



ESCOLA TÈCNICA SUPERIOR
D'ENGINYERIA
Universitat Rovira i Virgili



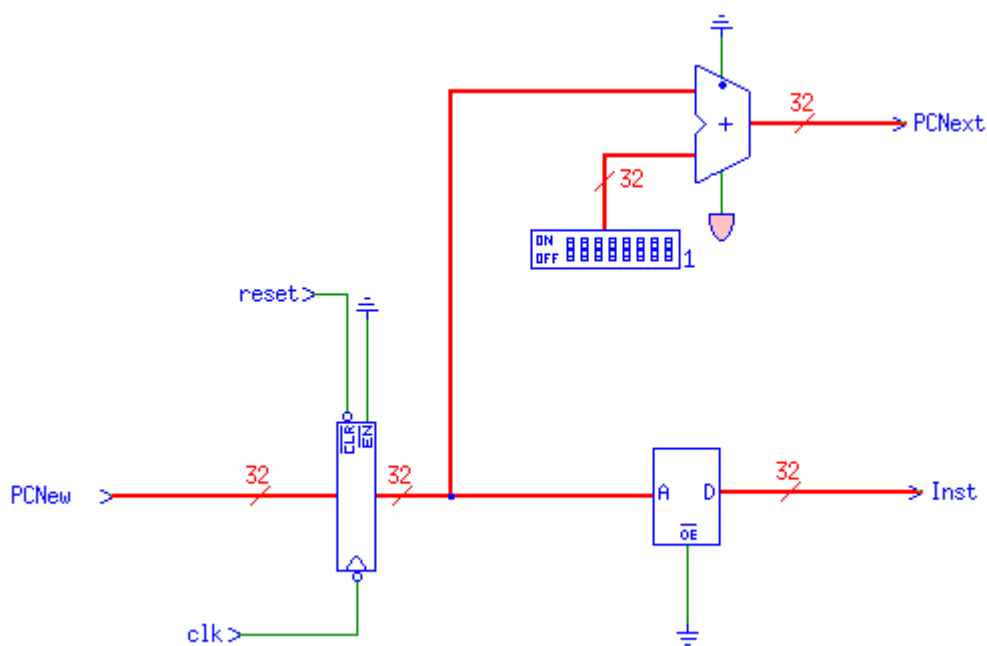
Practica 2

Estructura de computadores

Alumno/s: Dickinson Bedoya Perez
Anna Gracia Colmenarejo
Enseñamiento: Ingeniería Informática
Profesor/es: Carlos Aliagas
Carlos Molina
Fecha: 04/2021

ÍNDICE

FASE 1: Fetch y Read	2
TAREA 1: Fetch	2
TAREA 2: Read	5
FASE 2: Execute y Mem	6
TAREA 3: Ejecución de instrucciones aritmético-lógicas (Formato R)	6
TAREA 4: Ejecución de instrucciones básicas de formato I y J (beq y j)	10
TAREA 5: Instrucciones de acceso a memoria, lw y sw (Formato I)	12
FASE 3: Unidad de proceso y unidad de control	13
TAREA 6: Unidad de Proceso del procesador MIPS	13
TAREA 7: Unidad de Control del Procesador MIPS	14
FASE 6: Prueba de funcionamiento y evaluación del rendimiento	18
TAREA 10: Prueba del repertorio ISA básico	18
TAREA 11: Programa en ensamblador y evaluación de todas las instrucciones de la ISA	19



JUEGO DE PRUEBAS

Para probar el funcionamiento del circuito hemos cargado en la memoria ROM el programa *mult.mem* que es un programa de instrucciones codificadas en hexadecimal. Como el fetch se encarga de devolver la instrucción que se tiene que ejecutar hemos comprobado que se fuesen pasando correctamente las instrucciones. Hemos colocado una barra de leds que mostrará la instrucción codificada. También hemos puesto otras barras de leds para poder comprobar los diferentes rangos de los bits dentro de una instrucción.

CICLOS DE RELOJ	INSTRUCCIÓN	INSTRUCCIÓN CODIFICADA ESPERADA	INSTRUCCIÓN CODIFICADA RECIBIDA
1	lw	8c090064	8c090064
2	lw	8c090065	8c040065
3	lw	8c090066	8c050066
4	and	00008024	00008024
5	and	00008824	00008824
6	slt	0224402a	0224402a
7	beq	11000003	11000003

8	add	02058020	02058020
9	add	02298820	02298820
10	beq	1000fffb	1000fffb
11	sw	ac100067	ac100067
12	beq	1000ffff	1000ffff

A continuación ponemos los rangos de bits de algunas de las instrucciones del programa:

8C050066

El campo OP indica que es una operación de referencia de memoria y el FUNCT especifica que es la operación **lw**.

OP	RS	RT	RD	SHAMT	FUNCT
100011	00000	01001	00000	00001	100110
100011	00000	01001	00000	00001	100110

00008824

El campo OP indica que es una operación aritmético-lógica y el FUNCT especifica que es la operación **and**.

OP	RS	RT	RD	SHAMT	FUNCT
000000	00000	00000	10001	00000	100100
000000	00000	00000	10001	00000	100100

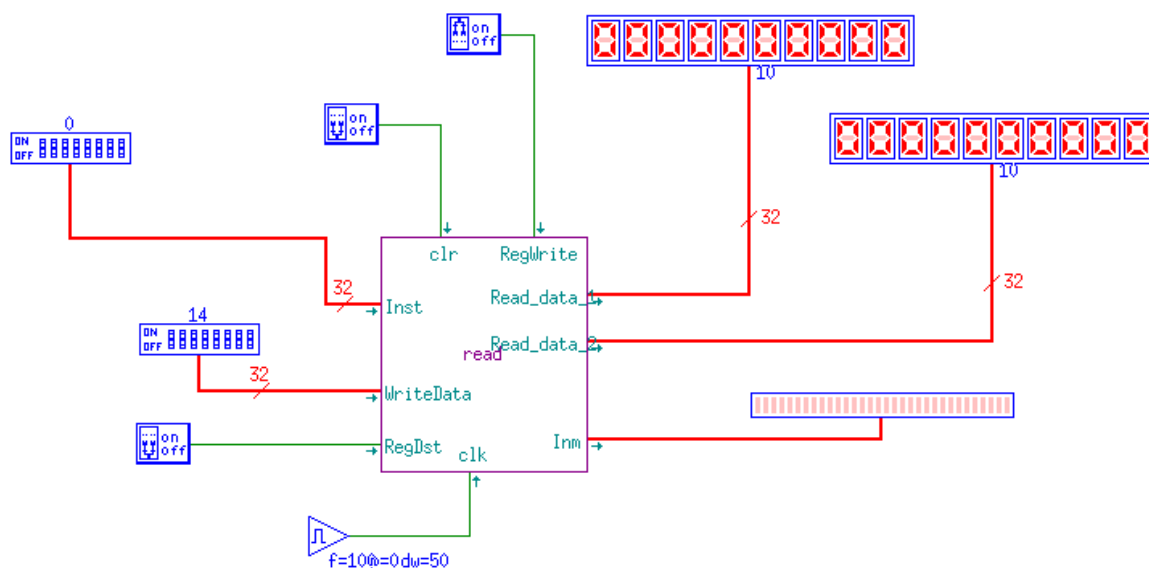
11000003

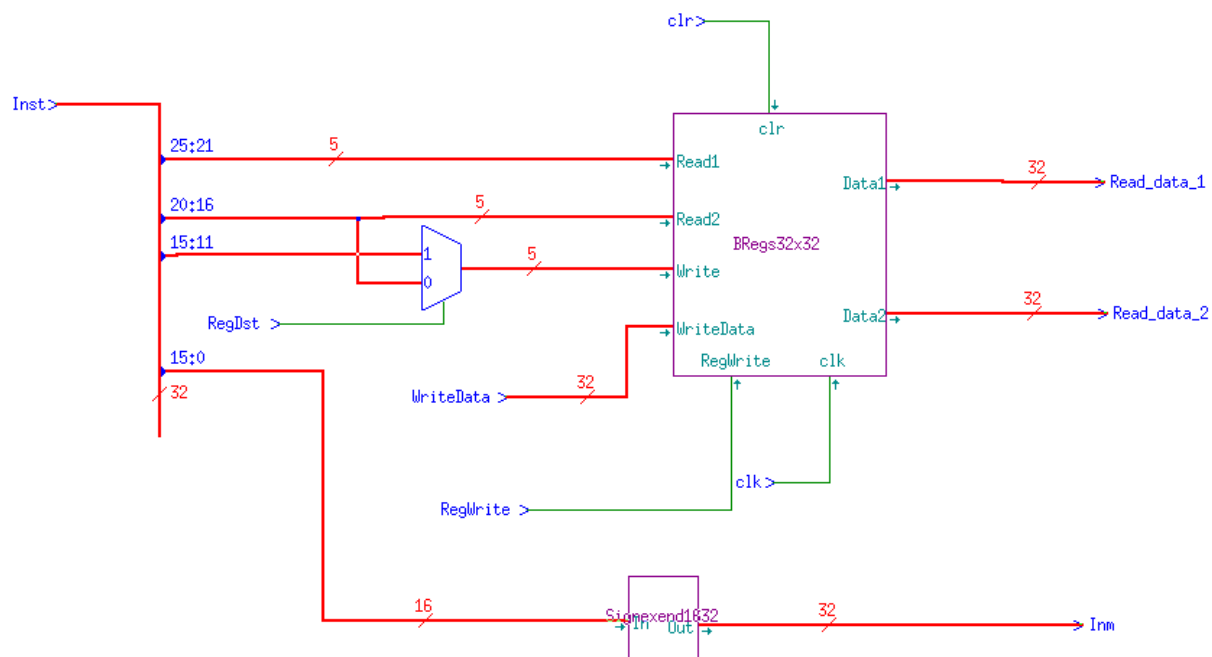
El campo OP indica que es una operación de control de flujo y el FUNCT especifica que es la operación **beq**.

OP	RS	RT	RD	SHAMT	FUNCT
000100	01000	00000	00000	00000	000011
000100	01000	00000	00000	00000	000011

TAREA 2: Read

El *read* es el conjunto de componentes conectados que implementan la lectura de registros. Para implementarlo hemos utilizado el banco de registros del fichero **BancoRegs.v**, este es capaz de leer el contenido de dos registros al mismo tiempo y además es capaz de escribir en un tercer registro. Por eso tiene 2 entradas de 5 bits por las que se indicará qué registros hay que leer, una entrada de 5 bits que indicará qué registro se quiere escribir y una entrada de 32 bits que será por la que entrarán los datos que se quiere escribir en el registro. Y tendrá dos salidas de 32 bits cada una con los datos de los 2 registros que se han leído. Para que este banco de registros sepa qué registros tiene que leer tenemos que conectar los bits que codifican la instrucción. Es decir, de los 32 bits que codifican la instrucción tendremos que pasarle los 5 bits que corresponden al primer registro fuente *RS* a la entrada *Read register 1*, los 5 bits del registro segundo registro fuente *RT* a la entrada *Read register 2*, y por último los 5 bits del registro destino *RD* a la entrada *Write register*. Esta etapa se conoce como *Decode*. El banco de registros también consta de una señal de 1 bit *RegWrite* que le indica cuando debe escribir en el registro, esta señal es generada por la unidad de control. Este procesador siempre va a leer el banco de registros aunque no sea necesario, por ejemplo una instrucción *jump* no precisa de ningún registro sin embargo los bits le llegarán igual a las entradas del banco pero no se escribirá en ningún registro.





FASE 2: Execute y Mem

TAREA 3: Ejecución de instrucciones aritmético-lógicas (Formato R)

Para ejecutar instrucciones aritmético-lógicas necesitamos implementar una unidad que las pueda realizar. Las instrucciones aritmético-lógicas siguen el formato R y un ejemplo sería realizar una suma o una resta.

Formato R (register): add, sub, and, or, slt

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

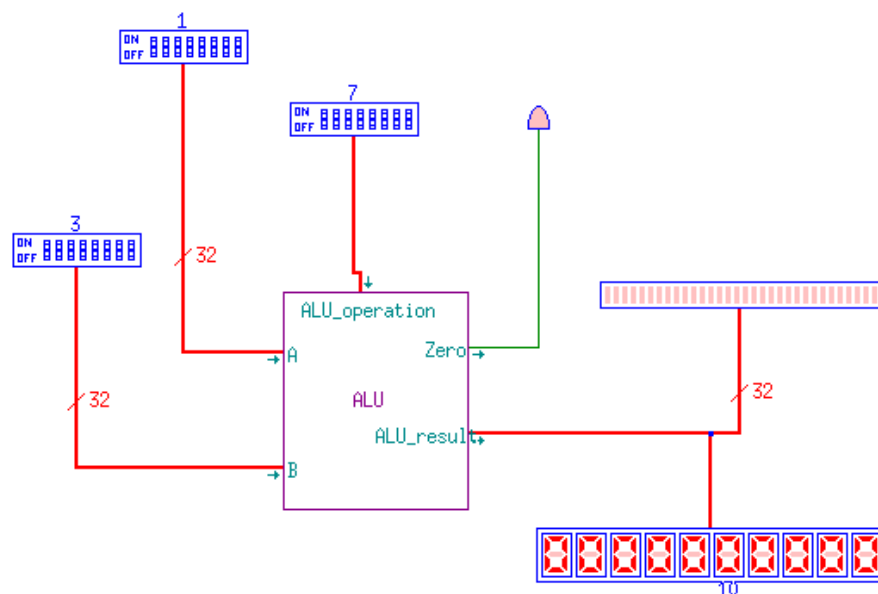
Esta unidad que realizará las operaciones será una ALU, la cual tendrá 3 entradas. La primera será la *ALU_op* que es de 4 bits y viene de la unidad de control, esta entrada le indica a la ALU que operación debe realizar. En la siguiente tabla se muestra la codificación de todas las operaciones aritmético-lógicas que realiza la ALU:

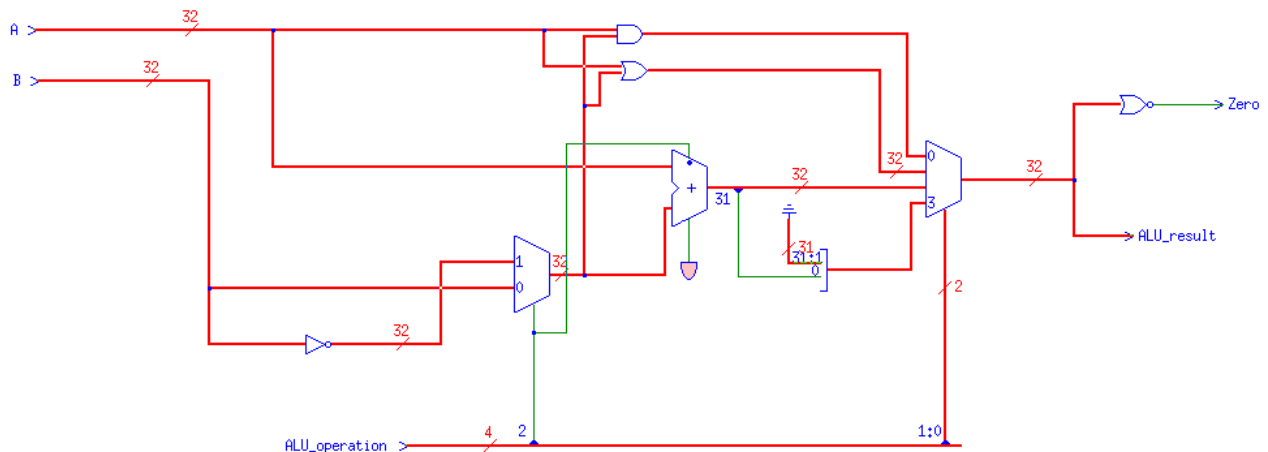
OPERACIÓN	SEÑAL DE CONTROL
-----------	------------------

A AND B	0000
A OR B	0001
A + B	0010
A - B	0110
A < B	0111

La segunda entrada *A* es el valor del primer operando que le vendrá del banco de registros (*Read_Data_1*). Y por último, la tercera entrada será el segundo operando que vendrá también del banco de registros (*Read_Data_2*).

Por otro lado, tiene 2 salidas. La primera salida será *ALU_result* la del resultado de la operación que haya realizado y posteriormente se llevará a la entrada *Write_Data* del banco de registros para que se guarde. La otra salida informa de si ha sido cero el resultado de la operación.





La elección de operaciones se hace a partir de los 2 bits de menor peso (0 y 1) pudiendo así aprovechar entradas en el multiplexor y ahorrar área y retardo, usando un multiplexor de 4 entradas para los resultados de las operaciones en vez de uno de 8 entradas.

En este caso se aprovecha la misma entrada para la suma y la resta ya que los dos bits de menor peso son 10, pero para diferenciar entre estas dos se coge el bit 2, que puede ser 1 o 0 según la operación que sea y así poder elegir entre una de las dos señales de B.

La operación **AND** la hemos implementado llevando las 2 entradas A y B a una puerta AND.

La operación **OR** la hemos implementado igual que la AND pero con una puerta OR.

Las operaciones de **suma (+)** y **resta (-)** las hemos combinado porque hacer $A - B$ es lo mismo que hacer $A + (-B)$. Entonces siempre utilizamos un sumador que sumará A con B positiva o negativa. El signo de B lo decidirá un multiplexor al que le entrará B y B en negativo ya que previamente habremos cambiado su valor. Cuando el bit 2 de la instrucción es 1 se precisa de una B en negativo.

Para saber si $A < B$ cogemos el bit 31 de la operación $A + (-B)$ ya que es el bit de signo. Si este bit es 1 significa que el resultado ha sido negativo y por lo tanto $A < B$ y devolvemos 1. Si no será el caso contrario donde $A > B$ y devolveremos 0.

Y finalmente, llevamos los resultados de todas las operaciones a un multiplexor que elegirá el resultado de la operación que se le indica.

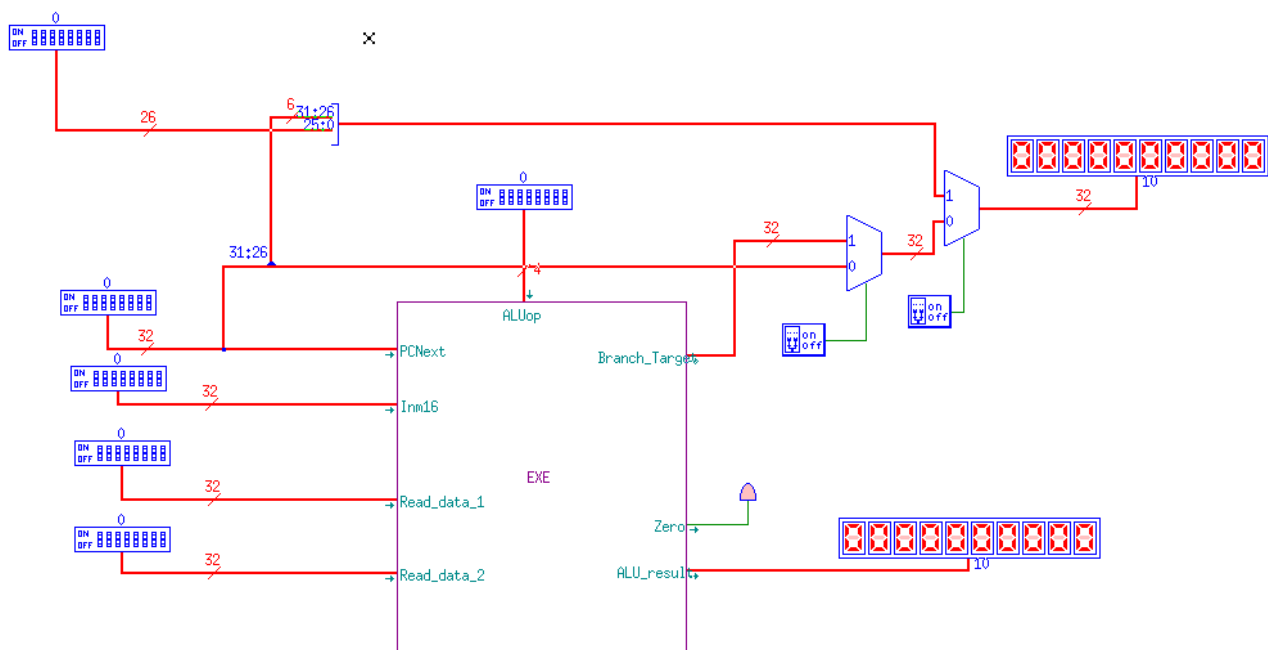
JUEGO DE PRUEBAS

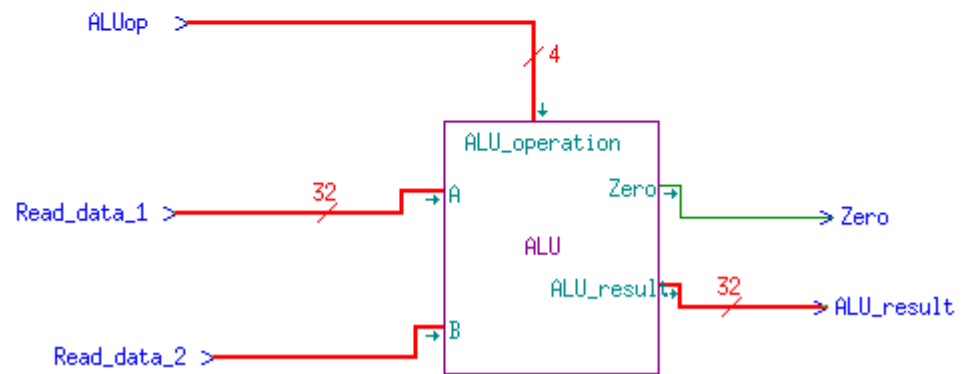
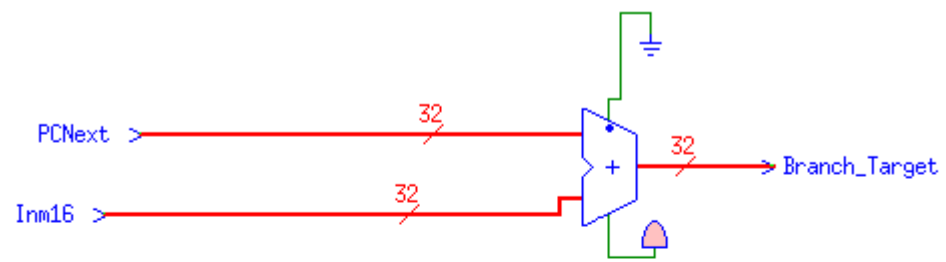
ENTRADAS			RESULTADOS	
$A_{(hex)}$	$B_{(hex)}$	OPERACIÓN	ALU RESULT _(hex)	ZERO
0	0	0000	0	1
5	5	0000	5	0
24	24	0000	24	0
1	4	0000	0	1
0	0	0001	0	1
5	5	0001	5	0
24	24	0001	24	0
1	4	0001	5	0
0	0	0010	0	1
5	5	0010	A	0
24	24	0010	48	0
1	4	0010	5	0
0	0	0110	0	1
5	5	0110	0	1
24	24	0110	0	1
4	1	0110	3	0
0	0	0111	0	1
5	5	0111	0	1
24	24	0111	0	1
1	4	0111	1	0

TAREA 4: Ejecución de instrucciones básicas de formato I y J (beq y j)

Las instrucciones de formato I y J, beq y j, son instrucciones que indican un salto en la ejecución del programa, es decir, si hay una de estas instrucciones en el programa la siguiente instrucción que se realice no será la indicada por $PC + 4$. En el caso del beq será $PC + 4 + \text{dirección}$, y en el del jump se cogerán los 6 de más peso del PCnext (31:26) y los 26 bits de la dirección que se indica en el jump (0-25).

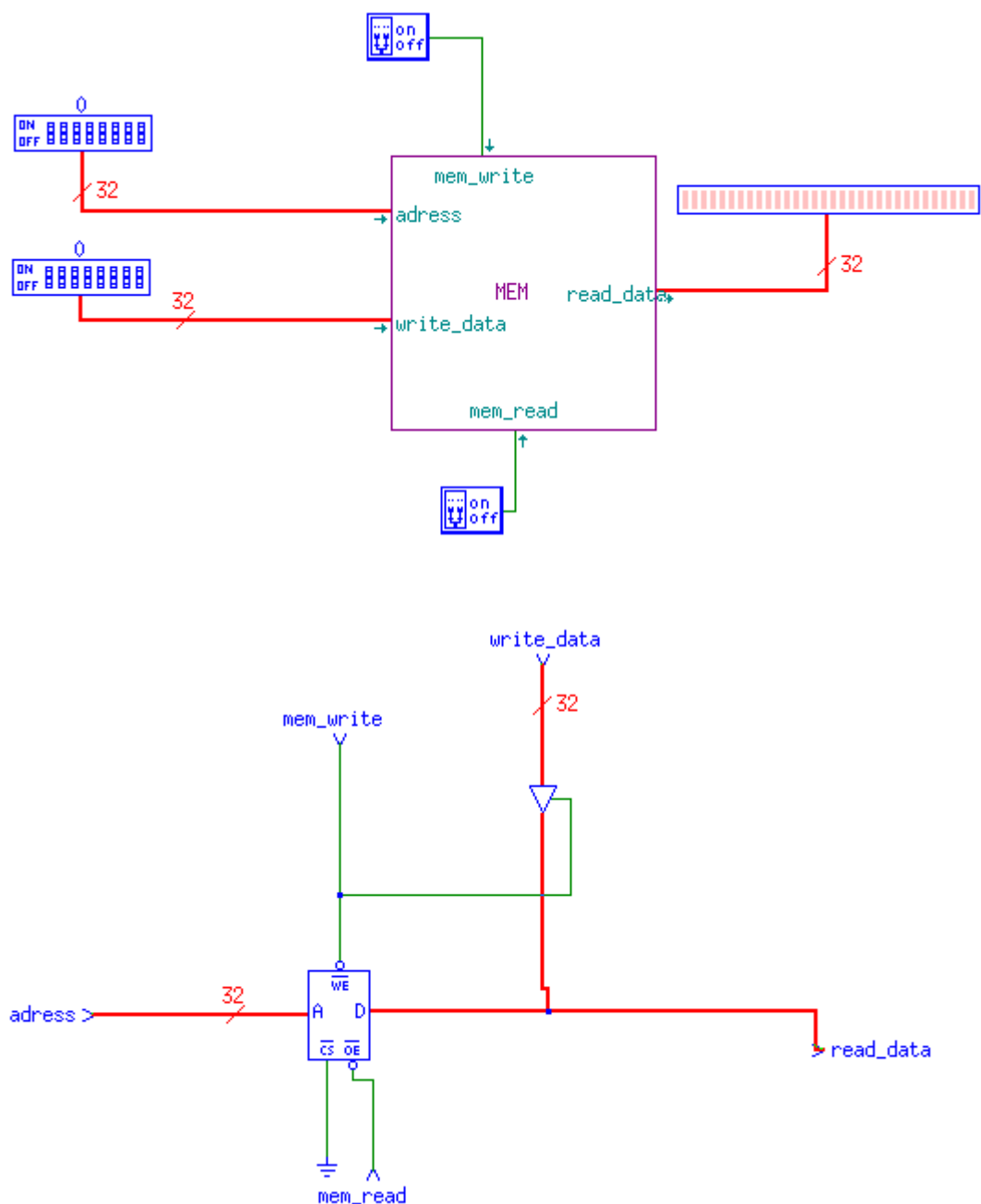
Para la implementación de esta parte hemos juntado en un módulo llamado EXE la ALU y el cálculo de este $PC + 4 + \text{dirección}$ (en el tkgate es $PC + 1 + \text{dir}$), este cálculo se realiza mediante una ALU+ en la que suma las dos entradas PCnext y Inm26, el resultado saldrá por la salida Branch_Target. El Branch_Target se llevará a un multiplexor junto con el valor de PCnext, ya que depende de la instrucción se tendrá que elegir un valor u otro. Posteriormente, la salida de este mismo multiplexor se llevará a otro con el resultado de juntar PCnext y el inmediato. La unidad de control es la responsable de enviar las señales a los multiplexores para que elijan la salida correcta.





TAREA 5: Instrucciones de acceso a memoria, lw y sw (Formato I)

El módulo MEM básicamente se comporta de una memoria RAM. En modo lectura, la RAM debe tener el **output enable**, el **chip select** en 0 y el **write enable** en 1. Con estos valores, el contenido indicado por el bus A sale por el bus D. En modo escritura, el chip select y el write enable a 0 y el output-enable a 1. Con estos valores, el bus D ahora es de entrada. Aquello indicado por el bus D será almacenado en la posición indicada por el bus A.



Como siempre queremos que el chip select esté a 0, usamos una conexión a tierra para tenerlo siempre a 0. A parte, como los valores de lectura y escritura van “invertidos”, las entradas de write enable y output enable están invertidas. Para el hecho de que el bus D sea de entrada y salida a la vez, se usa un *Tri-state Buffer* para controlar esto.

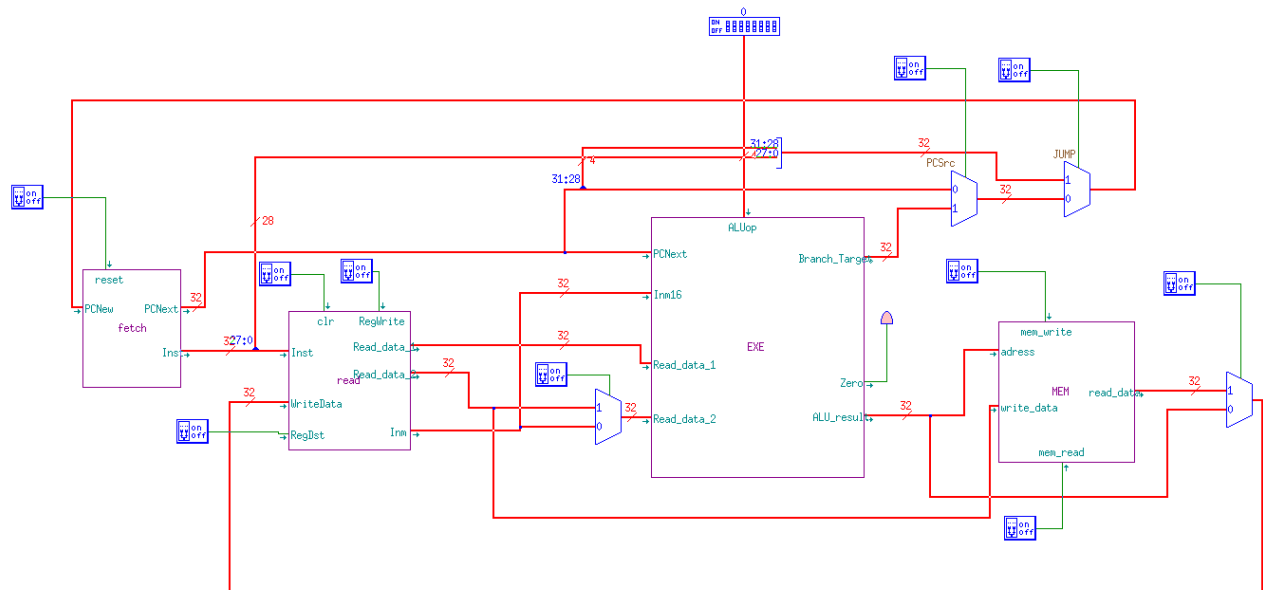
Con estas conexiones podemos almacenar y obtener datos de la memoria para poder ejecutar los programas.

FASE 3: Unidad de proceso y unidad de control

TAREA 6: Unidad de Proceso del procesador MIPS

La unidad de proceso es la encargada de ejecutar todos los programas, es decir, leer instrucciones, actualizar el contador del programa, calcular datos y direcciones, almacenar en registros y en memoria.

La característica de este procesador escalar monociclo que hemos implementado es que este lee una instrucción de memoria y hasta que esta instrucción no está ejecutada al completo no pasará a la siguiente. El problema es que necesita saber cuándo escribir en memoria, cuándo saltar a otra instrucción, etc, de esto se encargará la unidad de control. Para implementarla la unidad de proceso hemos unido todos los módulos hechos anteriormente que juntamente hacen todo lo necesario para poder ejecutar los programas. El fetch calcula el contador de programa y lee la instrucción correspondiente a este contador, esta instrucción se pasará al read el cual leerá los valores de los registros que se precisen o escribirá en un registro. Después los valores de estos registros se pasan al exe que realizará los cálculos con la ALU o hará un salto, y finalmente los resultados de la ALU se llevarán al MEM donde si se le indica escribirá en memoria o leerá.



TAREA 7: Unidad de Control del Procesador MIPS

Como se ha dicho anteriormente, la unidad de control es la encargada de decirle a la unidad de proceso que hay que hacer en cada momento, activando y desactivando los flags necesarios para poder realizar las instrucciones dadas. Dependiendo de las instrucciones que le lleguen, la unidad de control activará y desactiva los flags necesarios para la correcta ejecución, además de calcular e indicar la operación que debe realizar la ALU.

El **jump** se debe activar cuando haya una instrucción de salto incondicional.

El **RegDst** indica cuál de los dos registros es el que se va a usar en la entrada write del banco de registros

El **Branch** se debe activar cuando haya una instrucción de salto

El **MemRead** indica al módulo mem que la ram debe estar en modo lectura.

El **MemToReg** indica al multiplexor de la salida del módulo mem si debe coger el valor de la memoria o directamente el resultado de la ALU.

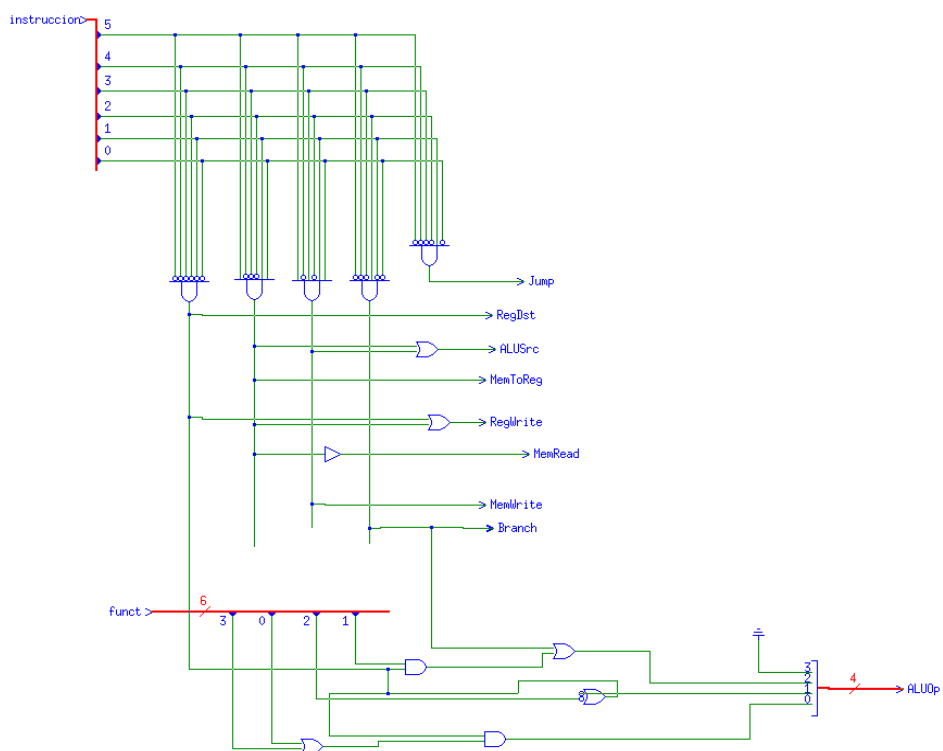
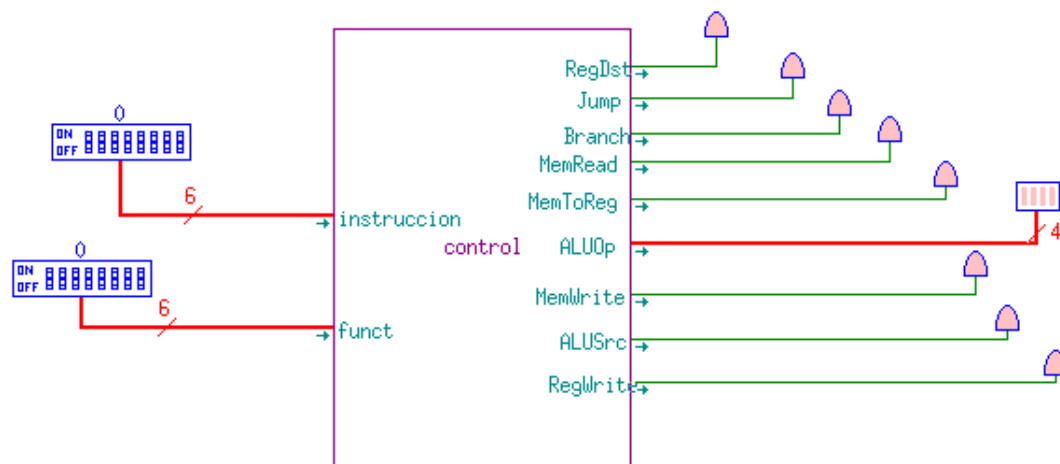
El **ALUOp** es de 4 bits y le indica a la ALU dentro del exe cuál de sus 5 operaciones debe realizar.

El **MemWrite** indica al módulo mem que debe escribir en la memoria.

El **ALUSrc** indica qué valor del banco de registros entre el read data 2 y el inmediato debe entrar en el módulo exe.

El **RegWrite** se activa cuando hay que escribir en un registro.

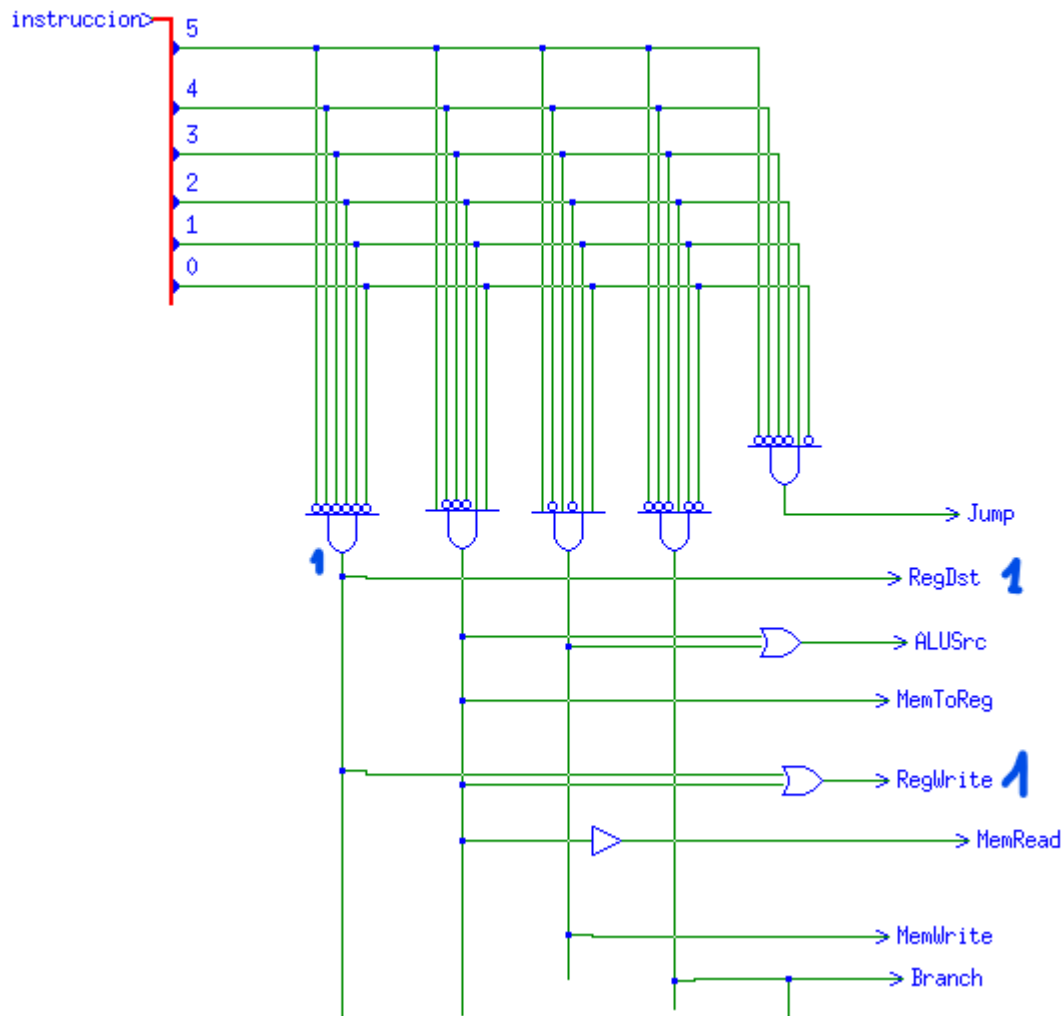
Estos flags de control se calculan mediante puertas lógicas a partir de los 6 bits de más peso de la instrucción que indica la operación (campo OP).



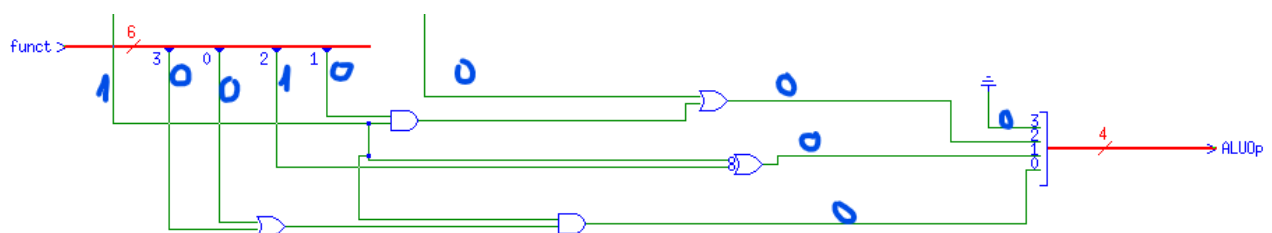
JUEGO DE PRUEBAS

Para comprobar el funcionamiento de UC primero hemos mirado que los flags se calculaban a partir de las operaciones que se indicaban. Por ejemplo, con la operación AND tenemos que los 6 bits de mayor peso son todo 0's. Con esto se puede ver el paso por cada puerta viendo que solo se deben activar el RegDst y el RegWrite. Como se

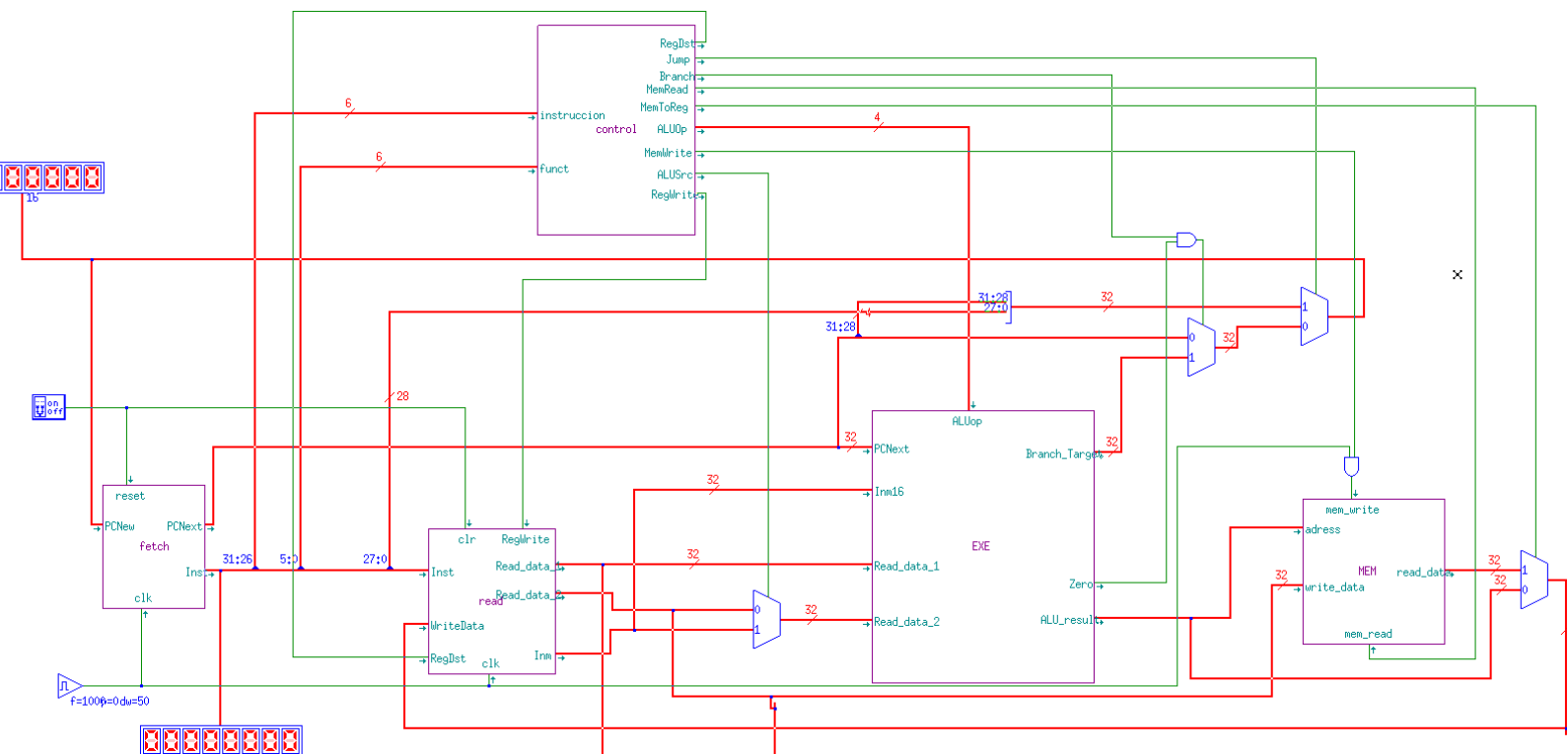
puede ver en la foto con los bits de operación y las puertas lógicas solo se activan los flags mencionados.



Para la parte del cálculo del ALUOp se depende de los 6 bits de menor peso (aunque de estos solo usemos del 0 al 3 ya que el 4 y el 5 no se usan), los que indican el **Funct**. Con una serie de puertas lógicas se saca el resultado que debería tener ALUOp. En este ejemplo, con la operación AND tenemos que ALUOp = 0000, que sería correcto ya que es lo que espera la ALU para hacer una AND.



OP	FUNCT	Regdst	jump	branch	Mem Read	MemtoReg	aluop	mem write	ALU Src	Reg Write
LW 23h	X	0	0	0	1	1	0010	0	1	1
BEQ 4h	X	0	0	1	0	0	0110	0	0	0
ADD 0h	20h	1	0	0	0	0	0010	0	0	1
AND 0h	24h	1	0	0	0	0	0000	0	0	1
SLT 0h	2Ah	1	0	0	1	1	0111	0	0	1
OR 0h	25h	1	0	0	1	1	0001	0	0	1
SUB 0h	22h	1	0	0	1	1	0110	0	0	1



Según el tkgate, el retardo total del circuito es de 566 y el área es de 27876. Con el valor del retardo podemos ajustar el valor del clock para los ciclos de reloj.

Lo hemos ajustado a una frecuencia de 570 y función del 90%. Poner la frecuencia del reloj con el mismo valor que el retardo no está bien ya que puede llegar antes la señal de escribir en memoria que el dato a escribir, por eso se debe posponer lo más que se pueda la escritura ya que se escribe cuando hay un flanco ascendente, por ello hemos ajustado el ciclo lo máximo posible modificando su frecuencia y la función. Con estos valores la escritura en memoria llega justo a tiempo.

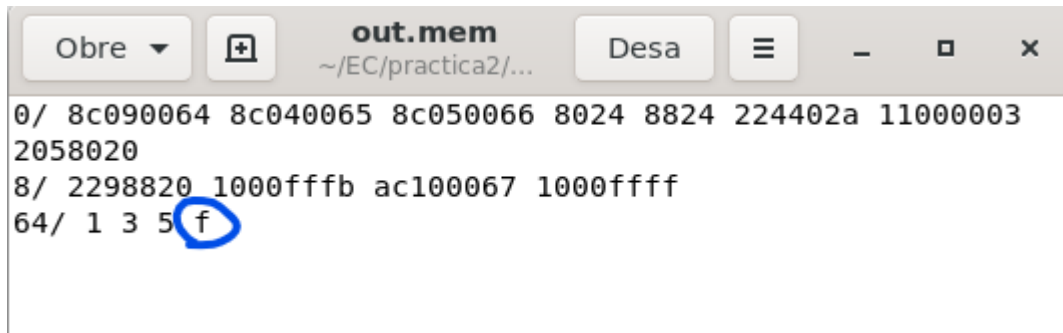
FASE 6: Prueba de funcionamiento y evaluación del rendimiento

TAREA 10: Prueba del repertorio ISA básico

Para poder probar el procesador hemos usado el programa de prueba mult.asm y su versión para las memorias **mult.mem**. El resultado de este programa es la multiplicación de $3 \times 5 = 15_d = F_h$. El resultado de este valor se ve cuando extraemos un nuevo archivo .mem del módulo mem. En ese archivo debería estar el valor indicado.

Para agilizar la ejecución del programa hemos hecho un script de ejecución que carga los archivos en las memorias y así no tener que estar poniendo estos archivos cada vez que queremos ejecutar el programa.

Cuando el programa llega a la última instrucción, un bucle infinito, es cuando podemos extraer el archivo y poder si realmente ha ido bien la ejecución del programa.



```
Obre ▾ [icon] out.mem ~/EC/practica2/... Desa [icon] [icon] [icon] [icon]
0/ 8c090064 8c040065 8c050066 8024 8824 224402a 11000003
2058020
8/ 2298820 1000ffffb ac100067 1000ffff
64/ 1 3 5 f
```

Como se ve en la foto, el archivo out.mem contiene el resultado de la multiplicación, es decir, el resultado es el correcto.

TAREA 11: Programa en ensamblador y evaluación de todas las instrucciones de la ISA

Hemos hecho un programa que intenta usar casi todas las instrucciones que puede ejecutar el procesador.

Se quiere hacer una multiplicación de dos números A y B, almacenar esta en una variable nueva S y comprobar si esta es mayor o menor que 10. En el caso de que sea mayor, se le suma 5; en el caso de que sea menor se le restan 5.

`s = a * b`

`si s > 10, s = s + 5`

`sino s = s - 5`

Como no podemos ejecutar este programa directamente, usamos la herramienta proporcionada en clase (Online MIPS Assembler) para tener el archivo .mem y poder ejecutar el programa.

El archivo .mem es el siguiente:

```
#main
00/ 8c090064 8c040065 8c050066 8c060068 8c070069 00008024 00008824
#per
07/ 0224402a 11000003 02058020 02298820 1000ffffb
#fiper
0c/ 0206502a 11400002 02078022 10000001
#suma
10/ 02078020
#salva
11/ ac100067
#fi
12/ 1000ffff

#DATA IN MEMORY
00000064/ 00000001 00000006 00000003 00000000 00000010 00000005
```

En que la primera parte es lo que ejecuta el fetch, donde estan todas las instrucciones y la parte de DATA IN MEMORY es lo que ejecuta el mem, todo lo que está almacenado en memoria.

En este caso, los valores que tenemos de A y B son 6 y 3 respectivamente. $S = 6 * 3 = 18$. Como el valor de S es mayor que 10, a S se le suman 5 más, quedando $S = 23_d = 17_h$. Para comprobar si el procesador lo ejecuta correctamente obtenemos un archivo out.mem que contiene los valores de memoria al llegar al bucle infinito, que es el siguiente:

```
0/ 8c090064 8c040065 8c050066 8c060068 8c070069 8024 8824 224402a
8/ 11000003 2058020 2298820 1000ffffb 206502a 11400002 2078022 10000001
```

```
10/ 2078020 ac100067 1000ffff
64/ 1 6 3 17
68/ 10 5
```

Como se puede ver, se obtiene el valor de $S = 17_h = 23_d$, por lo tanto el funcionamiento es el correcto. También hemos hecho más pruebas con el mismo programa pero con diferentes valores para probar ambas condiciones y ver que tanto la suma como la resta funcionan correctamente.