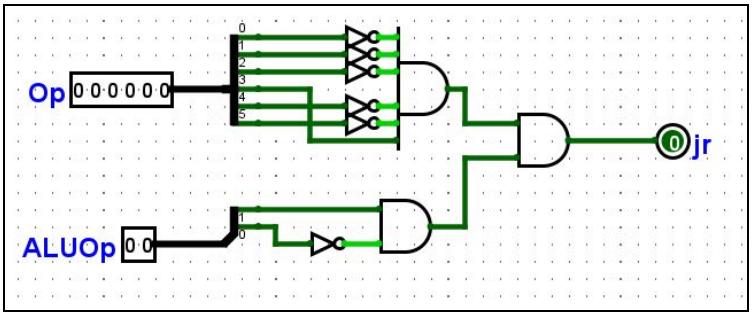


5. JR

JR: Para esta instrucción no es posible adicionar señales de control en el componente CONTROL debido a que esta instrucción tiene el **opcode** 000000, por lo que esta combinación ya está ocupada y es necesario analizar el campo función de la instrucción que son los bits 5-0.

Cuando la señal ALU op de dos bits es '10' indica que está procesando una instrucción tipo R y se proceda a analizar los 6 bits del campo función, donde la combinación debe ser '001000' para que la señal jr se active.

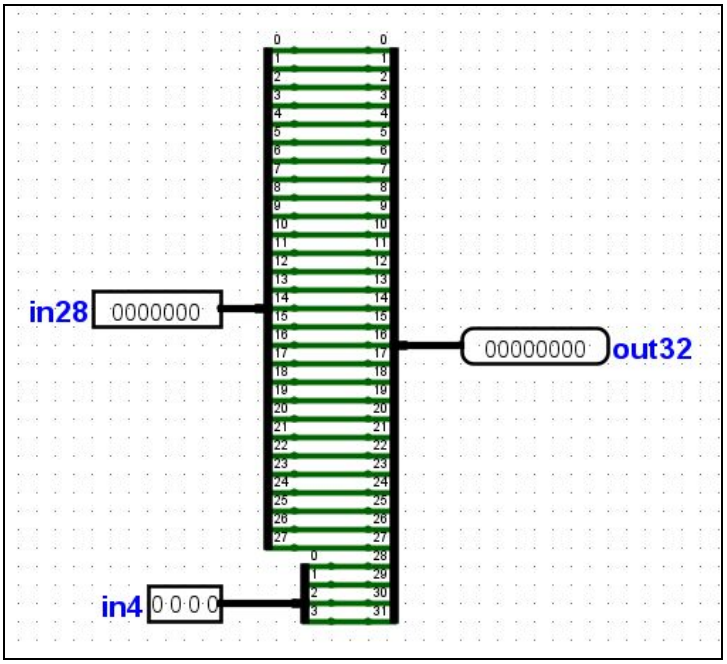
El circuito queda así



6. JOIN (J)

Este componente es necesario para la implementación de la instrucción *Jump* y *Jump and link,* donde según la arquitectura el Program Counter se obtiene de los 26 bits del campo address, de las instrucciones tipo J multiplicado por 4 del tal forma que quedan 28 bits, estos se unen con los cuatro más significativos de (PC + 4).

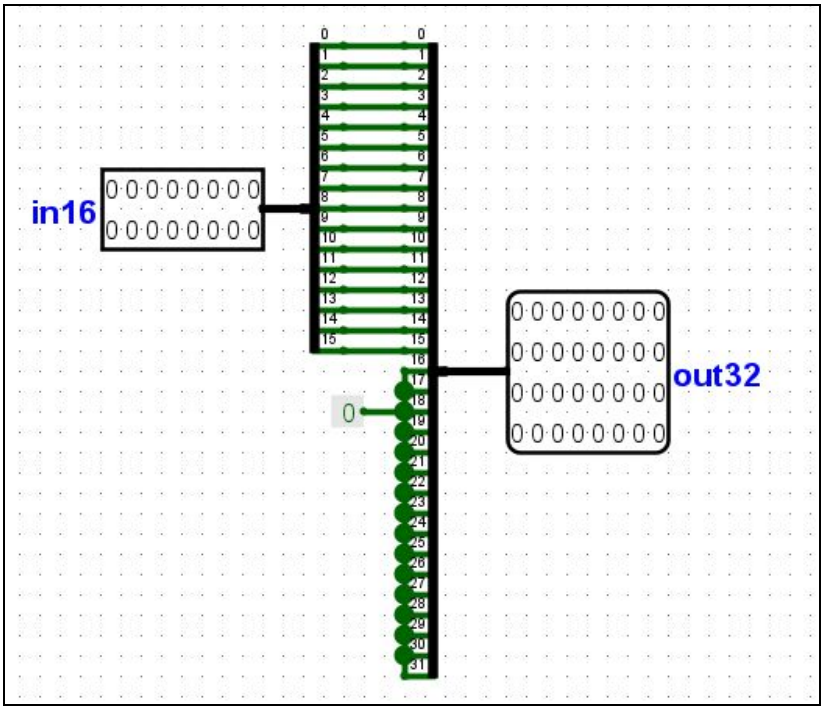
El circuito es el siguiente:



7. Extensión de Signo

Este componente toma un número de 16 bits y lo convierte en uno de 32. Este componente es para las instrucciones tipo **I** que necesitan convertir un campo inmediato de 16 bits a 32.

El circuito es el siguiente:

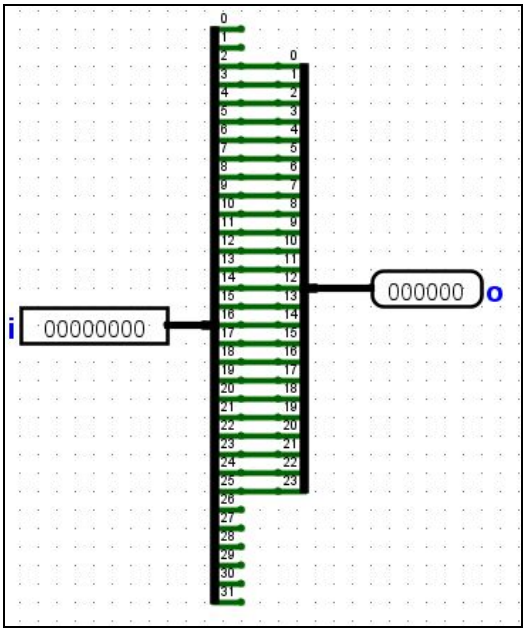


8. División por 4

Este componente es necesario porque la arquitectura MIPS accede a las instrucciones por lo que se conoce como palabra, esta está compuesta por 4 bytes. Significa que se tendrá que ir iterando en la memoria que guarda las instrucciones de a 4 Bytes.

La división por 4 en binario se libra extrayendo los dos bits menos significativos, y finalmente como

El circuito queda así:



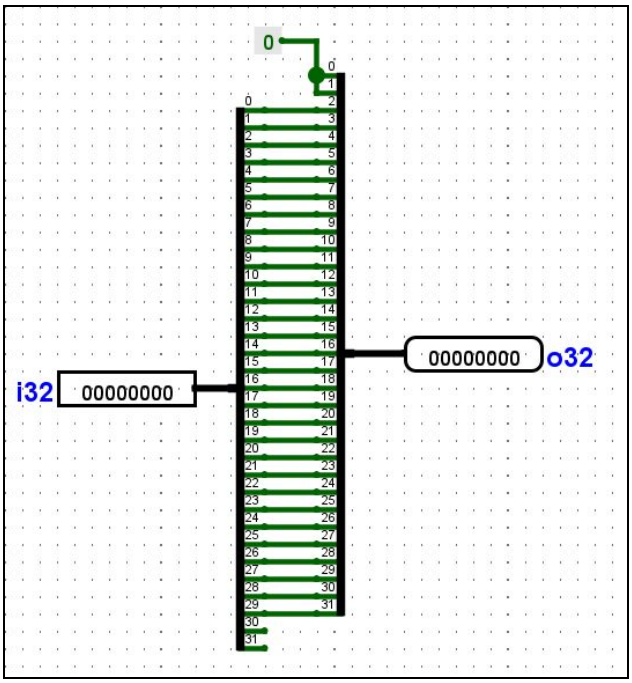
9. Desplazador de dos bits a las izquierda

Es necesario realizar un componente multiplicador por cuatro para el campo inmediato de la instrucción **beq,** el cual dice a cuantas instrucciones de la posición actual se encuentra la instrucción del salto a tomar.

En binario multiplicar por cuatro implica correr dos bits a la izquierda agregando dos ceros en la parte derecha.

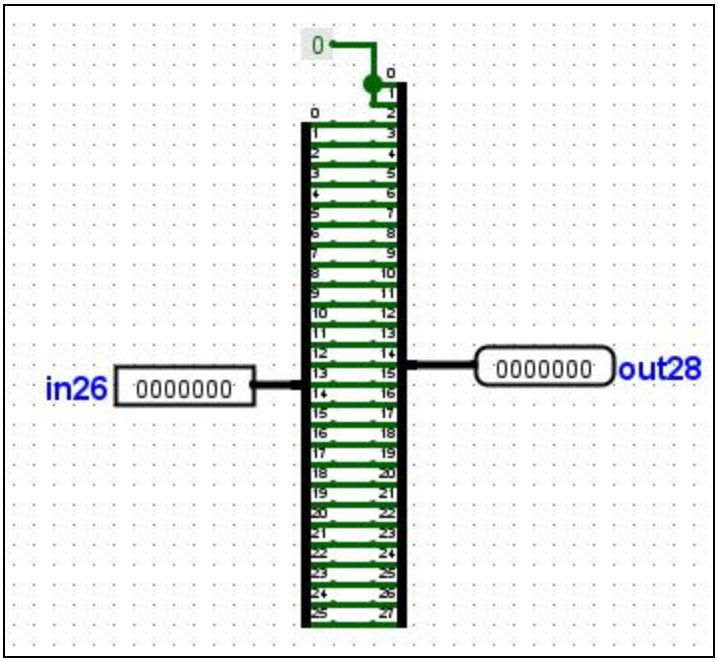
El circuito queda así:

Shift_left_2



Muy similar a este es el desplazador de 2 bits a la izquierda que de salida tiene 32 bits y es el siguiente:

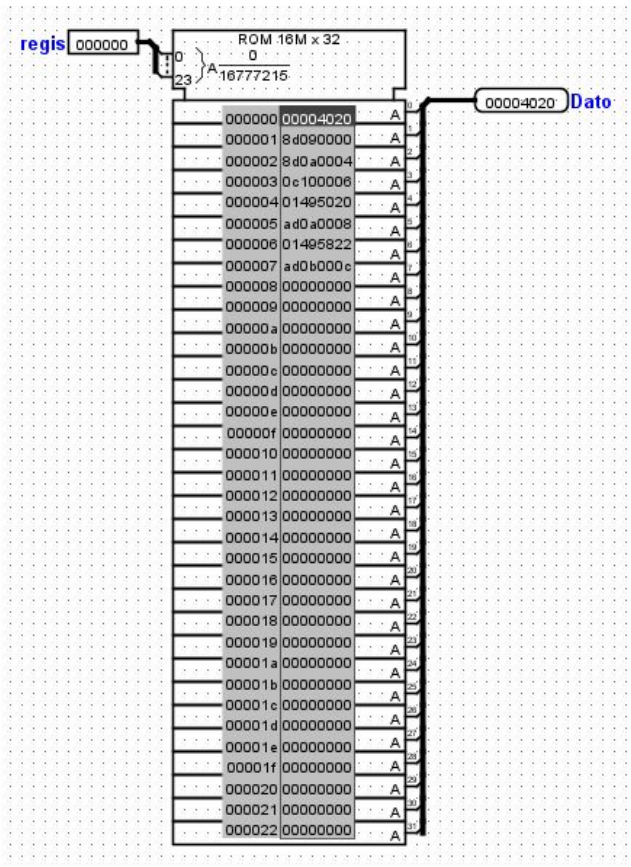
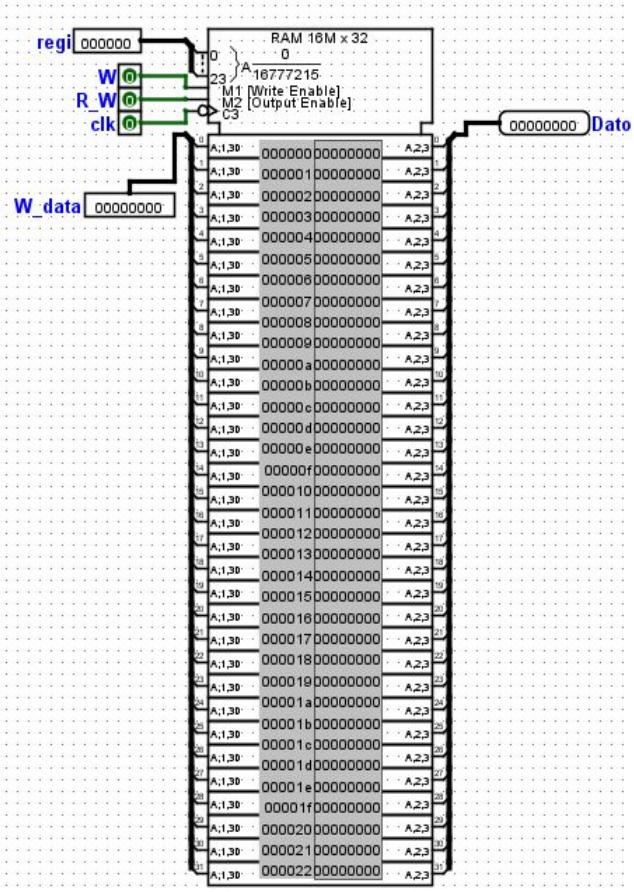
Shift_left_2_28:



10. Memoria RAM Y ROM

Estos componentes se utilizaron de la librería de Logisim, los cuales nos permiten guardar los datos y las instrucciones que se correrán el el programa.

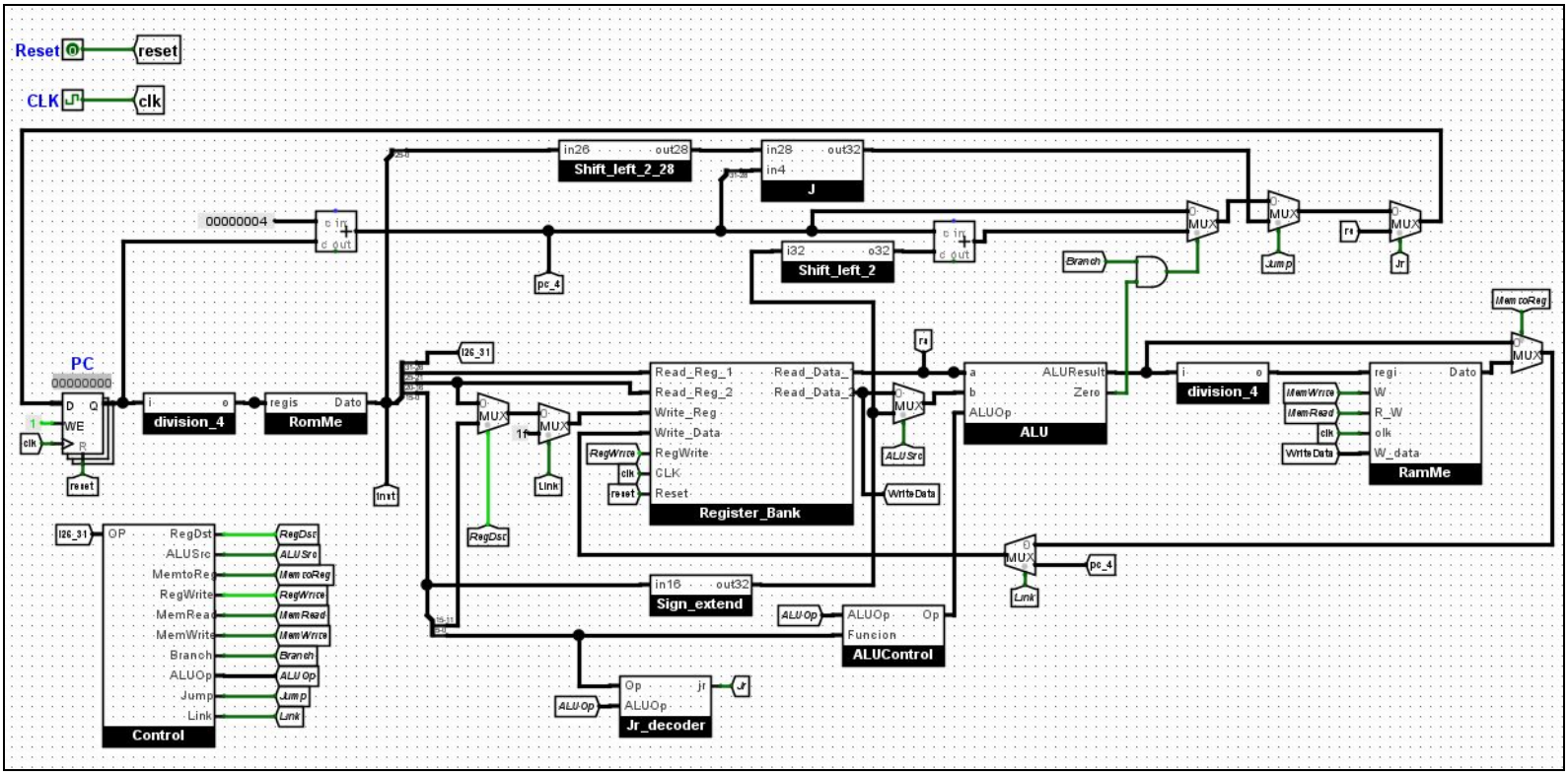
Por motivos de orden se decidió poner estos componentes en dos circuitos aparte. Los cuales serán utilizados finalmente por el circuito principal.

ROM: Memoria donde se almacenan las instrucciones que se van a correr en el programa.	**RAM**: Memoria donde se almacenan los datos que se van a utilizar en el programa.
	

11. PROCESADOR MONOCICLO

Para la conexión de los componentes anteriormente descritos se utilizaron algunos sumadores de bits y multiplexores de acuerdo a la ruta de datos de las instrucciones que se va a soportar.

El circuito final queda así:



PROBLEMA #4

El programa debe tomar una matriz de 10×10 números enteros y determinar cuáles son el mayor y el menor número almacenado en cada fila y en cada columna.

Para resolver este problema el algoritmo recorre la matriz como si fuera un vector unidimensional con 100 posiciones, el cual mediante dos contadores llamados **numf** y **numc** nos permitirán saber en qué fila y columna iteramos respectivamente simplemente contando la columna de 0 a 9, sabremos así que hemos finalizado con una fila pudiendo almacenar en ellas los valores de interés en unos vectores

destinados para eso, cada uno con 10 casillas para almacenar los resultados obtenidos mediantes comparaciones simples de menor o mayor que.

Solución en Código de alto nivel:

INICIO:

```
int[] vector = { -934, 80, 71, 819, -945, -469, 762, 790, 120, -718, -999,
-310, 647, -345, -847, -762, -584, 407, -574, -678, -583, -137, -66, -350,
-422, 580, 370, 717, 311, 54, 0, 251, 261, -979, -208, -806, 596, 215, 468,
298, -195, 644, -371, 464, 25, 784, 763, -992, -320, 516, -751, 284, 381, 53,
-881, 306, 966, 635, -174, -737, -194, 353, 59, -739, 535, -906, 193, 994,
-132, -693, -50, 559, -883, 91, -529, 140, -173, 866, 148, -28, -28, -159, 278,
-419, -989, -561, -121, 110, -636, -171, 889, 281, 783, 816, -651, 436, 500,
-509, -895, 17 };

int[] Mfila = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
int[] mfila = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
int[] Mcolumna = new int[10];
int[] mcolumna = new int[10];

int numf=0;
int numc=0;

for (int i = 0; i < 100; i++) {

   if (vector[i] < mfila[numf]) mfila[numf] = vector[i];

   if( vector[i] > Mfila[numf] ) Mfila[numf] = vector[i];

   if (vector[i] < mcolumna[numc]) mcolumna[numc] = vector[i];

   if( vector[i] > Mcolumna[numc] ) Mcolumna[numc] = vector[i];

   numc++;

   if( numc == 10){
    numc = 0;
        numf++; }

 }
```

FIN

Código en ensamblador mips

Segmento de Datos:

```
.data
contador: .word 400, 4, 28, 428, 468, 508, 548
vector: .word -934, 80, 71, 819, -945, -469, 762, 790, 120, -718, -999, -310, 647, -345, -847, -762, -584, 407, -574, -678, -583,
MayoresFila: .space 40 # 4 bytes * 10
MenoresFila: .space 40
MayoresColumna: .space 40
```

Segmento de Instrucciones:

```
.text

# Cargamos los parametros para llamar un procedimiento:
add $a0, $zero, $zero # numf
add $a1, $zero, $zero # numc
jal contarMatriz

j finalPrograma

#####################################################################
# Procedimiento contarMatriz
# Utilidad: Calcula el numero mayor y menor de las filas
# y columans dentro de una matriz 10x10
# Entrada:
# $a0 - contador de la fila
# $a1 - contador de la columna
contarMatriz:

        add $t0, $zero, $zero # contador = 0
        lw $t1, 0($zero) # cargamos en $t1 el numero 400 que es el tope.
        lw $t2, 4($zero) # cargamos en $t2 el numero 4 que sera el valor del incremento.
        lw $t3, 8($zero) # direccion base del vector.
        lw $t4, 12($zero) # direccion base del vector MayoresFila.
        lw $t5, 16($zero) # direccion base del vector MenoresFila.
        lw $t6, 20($zero) # direccion base del vector MayoresColumna.
        lw $t7, 24($zero) # direccion base del vector MenoresColumna.
```

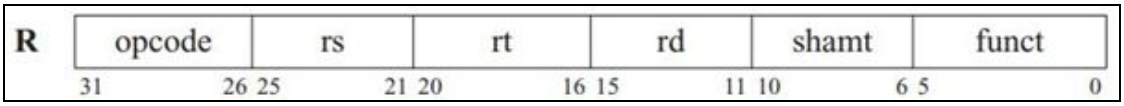
```
add $s0, $t3, $zero # copia direccion base del vector.
add $s1, $t4, $zero # copia direccion base del vector MayoresFila.
add $s2, $t5, $zero # copia direccion base del vector MenoresFila.
add $s3, $t6, $zero # copia direccion base del vector MayoresColumna.
add $s4, $t7, $zero # copia direccion base del vector MenoresColumna.

loop:   beq $t0, $t1, endLoop

        # if (vector[i] < mfila[numf])  (Menores fila)
        add $s0, $t3, $t0  # obtenemos el indice para buscar en el vector.
        lw $t8, 0($s0) # cargamos vector[i]

        add $s2, $t5, $a0 # obtenemos el indice para buscar en mfila
        lw $t9, 0($s2) # cargamos mfila[numf]

        slt $s5, $t8, $t9 # Si vector[i] < mfila[numf]

        beq $s5, $zero, endPrimerIf # Si $s5 == 0 no se cumplio el if
        sw $t8, 0($s2) # guardamos el valor de $t8 en donde esta $t9

        endPrimerIf:

         # if( vector[i] > Mfila[numf] ) (Mayores fila)
         add $s1, $t4, $a0 # obtenemos el indice para buscar en MFila
         lw $t9, 0($s1) # cargamos MFila[numf]

         slt $s5, $t9, $t8 # Si Mfila[numf] < vector[i]
```

```
            #  if( numc == 9) 588
            lw $s6, 588($zero) # cargo el valor 36 de la memoria

            beq $a1, $s6, quintoIf
            j endQuintoIf

            quintoIf:
                    add $a1, $zero, $zero # numc = 0
                    add $a0, $a0, $t2 # numf = numf + 4
            endQuintoIf:

           add $t0, $t0, $t2 # contador++
           j loop
        endLoop:

        jr $ra

finalPrograma:
add $s7, $zero, $zero
```

CÓDIGO EN MIPS A MÁQUINA

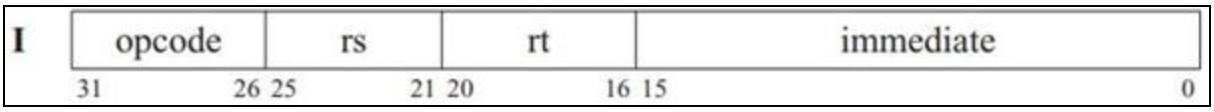
Para las instrucciones que soporta el procesador monociclo se utilizaron los formatos básicos en MIPS que son:

- Tipo R: Este formato lo utilizan las instrucciones, add, sub, or, and, nor, slt, jr

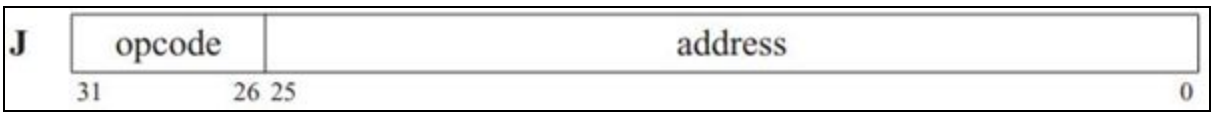


El opcode de una instrucción tipo R es : 0x000000

- Tipo I: Este formato lo utilizan las instrucciones, lw, sw, beq



- Tipo J: Este formato lo utilizan las instrucciones, jump, jump and link



Prueba

Para probar el correcto funcionamiento del procesador se hacen uso de diferentes instrucciones combinadas que usen saltos y acceso a la memoria, por ejemplo el siguiente conjunto de instrucciones MIPS:

Instrucciones MIPS:

```
.data
numeros: .word 15 25

.text
add $t0, $zero, $zero
lw $t1, 0($t0)
lw $t2, 4($t0)

jal pedazo
add $t2, $t2, $t1
sw $t2, 8($t0)

pedazo:
sub $t3, $t2, $t1
sw $t3, 12($t0)
```

Código de bajo nivel

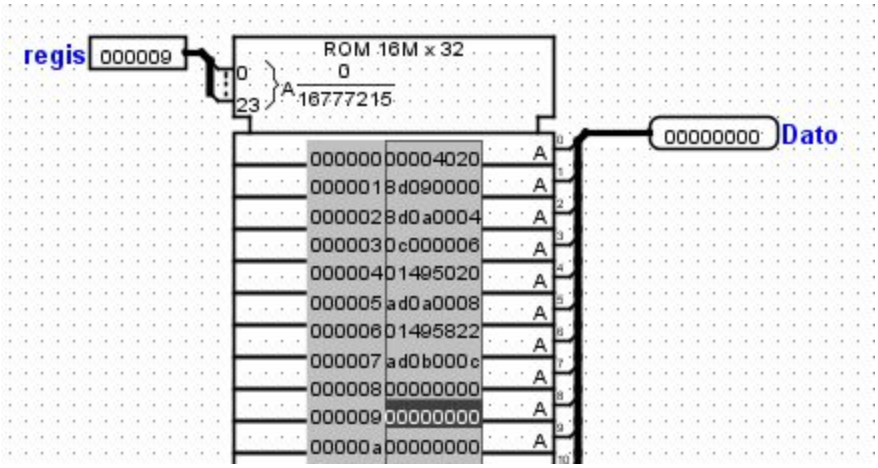
DATOS en bajo nivel:

```
v2.0 raw
0000000f
00000019
00000000
00000000
```

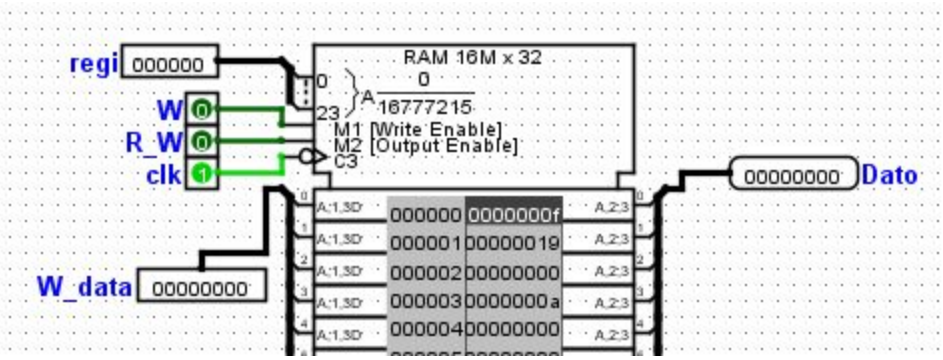
INSTRUCCIONES en bajo nivel:

```
v2.0 raw
00004020
8d090000
8d0a0004
0c000006
01495020
ad0a0008
01495822
ad0b000c
```

Instrucciones en Hexadecimal cargadas en la memoria de instrucciones:



Valores almacenados en la memoria luego de la ejecución del programa:



Conclusiones

- Cuando se desarrolla un sistema digital lo mejor es siempre pensar en un diseño jerárquico, puesto que al implementar los circuitos, se evidencia una disminución en la complejidad del diseño además de permitir un mayor entendimiento de cómo la reutilización puede anidarse a gran escala para solucionar un problema de diferente magnitud.

- Para realizar el procesador es conveniente comprender el funcionamiento de todas las instrucciones usadas junto con los formatos de instrucción para poder seguir la convención de la arquitectura MIPS32 y su recorrido dentro de la ruta de datos.

- Todos los temas vistos durante el curso fueron imprescindibles para poder llegar a diseñar el procesador monociclo brindando además fuertes bases para seguir aprendiendo las temáticas relacionadas con la línea de arquitectura de máquinas y sistemas operativos.

- Es de vital importancia saber la ruta de datos de cada una de las instrucciones que se va a implementar en el procesador para poder realizar un buen desarrollo de este.