Proyecto 2 – 2023 – Jerarquía de memoria de datos

Álvaro de Francisco Nievas - 838819

Guillermo Bajo Laborda - 842748

Índice

Breve descripción del proyecto

En este segundo proyecto de la asignatura se ha implementado una **jerarquía de memoria para el procesador MIPS** desarrollado en el proyecto anterior.

Esta jerarquía está compuesta por:

1. Dos **memorias** principales, una **clásica** y otra **scratch**, más rápida que la anterior.
2. Memoria cache de **dos vías**.
3. Un periférico de entrada salida (IO_MASTER)
4. **Bus semi-síncrono** que comunica los masters (IO_MASTER y MIPS) con las memorias (esclavos).
5. **Árbitro** que distribuye el acceso de los masters al bus.

La memoria cache es asociativa con dos vías, cada una de ellas tiene 4 bloques de 4 palabras. Funciona con escritura directa (**write-through**) y la política de fallo en escritura es **fetch on write miss**. La política de reemplazo es **FIFO**.

La memoria **Scratch** de datos es **pequeña y rápid**a, no tiene fallos, no se copian bloques, no se actualizan las escrituras, es una memoria en cierto modo independiente de los datos de la memoria cache. En cambio, es el controlador de la memoria cache el encargado de gestionar tanto la lectura como la escritura de palabras en la Scratch MD.

El **IO_Master** es un periférico de entrada salida que monitoriza la entrada del sistema IO input y la escribe una y otra vez en la primera palabra de la MD Scratch.

Respecto al procesador, se ha usado el mismo procesador que en el primer proyecto, sustituyendo la memoria de datos antigua por el nuevo subsistema de memoria.

Para completar la jerarquía de memoria, se ha diseñado el **autómata** de estados del controlador de la memoria cache que controla las señales del sistema de memoria. Una vez el comportamiento del sistema de memoria era adecuado, se ha integrado la jerarquía en el MIPS. Para ello, hemos modificado ciertas señales del procesador, así como parar_EX_internal, de la unidad de detención, y diversos contadores que se especifican en el enunciado.

Diagrama de estados de la unidad de control

1. Inicio

Estado inicial y de reposo de la máquina. Cuando la MC se encuentra en este estado no quiere ni está usando el bus semi-síncrono.

Seguirá en este estado si recibe del MIPS una petición de lectura de un bloque que se encuentra en cache (RE and HIT), y se mostrará el contenido de la cache (mux_output="00"). También cuando acceda a una dirección no alineada (RE and unaligned), caso en el que se cargará la dirección en el registro addr_error_reg usando la señal load_addr_error. Cuando se acceda a la dirección de memoria mapeada a este registro (RE and internal_addr) se activará la señal mux_output="10" que muestra el contenido del registro. En cualquiera de estos casos se activará la señal ready.

Por el contrario, **pasará al estado de arbitraje** si recibe del MIPS una petición de escritura o una lectura de un bloque que no está en cache (WE or (RE and not HIT)). Se activará entonces la señal Bus_req que indica la petición al árbitro del bus.

2. Arbitraje

En este estado la MC está pidiendo al árbitro acceso al bus. **Seguirá en el estado** mientras el árbitro no le conceda acceso (not Bus_grant). Cuando se le conceda acceso al bus (Bus_grant) enviaremos la dirección y la operación que queremos realizar. Solo escribiremos cuando haya hit y la dirección no sea cacheable (MC_send_addr_ctrl, MC_bus_Rd_Wr=WE and (hit or addr_non_cacheable)). Si ninguna de las memorias reconoce la dirección como suya (not Bus_DevSel) activaremos la señal ready y load_addr_error, que carga la dirección en el registro de error. Si una de las memorias reconoce la señal como suya (Bus_DevSel) se pasará a uno de los tres estados en los que se opera con las memorias. Si la dirección no es cacheable (addr_no_cacheable) se pasará al **estado No_cacheable**. Si la petición es de escritura y hemos acertado se pasará al **estado Write_through**. Por último, si la señal es de lectura o escritura y hemos fallado se pasará al **estado Traer_bloque** activando la señal block_addr.

3. No_cache

La máquina se encuentra en este estado al operar con la memoria scratch. Activamos la señal Frame, que indica que la operación no ha terminado. Nos quedaremos en el estado si la memoria no ha completado la operación (not Bus_TRDY). Si la memoria completa la petición (Bus_TRDY) volveremos al estado Inicio y devolveremos el resultado si es una lectura (mux_output="01") o mandaremos los datos y incrementaremos la cuenta de escrituras (MC_send_data, inc_w) si es una escritura.

4. Traer_bloque

En este estado traemos bloques de memoria principal a la memoria caché. Llegamos al fallar una dirección ya sea en lectura o escritura. Activamos la señal Frame, que indica que la operación no ha terminado.

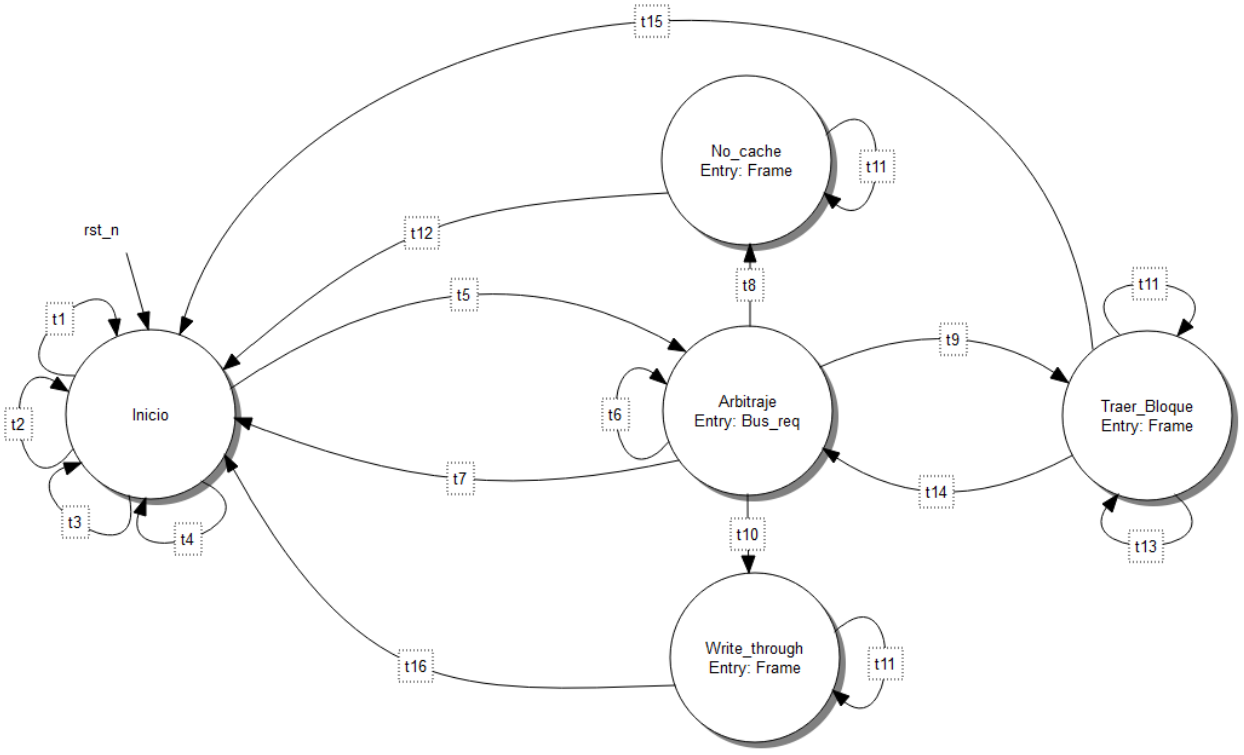
Nos **quedaremos en este estado** mientras la memoria no haya completado la operación (not Bus_TRDY) o la transferencia no haya terminado (not Last_word_block). Una vez terminada la operación (Last_word_block) pasaremos al **estado Inicio** si ha sido una lectura o al **estado Arbitraje** si ha sido una escritura.

Siempre que recibamos Bus_TRDY activaremos las señales para escribir en cache en la vía correspondiente y aumentar el contador (mux_origen='1', MC_WE0= not via_2_rpl, MC_WE1= via_2_rpl, count_enable). Cuando terminemos la transferencia escribiremos la etiqueta del bloque, indicaremos a la memoria que es la última palabra y aumentaremos la cuenta de fallos en memoria (MC_tags_WE, last_word, inc_m).

5. Write_through

En este estado escribimos en memoria cache el dato que nos da el MIPS. Activamos la señal Frame, que indica que la operación no ha terminado.

Mientras no se haya completado la operación (not Bus_TRDY) **nos quedaremos en el estado**. Una vez completada la operación (Bus_TRDY) volveremos al **estado Inicio** y activaremos las señales necesarias para escribir en cache (ready, MC_send_data, mux_origen='0', last_word, MC_WE0=hit0, MC_WE1=hit1, inc_w).



	Entrada	Salida
t1	not RE, not WE	ready
t2	(RE or WE) and UNALIGNED	ready, load_addr_error
t3	RE and internal_addr	ready, mux_output=”10”
t4	RE and HIT	ready, mux_output="00"
t5	WE or (RE and HIT)	Bus_req
t6	not Bus_grant	-
t7	Bus_grant, Bus_DevSel	MC_send_addr_ctrl, MC_bus_Rd_Wr=WE and (hit or addr_non_cacheable), load_addr_error, ready
t8	Bus_grant,Bus_DevSel,addr_no_cacheable	-
t9	Bus_grant, Bus_DevSel, (RE or (WE and HIT)	block_addr
t10	Bus_grant, Bus_DevSel, WE, HIT	-
t11	not Bus_TRDY	-
t12	Bus_TRDY	ready, mux_output=RE, MC_send_data=WE, inc_w=WE
t13	Bus_TRDY, not last_word_block	mux_origen, MC_WE0=not via_2_rpl, MC_WE1=via_2_rpl, count_enable
t14	Bus_TRDY,last_word_block,WE,HIT	mux_origen, MC_WE0=not via_2_rpl, MC_WE1=via_2_rpl, count_enable, MC_tags_WE, last_word, inc_m
t15	Bus_TRDY,last_word_block,RE,HIT	mux_origen, MC_WE0=not via_2_rpl, MC_WE1=via_2_rpl, count_enable, MC_tags_WE, last_word, inc_m
t16	Bus_TRDY	ready, MC_send_data, mux_origen=”0”, last_word, MC_WE0=hit0, MC_WE1=hit1, inc_w=WE

Diagrama de estados de error

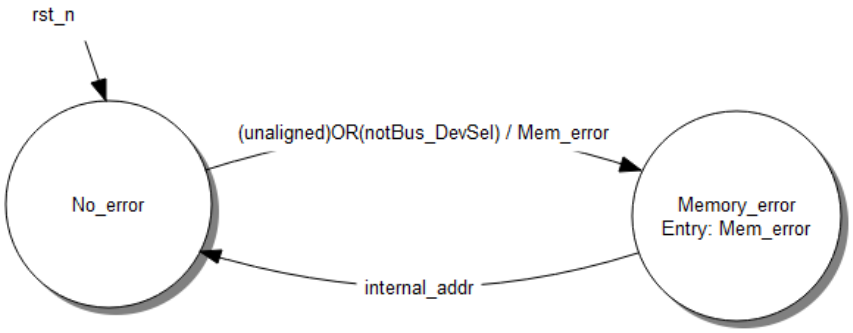
Para manejar los posibles errores de la jerarquía utilizamos un segundo autómata de dos estados que modifica la señal Mem_error que envía el controlador al procesador.

1. No_error

Es el estado por defecto de la máquina, en el cual la señal Mem_error vale 0. Seguiremos en este estado mientras no se accedan a direcciones no alineadas ni se acceda a direcciones no válidas (en las que ningún esclavo responda).

2. Error

Estado que activa la señal Mem_error y avisa al MIPS de un acceso incorrecto a la memoria. Llegaremos a este estado si la dirección que pedimos no está alineada (en nuestro MIPS, todas las direcciones deberán estar alineadas a la palabra), o si la dirección con la que queremos interactuar no pertenece a la MD ni a la MD Scratch. Lo sabremos ya que una vez concedido el bus mediante Bus_grant, no recibiremos la señal Bus_DevSel. Saldremos del estado de error cuando leamos el registro que guarda la dirección que ha causado el error (internal_addr).



Descomposición direcciones memoria

La memoria caché opera con un diseño asociativo de **2 vías**, lo que implica la presencia de 2 bloques por cada set. Además, cada vía consta de 4 bloques, y cada bloque almacena 4 palabras. Al considerar esta información, junto con las divisiones de bits presentes en el archivo de origen MC datos.vhd, la descomposición para el direccionamiento de la memoria caché es la siguiente:

Algoritmo y latencias de las transferencias

Especificación	Tiempo
if (internal_addr) { ret; }	1
else if (cacheable)	
look-up (@x)	1
if (RE) { MDScratch (rW, @x); **waitfor** MDScratch; }	CrW(MDS)
else { MDScratch (wW, @x); **waitfor** MDScratch }	CwW(MDS)
else	
look-up (@x);	1
if (RE)	
if (miss (@x)) { MD(rB, @x); **waitfor** MD; Mc+X }	CrB + 1
ret x;	0
if (WE)	C
if (miss (@x)) { MD(rB, @x); **waitfor** MD; }	CrB
MD (wW, @x); MC (wW, @x); ret;	CwW

- Leer bloque de MD, **CrB (MD)** = 6 + 3*2 = 12 ciclos
- Escribir palabra en MD, **CwW (MD) =** 6 ciclos
- Leer palabra de MD Scratch, **CrW(MD Scratch)** = 2 ciclos
- Escribir palabra en MD Scratch, **CwW(MD Scratch)** = 1 ciclo
- Leer palabra del registro interno, **CrW(Reg. Interno)** = 1 ciclo

Cálculo de los ciclos efectivos

En la expresión de cálculo de los ciclos efectivos que tarda nuestra jerarquía en responder a una petición hemos tenido en cuenta los diferentes escenarios que se pueden presentar. Los casos son los presentados en el algoritmo descrito anteriormente. Añadimos también 1,5 ciclos de arbitraje para algunos casos de escritura o lectura. La expresión obtenida es la siguiente:

$$Ceff = 1 + \frac{\sum rh}{\sum refs} + \frac{\sum r_{internal}}{\sum refs} + \frac{\sum rm \cdot (CrB_{MD} + 1{,}5)}{\sum refs} + \frac{\sum wh \cdot (CwW_{MD} + 1{,}5)}{\sum refs} + \frac{\sum wm \cdot (CrB_{MD} + CwW_{MD} + 3)}{\sum refs} + \frac{\sum r_{scratch} \cdot (CrW_{MDScratch} + 1{,}5)}{\sum refs} + \frac{\sum w_{scratch} \cdot (CwW_{MDScratch} + 1{,}5)}{\sum refs}$$

Programas de prueba

De cara a comprobar que el diseño funciona correctamente, se han ejecutado diversos **programas de prueba** que prueban todos los aspectos de este nuevo sistema de memoria, así como los efectos de su implementación en el MIPS. Inicialmente se comprobó el correcto funcionamiento del diseño con los test suministrados por los docentes de la asignatura a través de diversos recursos de Moodle, y posteriormente se diseñaron y comprobaron nuestros propios tests.

Tras terminar el diseño del sistema de memoria se ejecutaron los 15 test ubicados en el fichero *testbench_MD_mas_MC_alumnos.vhdl*, mediante los cuales se podía examinar el funcionamiento del nuevo sistema de memoria diseñado. Estos resultaron bastante útiles para cerciorarnos de que dicho funcionaba correctamente **antes de implementarlo en el MIPS**, ya que probaba prácticamente todas las facetas de este.

A continuación, se implementó el sistema de memoria en el MIPS y una vez pasados los tests *Codigo_retardado* y *Codigo_test_IRQ*, así como los tests diseñados por nosotros en el anterior proyecto, se procedió al diseño de **nuevos test**. Se ha decidido diseñar un par de tests, uno que pruebe **todos los aspectos del sistema de memoria** y otro más breve con una aplicación real como la suma de dos vectores.

Estos dos programas de prueba diseñados por nosotros reciben los nombres de *testMC* y *sumaVectores*.

1. TestMC

Este test comprueba el correcto funcionamiento de varios aspectos del sistema de memoria (fallos en lectura, fallos en escritura, aciertos en lectura, aciertos en escritura, el uso de los 4 conjuntos, el uso de ambas vías, el reemplazo de sets, la lectura y la escritura de la memoria scratch, el acceso a direcciones erróneas, etc…). Todo esto combinado con anticipación de operandos y demás utilidades implementadas en el primer proyecto de la asignatura.

A continuación, se describe el contenido inicial de la memoria de datos, antes de ejecutar el test, las direcciones que no aparecen tienen un valor inicial de 0x0:

Dirección	Contenido
@0	0x0
@4	0x4
@8	0x8
@12	0xA
@16	0x0
@20	0x100000004
@24	0x10000
@28	0x80
@32	0x40
@36	0x1000000
@48	0xC0

Código correspondiente al primer programa de pruebas:

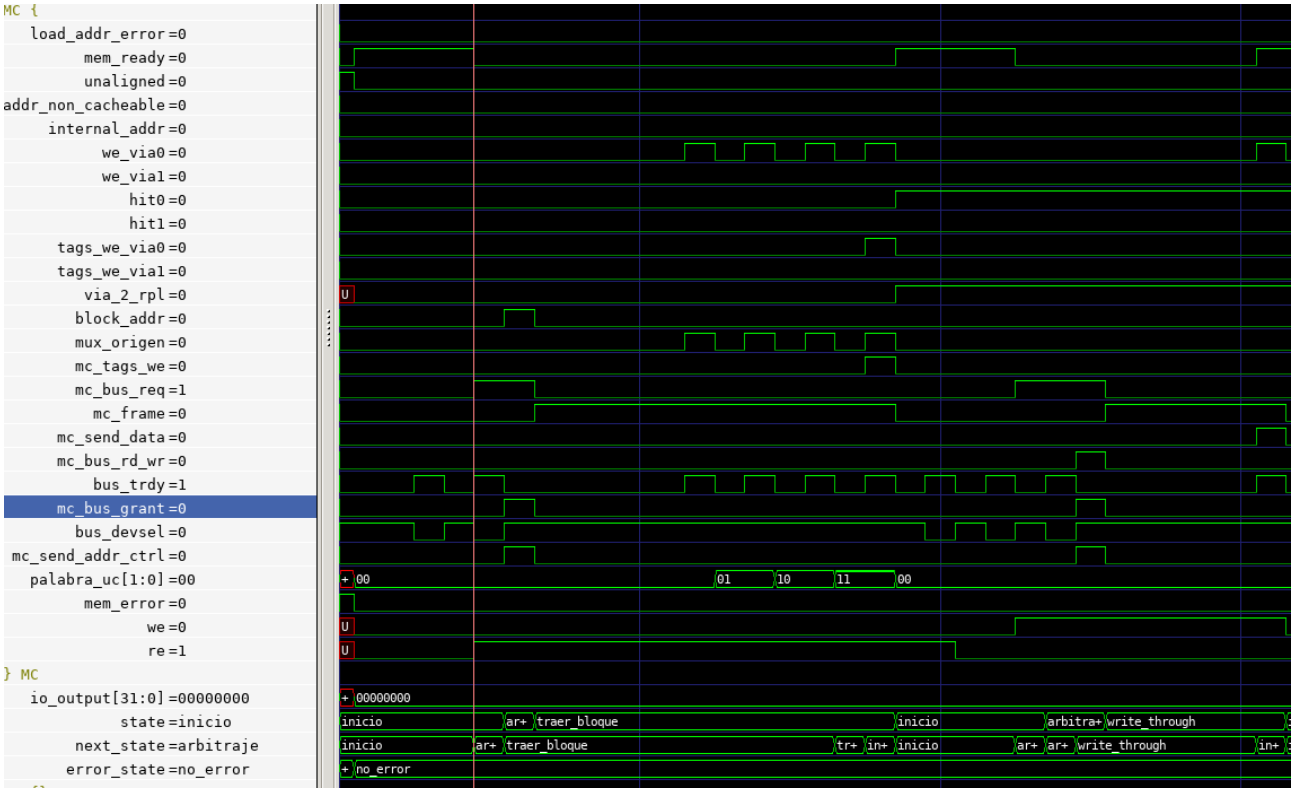
@	Instrucción	Evento	Tag	Set	Word	Way	Comentarios
@0x0	LW R1, 4(R0)	Read miss	0	00	01	0	Trae @0, @4, @8 y @12 de MD a vía 0 set 0; R1 <= 4:
@0x4	LW R2, 8(R0)	Read hit	0	00	10	0	R2 <= 8;
@0x8	ADD R3, R1, R2	—	—	—	—	—	R3 <= 4+8;
@0xC	SW R3, 8(R1)	Write hit	0	00	11	0	Escribe un 12 en via 0 set 0 palabra 3; MD(3) <= 12;
@0x10	SW R3, 12(R1)	Write miss	0	01	00	0	Trae @16, @20, @24 y @28 a MC via 0 set 1; MD(4) <= 12;
@0x14	LW R10, 20(R0)	Read hit	0	01	01	0	R10 <= 0x10000004;
@0x18	LW R9, 24(R0)	Read hit	0	01	10	0	R9 <= 0x10000;
@0x1C	LW R11, 0(R9)	—	—	—	—	—	Acceso a memoria inválido, se guardará en ADDR_Error_Reg “0x10000”, dirección que ha causado el error
@0x20	LW R11, 36(R0)	Read miss	0	10	00	0	Trae @32, @36, @40 y @44

							de MD a vía 0 set 2; R11 <= 0x1000000;
@0x24	SW R1, 0(R10)	—	—	—	—	—	MDScratch(1) <= 4;
@0x28	LW R2, 0(R10)	—	—	—	—	—	Lee de memoria scratch; R2 <= 4;
@0x2C	LW R4, 32(R0)	Read hit	0	10	00	0	R4 <= 0x40;
@0x30	SW R1 0(R4)	Write miss	1	00	00	1	Se trae el bloque correspondiente al set 0 via 1 MD(16) <= 4;
@0x34	LW R2, 48(R0)	Read miss	0	11	00	0	Se trae el bloque correspondiente al set 3 via 0 R2 <= 0xC0;
@0x38	LW R9, 28(R0)	Read hit	0	01	11	0	R9 <= 0x80
@0x3C	SW R3, 0(R9)	Write miss	2	00	00	0	Reemplazo de vía 0 set 0; Mem(32) <= C;
@0x40 (FIN)	BEQ R1, R1,FIN	—	—	—	—	—	Bucle infinito de terminación

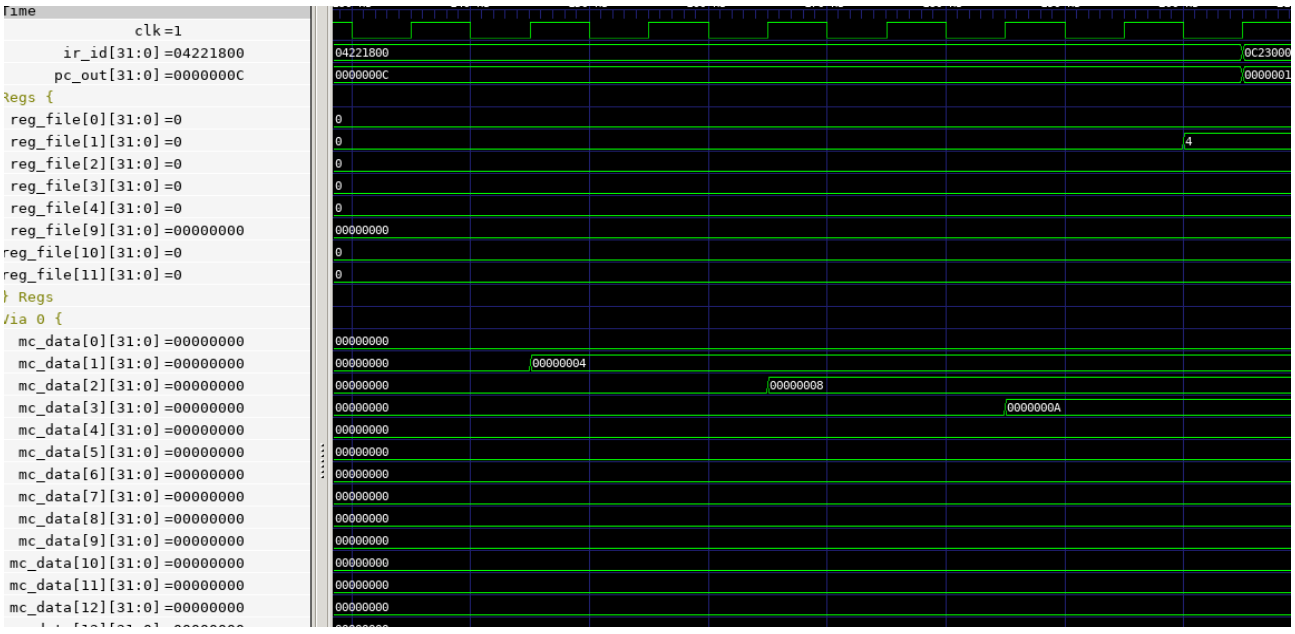
Depuración con GTKWAVE

A continuación, se procederá a mostrar capturas de pantalla del visualizador de formas de onda **GTKWAVE** para el programa anterior. En las capturas se mostrarán a modo de resumen algunos de los casos significativos, para mostrar de una forma más ilustrativa qué señales se activan en cada caso y el funcionamiento del autómata descrito anteriormente.

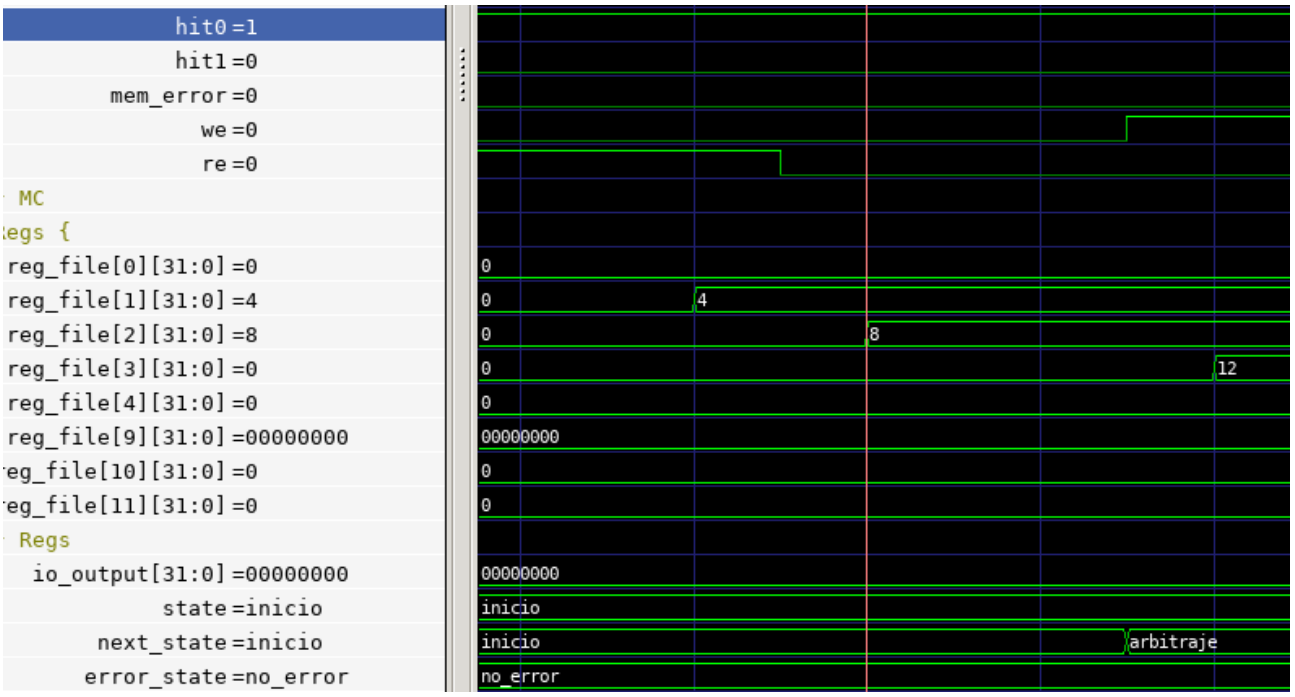
La primera instrucción de carga de datos en un registro siempre va a ser obviamente un read miss, que en WT + AF supone que MP traiga el bloque entero a MC, para que este pueda servir al procesador la palabra fallada.



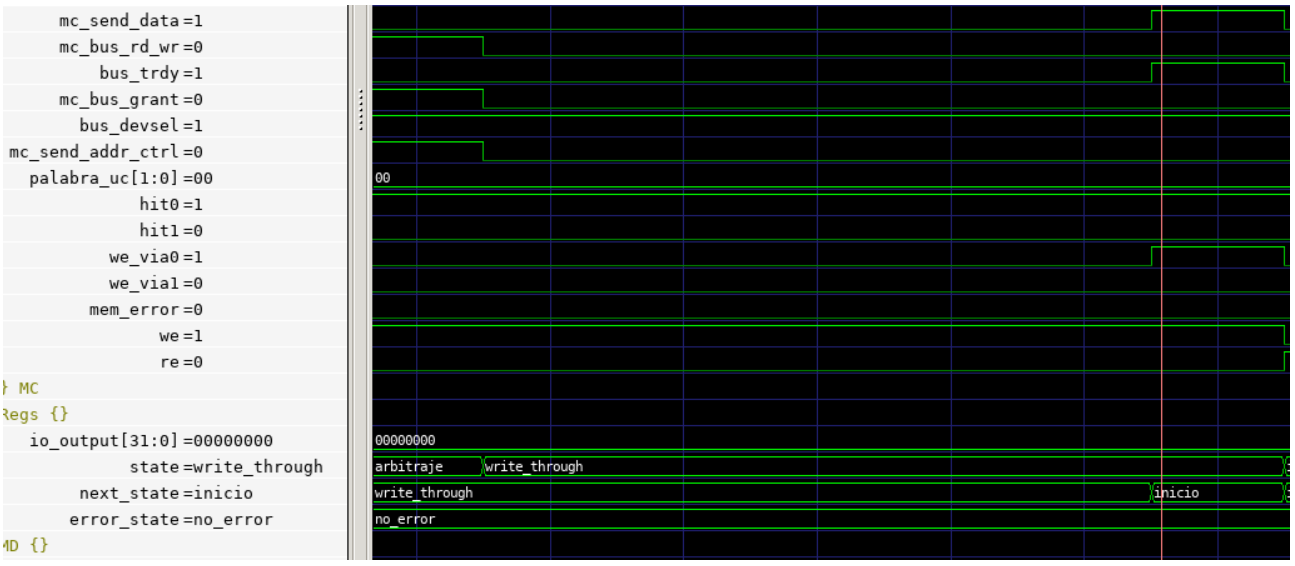
Como se aprecia en la imagen, al iniciar la etapa de memoria de esta instrucción, se activa la señal RE y no se activa ni hit0 ni hit1 (ya que ha sido un fallo), por lo que activamos bus_req y vamos al estado de arbitraje, en el que rápidamente nos conceden el bus y la MD responde a nuestro mc_send_addr_ctrl (activado nada más tener el bus_grant) activando bus_devsel. A partir de ahí, comienza la copia del bloque, como se ve en la imagen, palabra_uc va aumentando de uno en uno conforme se van trayendo palabras a MC. Finalmente, se vuelve al estado de arbitraje y se detecta hit (ya que tenemos el bloque recién traído a MC), por lo que pasamos al estado write_through y se sirve la palabra al MIPS (reg1). En la siguiente imagen se aprecia come se sirve el 4 al MIPS y se actualiza el set 0 de la vía 0 de la cache:



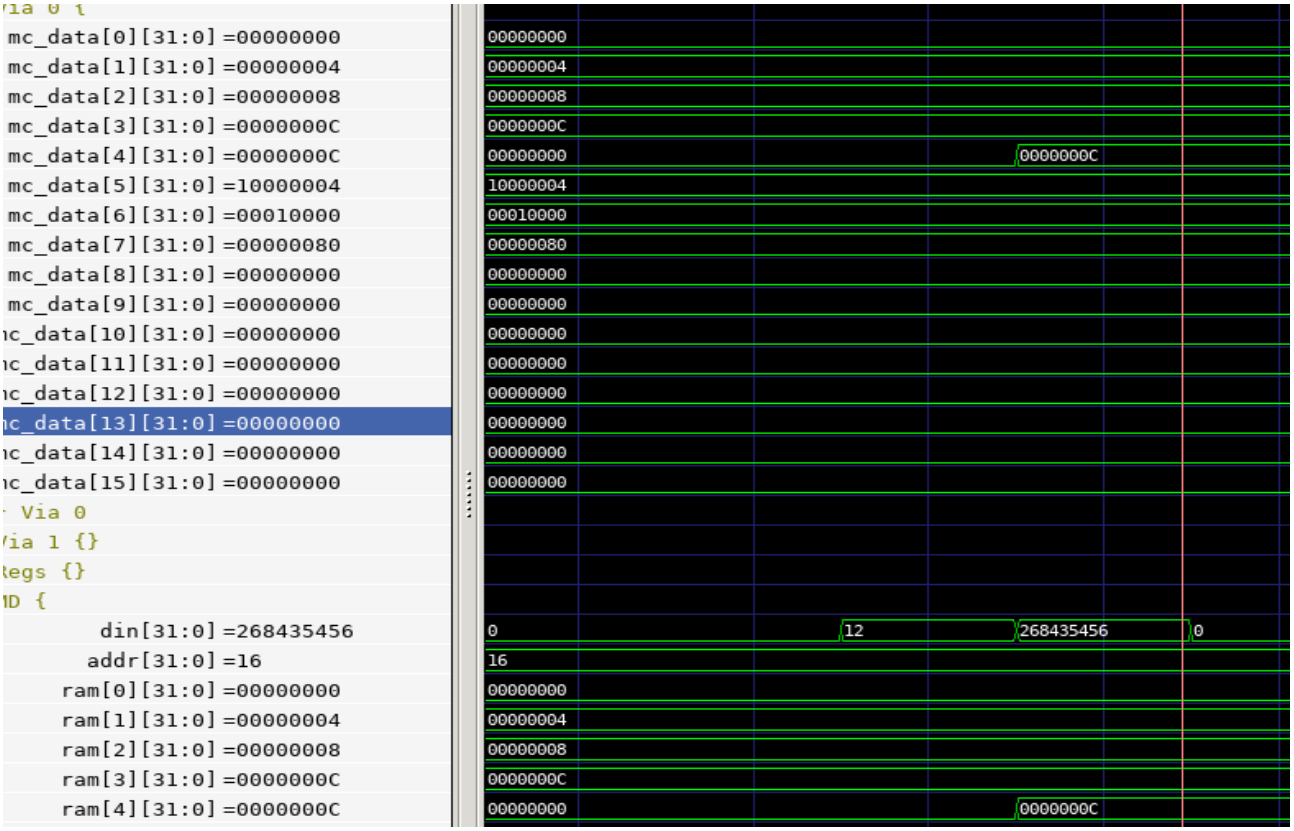
La siguiente instrucción es un read hit, por lo que simplemente se activará la señal hit0 y se servirá la palabra solicitada al procesador (un 8 al registro 2). Esta irá seguida de un simple add que sumará en el registro 3 los dos datos que acaban de ser almacenados en los registros. Todo esto se puede apreciar en la siguiente imágen:



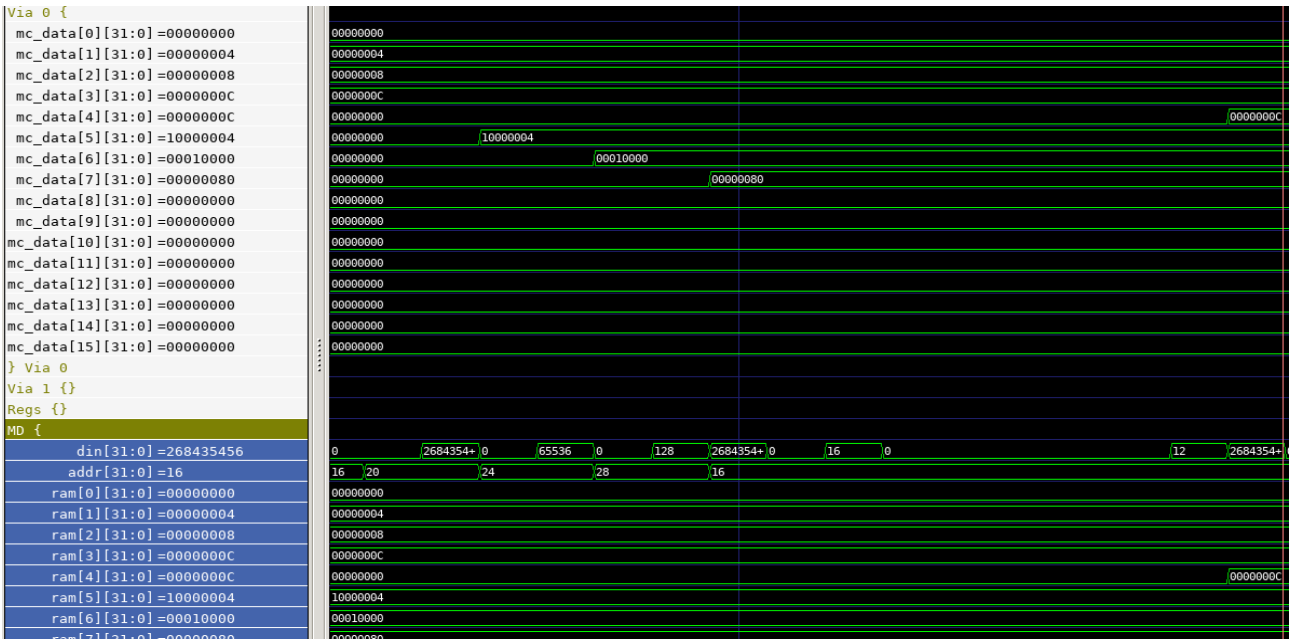
A continuación viene un write hit en el que se escribirá un 12 en la palabra 3 del único set que tenemos en MC. También se escribirá en memoria un 12 en la dirección 3. Como se puede observar en la siguiente imagen, se pasará del estado de arbitraje al de write_through directamente, sin pasar por traer_bloque, puesto que ha sido un acierto en escritura. También se activarán las señales correspondientes (WE, hit0, mc_send_addr_ctrl…) Una vez en write_through, en cuanto bus_TRDY valga 1, se activará mc_send_data, inc_w y we_via0:



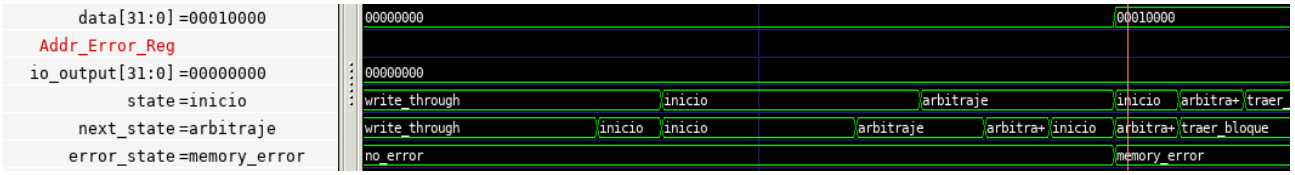
También escribe tanto en MD como en MC:



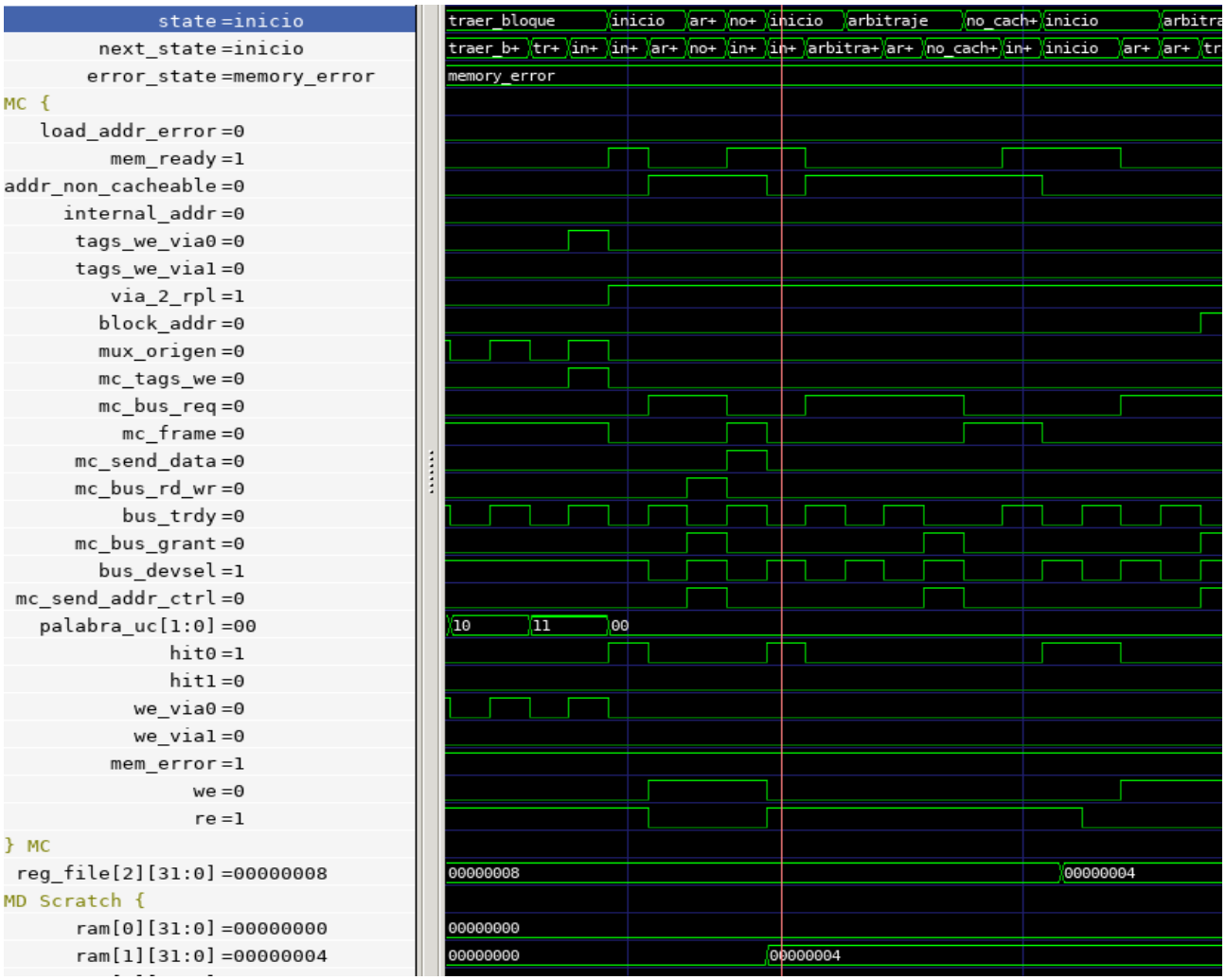
A continuación tenemos el primer write miss. Lo que ocurrirá será similar a lo visto anteriormente. Primero traeremos el bloque fallado de MD a MC, para posteriormente hacer hit y a partir de ahí ya actuaría como un write hit normal. El resultado de esto en las memorias, es que traemos 0x0, 0x100000004, 0x10000 y 0x80 al segundo set de la primera vía de la cache, para a continuación reescribir la primera palabra de este set con un 0xC. También se escribe 0xC en MD(4).



Posteriormente se realiza un acceso a una dirección inválida (0x1000), por lo que se almacena en Addr_Error_Reg la dirección que ha causado este fallo. También se pasa al estado de memory_error en el autómata de error del cual no se saldrá hasta que sea leído el contenido del registro de error. En ese caso volverá al estado de no error.

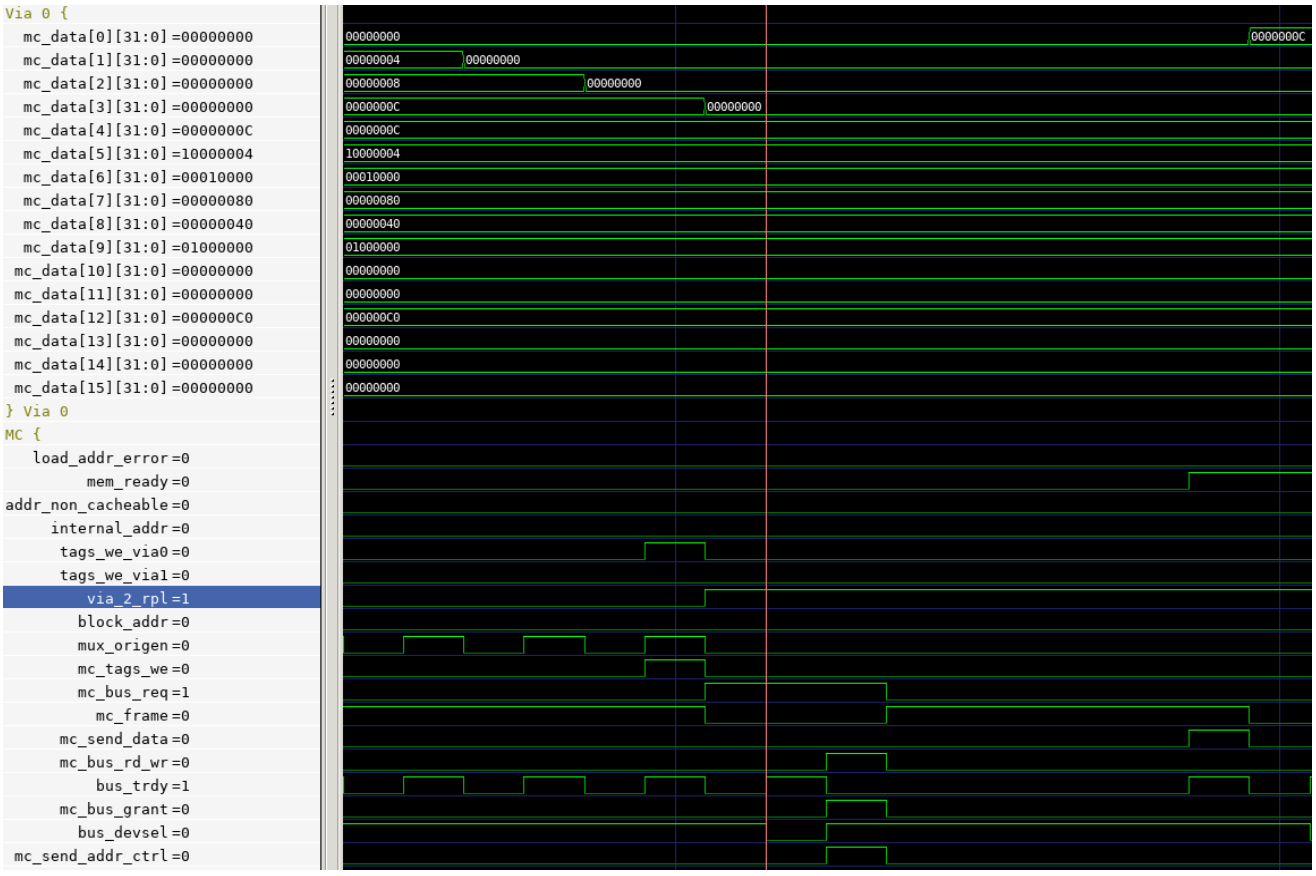


En este test también se prueba el correcto funcionamiento de la memoria scratch, primeramente escribiremos en Scratch Mem(1), ya que si escribimos en el la dirección anterior, ese dato rápidamente se sobreescribirá debido a que IO Master escribe continuamente el valor leído de IO_input en la primera dirección de la memoria Scratch. Una vez escrito el dato, el MIPS solicitará mediante una instrucción load que se cargue ese mismo dato en el registro 2, como vemos en la siguiente imagen:



Como se observa en la imagen, del estado de arbitraje se pasa al estado no cacheable y a continuación se escribe el dato deseado en la memoria scratch. Acto seguido, el MIPS solicita leer ese dato de scratch y cargarlo en el registro 2, solicitud que es completada con éxito. Como se observa en la imagen, la señal de addr_non_cacheable se pone a 1.

Por último, comprobamos también que el reemplazo de vías funciona correctamente, ya que hemos escrito en el conjunto 0 de ambas vías, por lo que ahora probamos a escribir otra vez en dicho conjunto pero con un tag distinto. Como se observa en la siguiente imagen, el conjunto que se reemplaza es el de la vía 0, ya que via_2_rpl toma el valor de 0 en el momento del reemplazo. El conjunto se reemplaza correctamente (se escriben ceros porque se reemplaza con direcciones de memoria que no almacenan ningún dato) y posteriormente se escribe la palabra deseada tanto en la palabra correspondiente de MC como en M, dado que el evento que ha generado este fallo ha sido una escritura.



2. SumaVectores

Este test realiza la suma de dos vectores sobre otro vector utilizando la memoria principal para almacenar los vectores a sumar y la memoria scratch para almacenar los resultados. Es un ejemplo más práctico y cercano a los posibles usos que daríamos al MIPS en un entorno real.

Pseudocódigo:

```
int a[4] = {1,2,3,4}; int b[4]={1,2,3,4}; int sum[4]
for i..4:
   sum[i] = a[i]+b[i]
end
```

Contenidos en memoria:

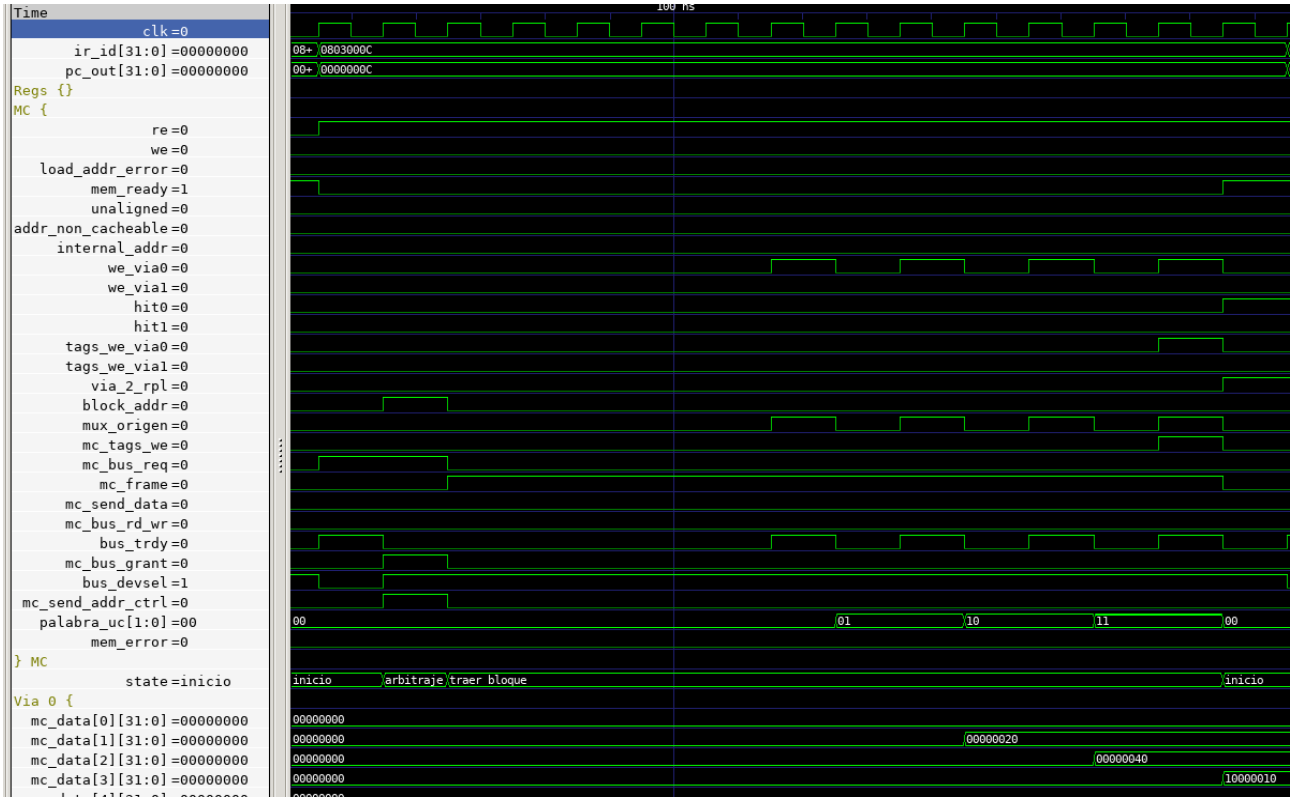
A continuación se describe el estado inicial de la memoria de datos y las instrucciones en la memoria de instrucciones.

Dirección	Contenido
@0	0x0
@4	0x20
@8	0x40
@12	0x10000010
@16	0x4
@20	0x1
@24	0x4
@32	0x1
@36	0x2
@40	0x3
@44	0x4
@64	0x1
@68	0x2
@72	0x3
@76	0x4

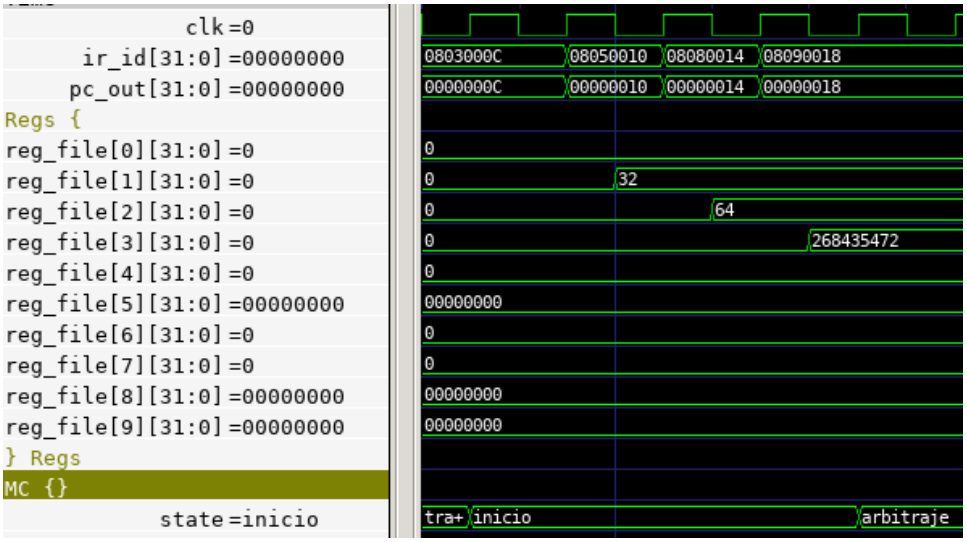
@	Instrucción	Evento	Tag	Set	Word	Way	Comentarios
@0x0	lw r1, 4(r0)	Read miss	0	00	01	0	R1 <= @a Principio de a[ ]. Miss, lleva @0..12 de MD a MC
@0x4	lw r2, 8(r0)	Read hit	0	00	10	0	R2 <= @b Principio de b[ ]. Hit, coge dato de MC
@0x8	lw r3, c(r0)	Read hit	0	00	11	0	R3 <= @sum Principio de sum[ ]. Hit, coge dato de MC
@0xC	lw r5, 10(r0)	Read miss	0	01	00	0	R5 <= 4 (n) Número de componentes de los vectores. Miss, lleva @C..18 de MD a MC
@0x10	lw r8, 14(r0)	Read hit	0	01	01	0	R8 <= 1 Auxiliar para sumar 1 a la iteración. Hit, coge dato de MC
@0x14	lw r9, 18(r0)	Read hit	0	01	10	0	R9 <= 4 Auxiliar para sumar 4 las direcciones. Hit, coge dato de MC
@0x18 (BU)	beq r0, r5, FIN	—	—	—	—	—	while r0(i) < r5 (n) Empezamos bucle
@0x1C	lw r6, 0(r1)	Read miss	0	10	00	0	R6 <= a[i]
@0x20	lw r7, 0(r2)	Read miss	1	00	00	1	R7 <= b[i]
@0x24	add r4, r6, r7	—	—	—	—	—	R4 <= R6+R7 Suma de a[i] y b[i]
@0x28	sw r4, 0(r3)	—	—	—	—	—	sum[i] <= R4 Guardamos resultado en memoria scratch
@0x2C	add r1, r9, r1	—	—	—	—	—	@a = @a++ (+4)
@0x30	add r2, r9, r2	—	—	—	—	—	@b = @b++ (+4)
@0x34	add r3, r9, r3	—	—	—	—	—	@sum = @sum++ (+4)
@0x38	add r0, r8, r0	—	—	—	—	—	i++
@0x3C	beq r0, r0, BU	—	—	—	—	—	Volvemos al principio del bucle
@0x40 (FIN)	beq r1, r1,FIN	—	—	—	—	—	Bucle infinito de fin

Depuración con GTKWAVE

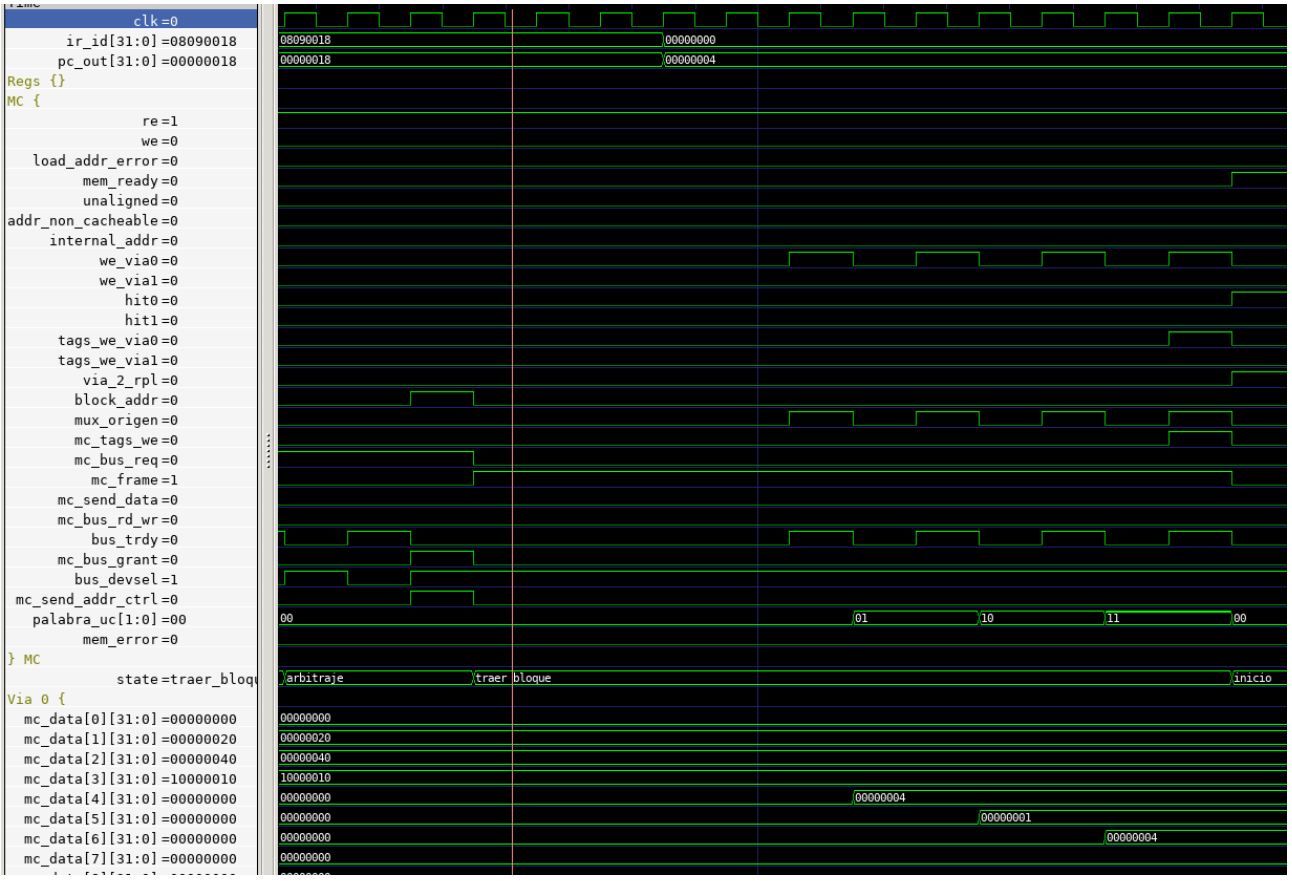
A continuación, mostramos los resultados de la simulación obtenida para el programa descrito anteriormente.



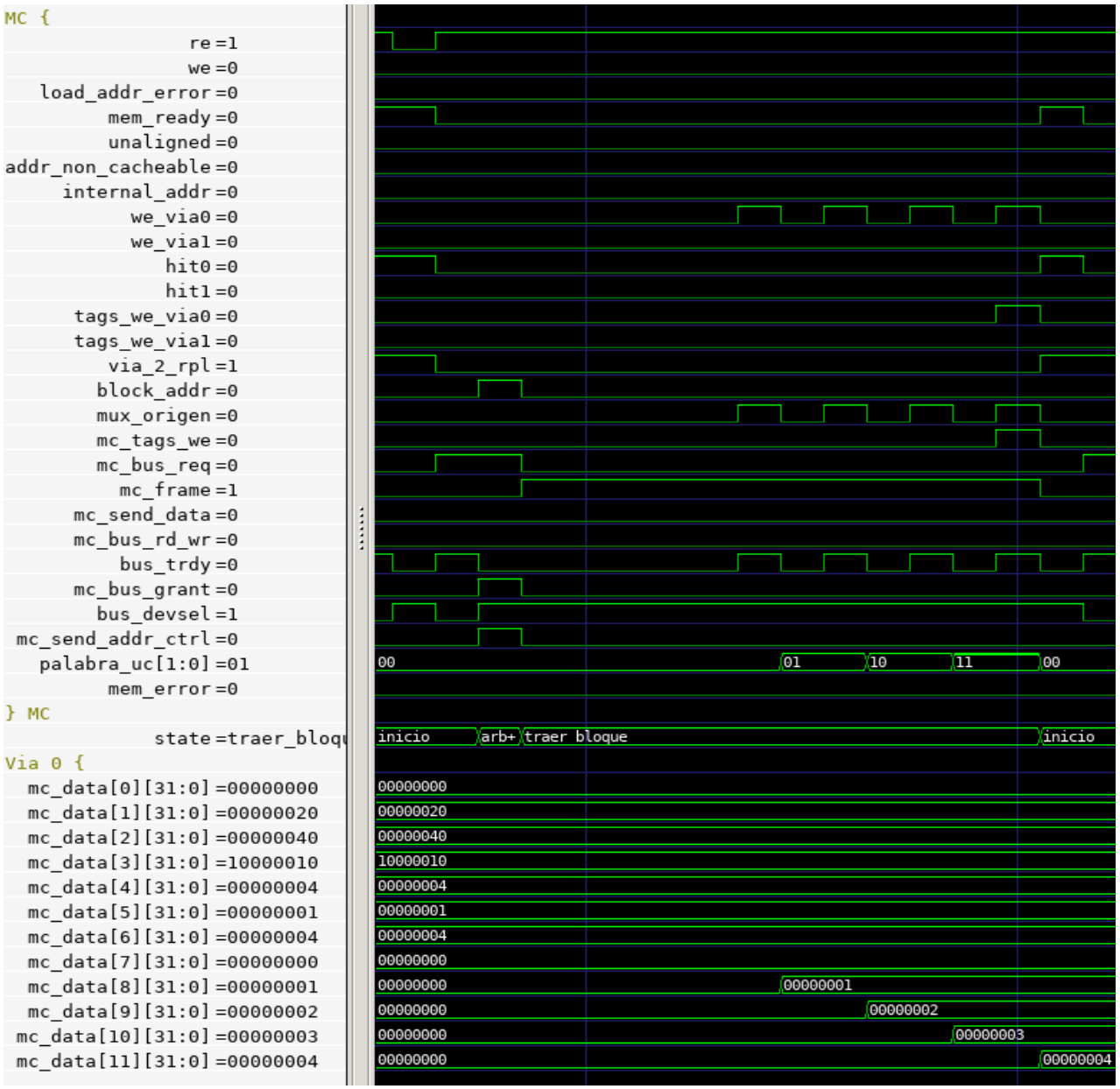
En esta primera imagen observamos la ejecución de las tres primeras instrucciones del programa. El primer load será fallo, por lo que traeremos el primer bloque de memoria a cache. Cuando recibimos la señal RE y no es hit, bajamos ready y activamos mc_bus_req para pedir el bus. Una vez nos han dado el bus activamos block_addr, que pasará la primera direccion del bloque a memoria, y pasamos al estado Traer_bloque. Una vez en este, esperamos al dato y cuando recibimos bus_TRDY activaremos la señal de escritura en la vía a reemplazar. Al ser la primera escritura lo haremos en la vía 0. Una vez hemos escrito las 4 palabras volvemos a Inicio y al ser hit activaremos la señal ready.



Una vez recibida la señal ready el MIPS escribirá el dato en el registro correspondiente, r1 en este caso. Dado que hemos traído las dos siguientes palabras a cache, las dos siguientes instrucciones son hit y escriben en 1 ciclo el dato pedido en el registro indicado por cada instrucción (r2 y r3).

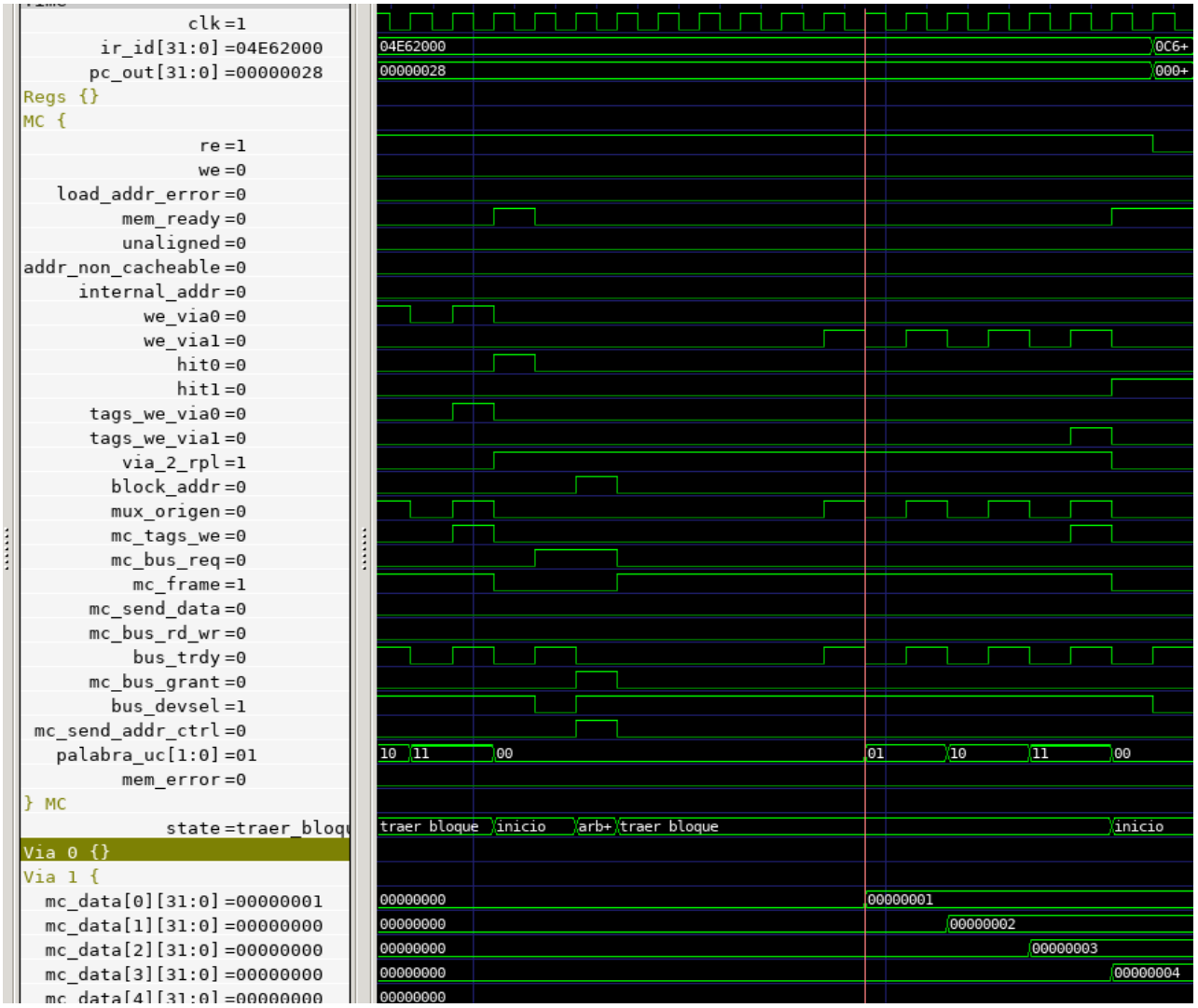


En este caso, repetimos el proceso anterior para los tres loads que siguen a los anteriores. Estos cargan en los registros tres datos que encontramos en el siguiente bloque de memoria. Por tanto, la primera lectura fallará y traeremos el bloque a memoria cache. Dado que el conjunto de estos datos es distinto de los anteriores escribimos en la misma vía de cache. Si el conjunto de ambos bloques hubiera sido el mismo, pondríamos los datos en la vía 1.

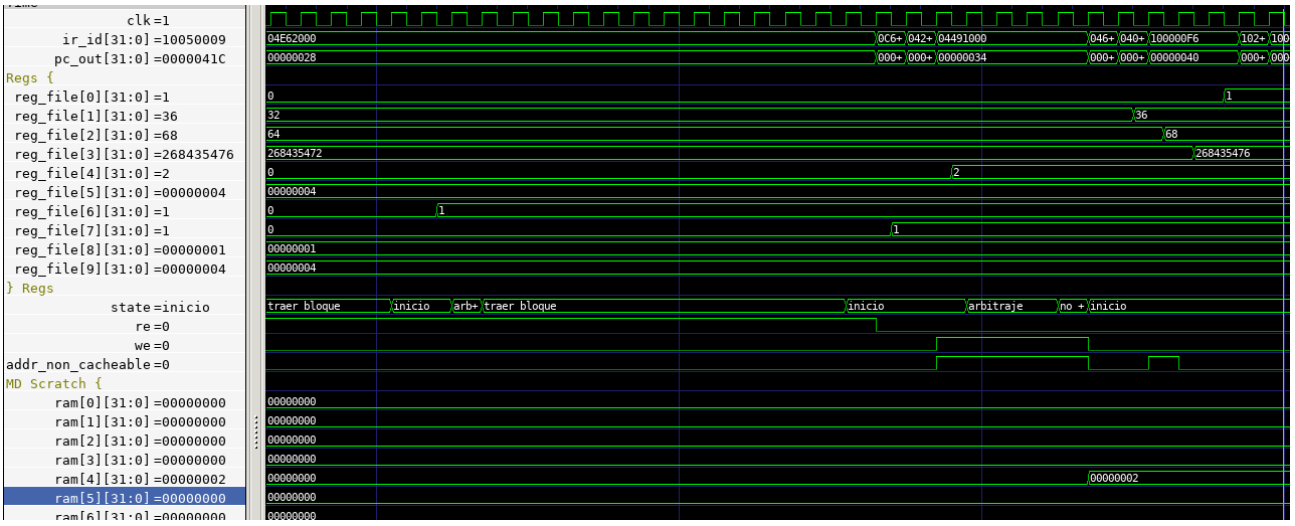


Ahora estamos en el bucle, por lo que al leer el primer dato del vector a fallaremos y traeremos el bloque de memoria en el que se encuentra. En este caso, hemos optado por optimizar el código situando las 4 componentes que vamos a sumar en el mismo bloque. Esta es una gran ventaja de esta jerarquía de memoria, ya que las próximas tres iteraciones accederemos a los datos con un coste mínimo (1 ciclo).

Además, observamos que una vez hemos leído el primer dato la MC baja la señal mem_ready. Esto indica que está buscando el siguiente bloque.

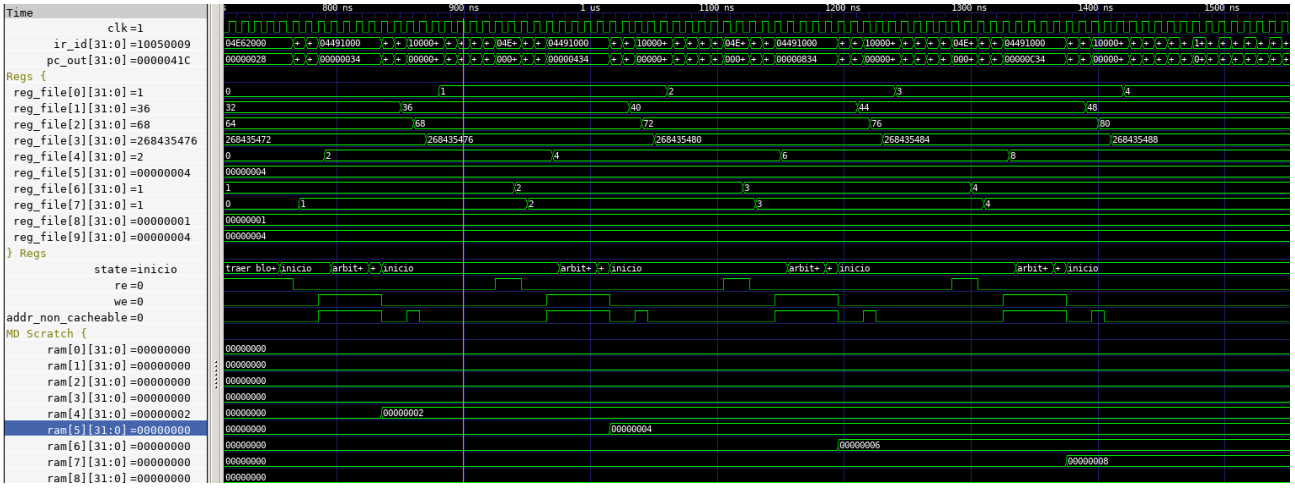


Podemos ver que en esta ocasión no hemos escrito el bloque en la vía 0, sino en la 1. Esto es así ya que el conjunto de las direcciones en las que se encuentran los datos es el 0 (@0x40). Por tanto, via_2_rpl es igual a 1 y escribiremos en esta vía.



Una vez hemos cargado los datos en los registros correspondientes (r6 para a[ ], r7 para b[ ]) sumamos los contenidos de estos en el registro 4. En el momento siguiente a la escritura observamos cómo se activa la señal WE. Al mismo tiempo se activa la señal addr_non_cacheable, que nos indica que la dirección donde queremos escribir corresponde a MD Scratch. Vemos el resultado de la suma (1+1=2) escrito en scratch 3 ciclos después.

En los siguientes ciclos sumamos 4 a los valores que tomamos para acceder a los vectores y 1 en el nº de iteraciones completadas.



Este proceso se repite en las siguientes tres iteraciones. vemos la escritura de los resultados en MD Scratch y como la cuenta de iteraciones aumenta hasta 4. En este momento vamos al bucle de fin y finaliza la ejecución.

Ejemplo de código que muestra mejora

La jerarquía diseñada en este segundo proyecto aporta mejoras de eficiencia principalmente en aquellos casos en los que el número de aciertos lectura sea elevado, ya que esta operación se resuelve en un solo ciclo.

Para analizar el speedup del nuevo sistema de memoria frente a la una memoria de datos básica sin memoria cache (como la proporcionada en el recurso de Moodle), hemos diseñado un fragmento de código para comprobar su eficiencia en el MIPS con ambos sistemas de memoria.

El programa de prueba para este apartado será el programa de prueba *SumaVectores*. Antes de ejecutar el bucle hacemos 6 lecturas a memoria principal, y en el bucle hacemos 2 lecturas a memoria principal y 1 escritura en memoria scratch por iteración. En las 6 primeras lecturas fallamos 2 veces y acertamos 4. Realizamos cuatro iteraciones sobre nuestro bucle y por la localidad espacial de los datos utilizados fallamos 2 lecturas (una vez por cada vector) y acertamos las 6 lecturas restantes. Cada vector ocupa un bloque que copiamos en cache en la primera iteración. En el bucle hacemos también 4 escrituras en memoria scratch.

Con la jerarquía de memoria implementada en este proyecto, que incluye una memoria scratch y una memoria caché de 4 conjuntos y 2 vías obtenemos 5,1 ciclos por acceso a memoria. Sin contar con esta jerarquía nos costaría 8,5 ciclos acceder a memoria.

$$C = 1 + \frac{10}{18} + \frac{4 \cdot (12 + 1{,}5)}{18} + \frac{4 \cdot (1 + 1{,}5)}{18} = 5,\hat{1}$$

$$C' = 1 + 1{,}5 + 6 = 8{,}5$$

Con estos datos calculamos el speedup:

$$speedup = \frac{Tex'}{Tex} = \frac{I' \cdot CPI' \cdot Tc'}{I \cdot CPI \cdot Tc} = \frac{I' \cdot \frac{C'}{I'} \cdot Tc'}{I \cdot \frac{C}{I} \cdot Tc} = \frac{46 \cdot \frac{181}{46} \cdot 10(ns)}{46 \cdot \frac{119{,}8}{46} \cdot 10(ns)} = 1{,}5108$$

Horas dedicadas

Apartado	**Horas dedicadas**		**Total**
	Guillermo Bajo Laborda	Álvaro de Francisco Nievas	
Estudio del enunciado y del código fuente	4 horas	4 horas	8 horas
Diseño del autómata y código del mismo en VHDL	4 horas	4 horas	8 horas
Implementación del sistema de memoria en el MIPS	45 minutos	15 minutos	1 hora
Pruebas, depuración y cambios	12 horas	10 horas	22 horas
Memoria	4 horas	5 horas 30 minutos	9 horas 30 min
Total	24 horas y 45 minutos	23 horas 45 minutos	48 horas y 30 min

Autoevaluación

Guillermo Bajo Laborda

• ¿Crees que has cumplido los objetivos de la asignatura?

Desde el punto de vista didáctico, considero que he cumplido con los objetivos de la asignatura de arquitectura y organización de computadores II. Esto se debe a que entre clases teóricas, problemas, prácticas y proyectos, a mi parecer he consolidado y comprendo la materia de la asignatura, tanto teóricamente como llevada a la práctica. En parte considero que esto ha sido posible gracias a tanta práctica, ya que por mucho que se comprendan los contenidos teóricos, si estos no son llevados a la práctica es complicado aprender de verdad. Honestamente, los temas 2 y 3 me resultaron inicialmente bastante más complicados que el primero, incluso después de haber asistido a clases teóricas y visto los vídeos de los temas. Sin embargo, gracias a la práctica 2 y a este proyecto los he logrado comprender y me han resultado inclusive interesantes.

• ¿Qué nota te pondrías si te tuvieras que calificar a ti mismo?

En este proyecto me calificaría a mí mismo con una nota en torno al 8, puesto que creemos que el sistema diseñado funciona correctamente. Es cierto que no se han realizado los apartados opcionales, principalmente por falta de tiempo más que por pereza, pero considero que en general, hemos hecho un proyecto que cumple con las expectativas.

Álvaro de Francisco Nievas

• ¿Crees que has cumplido los objetivos de la asignatura?

Sí. Creo que entiendo todos los aspectos del proyecto y las prácticas realizadas durante este cuatrimestre. Además, con las clases de problemas he asentado los conocimientos más teórico-prácticos de la asignatura. Tras el primer proyecto, me perdí un poco en la parte de la jerarquía de memoria. Lo noté especialmente en la realización de los problemas de esas semanas. No obstante, la práctica de la cache en logisim y este proyecto me han ayudado a asentar los conocimientos de esa parte del temario. Además, los videos de la asignatura me ayudaron cuando con las diapositivas no lo entendía todo. Son un recurso excelente. La parte de buses me ha resultado especialmente entretenida. La coordinación de los distintos elementos de un procesador y su comunicación me ha parecido interesante.

• ¿Qué nota te pondrías si te tuvieras que calificar a ti mismo?

El proyecto elaborado funciona correctamente para todos los escenarios que se nos han ocurrido y considero que hemos hecho un buen trabajo en todas las tareas propuestas. Es cierto que no hemos realizado los apartados voluntarios, pero considero que la carga de trabajo de este último mes nos obliga a priorizar las partes obligatorias de los trabajos y proyectos propuestos en el grado. Me calificaría entorno a un 8-8,5.