

Proyecto 2 – 2025 – Jerarquía de memorias de datos

Langa Ibáñez, Ana - 839811

Pastor Peralta, Fernando - 897113

Junes, 12 de mayo de 2025

Resumen:

Este proyecto consiste en el diseño de la unidad de control de una memoria caché que atienda las peticiones del procesador y gestione en consecuencia las transferencias de datos a través de un bus semi-síncrono. Esta jerarquía de memoria incluye una memoria caché (MC), una memoria de datos (MD) más lenta que la empleada en el proyecto 1, y una memoria de datos adicional denominada MD Scratch. Esta última es considerablemente más rápida, pero posee un rango de direcciones limitado y su contenido no debe almacenarse en la memoria caché (MC). El procesador utilizado será el mismo que en el proyecto anterior. El sistema de memoria, además, dispone de dos maestros en el bus (el controlador de la MC y el IOMaster). Al existir dos maestros, el bus requiere de un árbitro para evitar conflictos. Este árbitro utiliza una asignación dinámica de prioridades.

Para la realización de este proyecto se ha hecho uso de los módulos SoC del primer proyecto, excepto el sistema IO_MD.

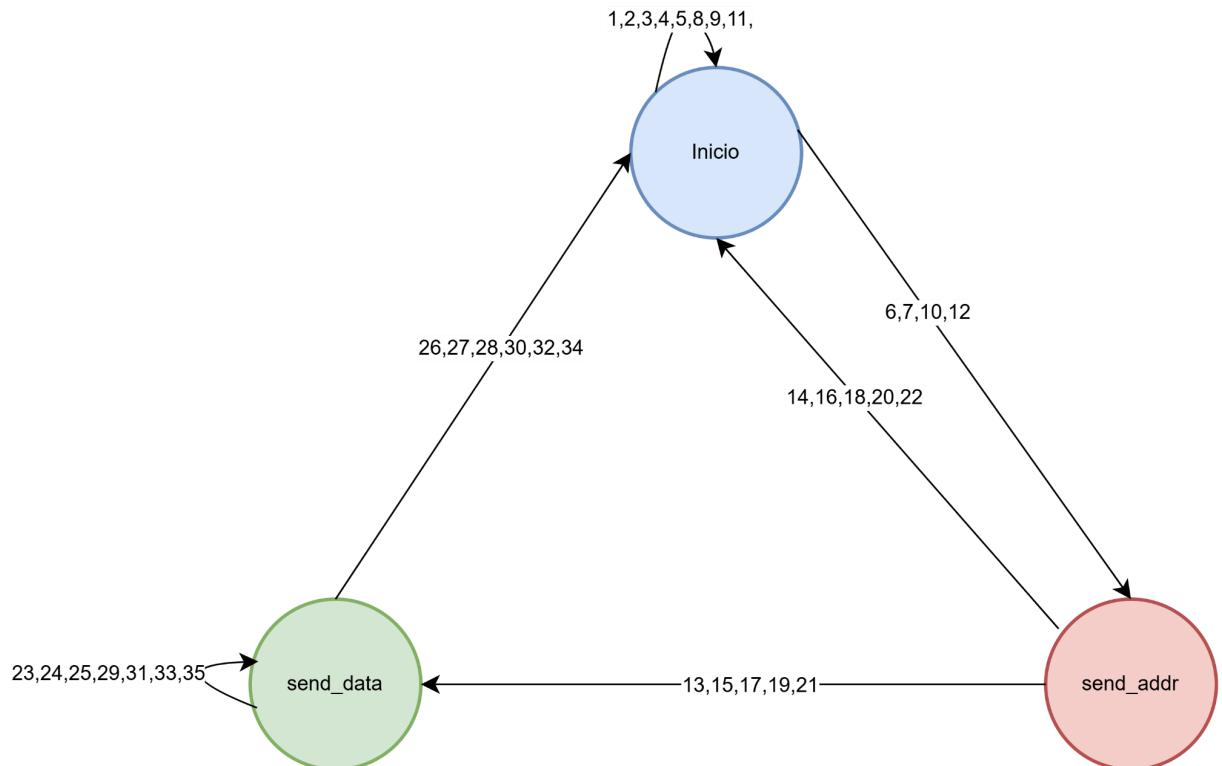
La MC es una caché asociativa de dos vías con 128 bytes, sustitución FIFO, política de actualización escritura a escritura (write-through) y política convencional de asignación y recuperación en caso de escritura fallida. Cada vía contiene 4 bloques de 4 palabras.

La MC incluye cuatro contadores:

- cont_m: Fallos en MC
- cont_w: Operaciones de escritura en MC, (ignorando las escrituras en registros de E/S)
- cont_r: Operaciones de lectura desde MC, (ignorando las lecturas desde registros E/S o MD Scratch).
- cont_inv: Contar bloques almacenados en MC invalidados debido a una lwi.

Para el diseño de la unidad de control se ha seguido un proceso para abarcar todas los requisitos de este proyecto y comprobar su correcto funcionamiento. Tras leer detalladamente el guión del proyecto y tener claro qué es lo que se pedía hemos analizado detalladamente el resto de componentes proporcionados por el profesorado para tener un amplio conocimiento de la conexión de las señales. Tras tener claro el funcionamiento, definimos los estados principales y sus respectivas transiciones. Para finalizar el proceso hicimos un debugging tanto con el test proporcionado como con los nuestros propios.

Diagrama de estados de la unidad de control:



Resumen de la estructura del autómata:

En el estado **Inicio**, la UC actúa como punto de reposo, con ready a '1' simplemente aguarda peticiones del MIPS, y según la primera condición que se cumpla, selecciona la transición apropiada. Si la solicitud es una dirección desalineada o un intento de escribir un registro interno de solo-lectura, nos quedaremos en Inicio y se carga la dirección en el registro de error activando load_addr_error y poniendo next_error_state en memory_error (la gestión de errores en el resto de estados será similar) y si la dirección pertenece a un registro interno (internal_addr=1 and RE=1) se responde de inmediato vía mux_output="10" sin tocar el bus ni la caché. Para accesos normales, las transiciones se clasifican:

Fetch_inc: invalida la línea en caché si estaba presente (invalidate_bit, inc_inv), tras obtener el árbitro (Bus_req => Bus_grant), salta a send_addr.

"RE and hit"=1 (read-hit): devuelve el dato con mux_output="00" y finaliza.

"WE and hit" (write-hit): solicita el bus (Bus_req), prepara las líneas de escritura (MC_WE0 y MC_WE1) y, al obtenerlo (Bus_grant), pasa a send_addr.

Cualquier miss cacheable o acceso a MD Scratch/IO requiere el bus y también conduce a send_addr.

En **send_addr** se emite la dirección. Fijamos Frame=1, se multiplexa la dirección (MC_send_addr_ctrl=1) y se marca el tipo de operación (MC_bus_Read, MC_bus_Write o MC_bus_Fetch_inc). Si estamos ante un write-miss o read-miss, cogemos la primera palabra del bloque (block_addr=1), si no, para accesos no cacheables, block_addr=0, mandando solo la dirección de la palabra correspondiente. Cuando el dispositivo destino reconoce la dirección (Bus_DevSel=1) se avanza a **send_data**.

En **send_data** se lleva a cabo la fase de envío de datos.

Para read-miss y write-miss se mantiene Frame y MC_bus_Read mientras bus_TRDY=0, cuando el dato llega (bus_TRDY=1) se escribe en la vía elegida (MC_WE0 o MC_WE1 según via_2_rpl), se cuenta la palabra (count_enable), se selecciona como origen de los datos el bus (mux_origen=1) y en la última palabra (last_word_block=1), se cierra la ráfaga con last_word, MC_tags_WE y count_enable (para que palabra haga overflow y vuelva a ser 0). Hay que destacar que ready no se pone a 1, porque aún falta que la instrucción haga hit y obtenga el dato de caché.

Para write-hit se activa MC_send_data, MC_bus_Write y Frame, al recibir bus_TRDY se marca last_word para transferir solo la palabra a escribir.

Los accesos no cacheables (MD Scratch o registros de E/S), solo transfieren una palabra (se manda automáticamente last_word): en lectura se activa MC_bus_Read y se selecciona el bus con mux_output="01" tras bus_TRDY = 1, mientras que en escritura se activa MC_bus_Read antes de continuar cuando bus_TRDY se lo permite.

El caso Fetch_inc es similar a una lectura no cacheable pero activando MC_bus_Fetch_inc.

En cuanto a los contadores:

inc_m (miss en caché): se activa cada vez que metemos un nuevo bloque en caché.

inc_w (write en caché): cada vez que se hace un write-hit.

inc_r (read en caché): cada vez que se hace un read-hit.

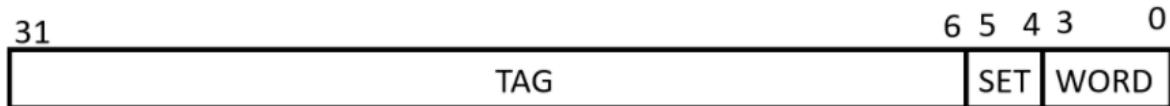
inc_inv (invalidación en caché): cada vez que fetch_inc hace hit.

A continuación, se exponen todas las transiciones del autómata:

Nº TRANSICIÓN	CONDICIÓN	SEÑALES ACTIVAS
1	RE = '0' and WE = '0' and Fetch_inc = '0'	ready <= '1';
2	((RE = '1') or (WE = '1') or (Fetch_inc = '1')) and (unaligned = '1')	ready <= '1'; next_error_state <= memory_error; load_addr_error <= '1';
3	(RE = '1' and internal_addr = '1')	ready <= '1'; mux_output <= "10"; next_error_state <= No_error;
4	((WE = '1') or (Fetch_inc = '1')) and internal_addr = '1'	ready <= '1'; next_error_state <= memory_error; load_addr_error <= '1';
5	(Fetch_inc = '1' and Bus_grant = '0')	Bus_req <= '1';
6	(Fetch_inc = '1' and Bus_grant = '1' and hit = '0')	Bus_req <= '1'; [inc_m <= '1'](si addr_non_cacheable = '0')
7	(Fetch_inc = '1' and Bus_grant = '1' and hit = '1')	Bus_req <= '1'; invalidate_bit <= '1'; inc_inv <= '1';
8	(RE = '1' and hit = '1')	ready <= '1'; inc_r <= '1'; mux_output <= "00";
9	(WE = '1' and hit = '1' and Bus_grant = '0')	Bus_req <= '1'; MC_WEO <= hit0;
10	(WE = '1' and hit = '1' and Bus_grant = '1')	Bus_req <= '1'; MC_WEO <= hit0; MC_WE1 <= hit1;
11	(hit = '0' and Bus_grant = '0')	Bus_req <= '1';
12	(hit = '0' and Bus_grant = '1')	Bus_req <= '1';
13	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and Bus_DevSel = '1')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Read <= '1'; block_addr <= '1'; next_error_state <= No_error;
14	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and Bus_DevSel = '0')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Read <= '1'; block_addr <= '1'; load_addr_error <= '1'; next_error_state <= memory_error; ready <= '1'; mux_output <= "01";
15	(WE = '1' and hit = '1' and addr_non_cacheable = '0' and Bus_DevSel = '1')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Write <= '1';
16	(WE = '1' and hit = '1' and addr_non_cacheable = '0' and Bus_DevSel = '0')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Write <= '1'; load_addr_error <= '1'; next_error_state <= memory_error; ready <= '1'; mux_output <= "01";
17	(addr_non_cacheable = '1' and RE = '1' and Bus_DevSel = '1')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Read <= '1'; inc_r <= '1'; next_error_state <= No_error;
18	(addr_non_cacheable = '1' and RE = '1' and Bus_DevSel = '0')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Read <= '1'; load_addr_error <= '1'; next_error_state <= memory_error; ready <= '1'; mux_output <= "01"; load_addr_error <= '1';
19	(addr_non_cacheable = '1' and WE = '1' and Bus_DevSel = '1')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Write <= '1'; inc_w <= '1'; next_error_state <= No_error;
20	(addr_non_cacheable = '1' and WE = '1' and Bus_DevSel = '0')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Write <= '1'; load_addr_error <= '1'; next_error_state <= memory_error; ready <= '1'; mux_output <= "01";

		Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Fetch_inc <= '1'; next_error_state <= No_error;
21	(Fetch_inc= '1' and Bus_DevSel = '1')	Frame <= '1'; MC_send_addr_ctrl <= '1'; MC_bus_Fetch_inc <= '1'; load_addr_error <= '1'; next_error_state <= memory_error; ready <= '1'; mux_output<="01";
22	(Fetch_inc= '1' and Bus_DevSel = '0')	
23	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and (Fetch_inc = '0')) and (bus_TRDY = '0')	next_state <= send_data; Frame <= '1'; MC_bus_Read <= '1';
24	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and (Fetch_inc = '0')) and((bus_TRDY = '1') and (last_word_block = '0') and (via_2_rpl = '0'))	next_state <= send_data; Frame <= '1'; MC_bus_Read <= '1'; count_enable <= '1'; mux_origen <= '1'; MC_WEO <= '1';
25	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and (Fetch_inc = '0')) and ((bus_TRDY = '1') and (last_word_block = '0') and (via_2_rpl = '1'))	next_state <= send_data; Frame <= '1'; MC_bus_Read <= '1'; count_enable <= '1'; mux_origen <= '1'; MC_WE1 <= '1';
26	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and (Fetch_inc = '0')) and((bus_TRDY = '1') and (last_word_block = '1') and (via_2_rpl = '0'))	next_state <= Inicio; Frame <= '1'; MC_bus_Read <= '1'; last_word <= '1'; mux_origen <= '1'; MC_WEO <= '1'; MC_tags_WE <= '1'; count_enable <= '1'; inc_m <= '1';
27	((RE = '1' or WE = '1') and (hit = '0') and addr_non_cacheable = '0' and (Fetch_inc = '0')) and ((bus_TRDY = '1') and (last_word_block = '1') and (via_2_rpl = '1'))	next_state <= Inicio; Frame <= '1'; MC_bus_Read <= '1'; last_word <= '1'; mux_origen <= '1'; MC_WE1 <= '1'; MC_tags_WE <= '1'; count_enable <= '1'; inc_m <= '1';
28	(WE = '1' and hit = '1' and addr_non_cacheable = '0' and (Fetch_inc = '0')) and bus_TRDY = '1'	Frame <= '1'; MC_send_data <= '1'; MC_bus_Write <= '1'; last_word <= '1'; inc_w <= '1'; ready <= '1'; next_state <= Inicio;
29	(WE = '1' and hit = '1' and addr_non_cacheable = '0' and (Fetch_inc = '0')) and bus_TRDY = '0'	Frame <= '1'; MC_send_data <= '1'; MC_bus_Write <= '1';
30	(addr_non_cacheable = '1') and RE='1' and bus_TRDY='1'	Frame <= '1'; MC_bus_Read <= '1'; mux_output <= "01"; last_word <= '1'; ready <= '1'; next_state<= Inicio; inc_r <= '1';
31	(addr_non_cacheable = '1') and RE='1' and bus_TRDY='0'	Frame <= '1'; MC_bus_Read <= '1';
32	(addr_non_cacheable = '1') and WE='1' and bus_TRDY='1'	Frame <= '1'; MC_bus_Write <= '1'; MC_send_data <= '1'; last_word <= '1'; ready <= '1'; inc_w <= '1';
33	(addr_non_cacheable = '1') and WE='1' and bus_TRDY='0'	Frame <= '1'; MC_bus_Write <= '1'; MC_send_data <= '1';
34	(Fetch_inc= '1') and bus_TRDY='1'	Frame <= '1'; MC_bus_Fetch_inc <= '1'; last_word <= '1'; mux_output <= "01"; ready <= '1';
35	(Fetch_inc= '1') and bus_TRDY='0'	Frame <= '1'; MC_bus_Fetch_inc <= '1';

Descomposición de la dirección:



Análisis detallado:

Tenemos 32 bits de dirección.

- Tamaño de bloque = 4 palabras * 4 B = 16 B.

Bits de word = $\log_2(16)$ = 4 bits de word

De los cuales los bit 3:2 eligen la palabra y 1:0 el byte específico.

- Número de sets:

Bloques totales = $128\text{ B} / (16\text{ B/bloque}) = 8$ bloques

Tenemos asociatividad de 2 vías => $8/2 = 4$ sets

Bits de set = $\log_2(4)$ = 2 bits de set

- Tag:

Son el resto de bits de la dirección: $32 - (4+2) = \underline{26\text{ bits de tag}}$

Análisis de latencias de las distintas transferencias en el bus:

- **CrB(MD):** $3(L) + 3*1(R) = 6$ ciclos
- **CwW(MD):** $3(\text{hit})/ 9(\text{miss})$ ciclos
- **Clwi(MD):** 3 ciclos (4 si lw_inc accede a memoria de datos (por el ciclo de incremento interno de la dirección en la RAM))
- **CrW(MDscratch):** 3 ciclos
- **CwW(MDscratch):** 3 ciclos
- **CrW or CwW(IO):** 1 ciclo

Expresión de cálculo de los ciclos efectivos:

$$C_{ef} = 1 + \frac{\frac{Nr.h + Nr.int + Nw.int + Nr.md.miss(Crb.md + 1 + 1.5) + Nw.md.hit(CwW.md + 1.5) + Nw.md.miss(CrB.md + CwW.md + 2 \cdot 1.5) + Nr.scratch(CrW.mds + 1.5) + Nw.scratch(CwW.mds + 1.5) + Nlwi(Clwi + 1.5)}{\Sigma_{inst}}}{\Sigma_{inst}}$$

Pruebas:

Test realizado	Página
Test read miss y read hit	8
Test write miss y write hit (write-through)	9
Test de Fech-inc	11
Test de lectura con lw en memoria scratch	13
Test de escritura con sw en memoria scratch	14
Test de Fech-inc en memoria scratch	15
Test general	16
Test reg interno MC -> Addr_error_reg	19
Test registros Addr_input y Addr_output	19

¿Cómo se comportan los bloques y vías en la caché?

Todos nuestros test que trabajan con caché, utilizan el test original proporcionado por el profesorado para desarrollar las distintas pruebas.

Los distintos test tienen particularidades propias, que hacen que se escriban distintas cosas en palabras de los bloques, o que provocan que se metan 2 bloques con mismo set en caché por cada iteración (si zona set no está llena, se meterán en orden, y si está llena, se meterán los bloques según dicte la política FIFO).

La idea del bucle es que en cada iteración se aumente el set, cuando se rellena toda la vía 0, empieza desde el primer set otra vez, llenando la vía 1. Cuando la vía 1 está llena, expulsará bloques de la vía 0 y cuando se acabe de llenar la vía 0, se expulsarán bloques de la vía 1 y se irá llenando. Así en bucle, hasta que la dirección de memoria sea mayor de la permitida.

Aquí la tabla de rellenado en el primer test (igual o muy similar para el resto de test):

Bloque	Dirección base	Dirección[5:4]	Vía elegida	Bloque expulsado
0	0x0000	0	0	-
1	0x0010	1	0	-
2	0x0020	2	0	-
3	0x0030	3	0	-
4	0x0040	0	1	-
5	0x0050	1	1	-
6	0x0060	2	1	-
7	0x0070	3	1	-
8	0x0080	0	0	expulsa 0x0000
9	0x0090	1	0	expulsa 0x0010
10	0x00A0	2	0	expulsa 0x0020
11	0x00B0	3	0	expulsa 0x0030
12	0x00C0	0	1	expulsa 0x0040
13	0x00D0	1	1	expulsa 0x0050
14	0x00E0	2	1	expulsa 0x0060
15	0x00F0	3	1	expulsa 0x0070
...

Test read miss y read hit:

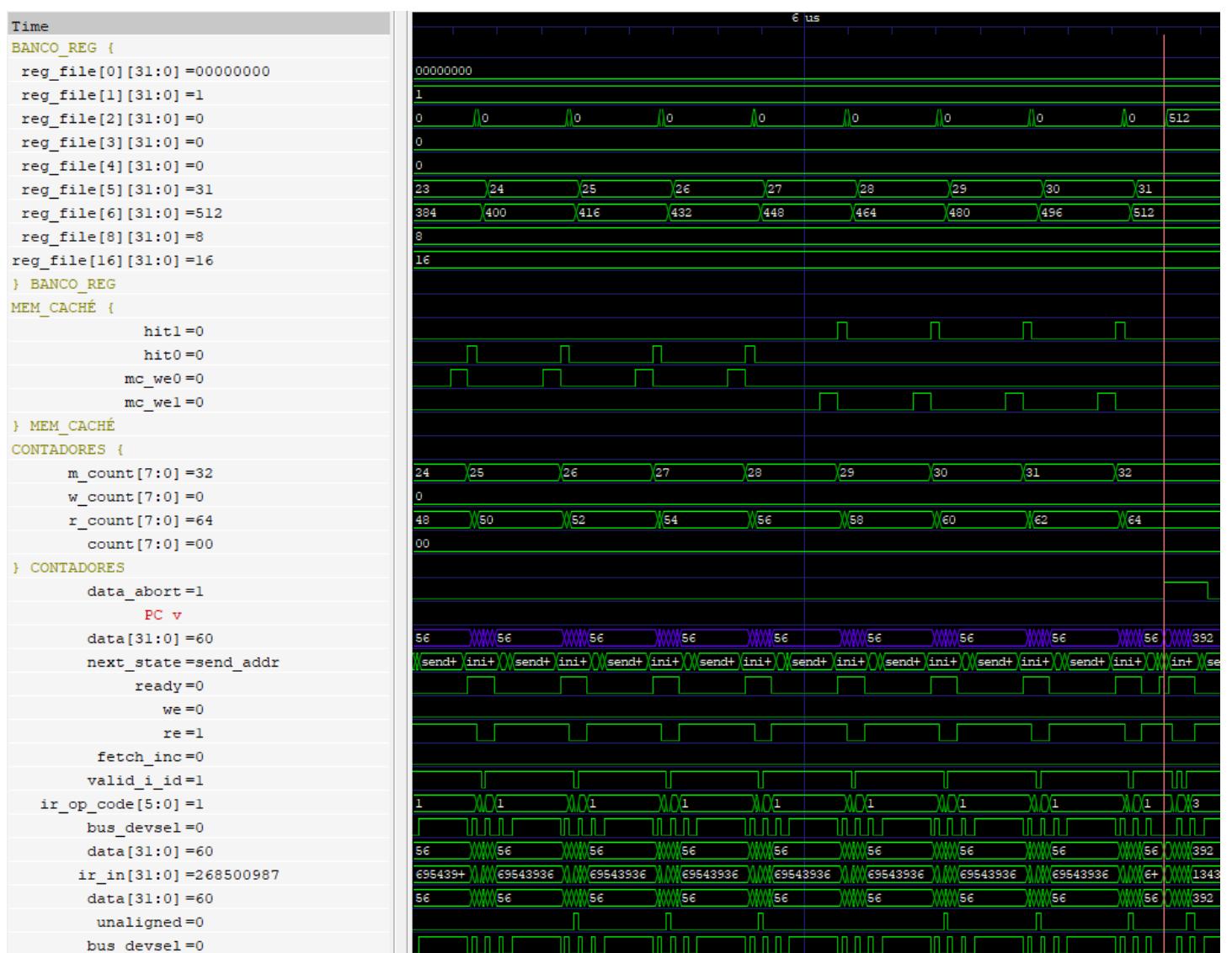
Utilizamos el test proporcionado por los profesores. Este test nos servirá como plantilla para muchos otros test, lo indicaremos cuando sea así.

Memorias que se usan:

Memoria instrucciones: **TEST ORIGINAL**

Memoria datos: **DATOS ORIGINALES**

Memoria Scratch: **SCRATCH**

Captura de GTKWave:

Test write miss y write hit (write through):

En el test original, cambiamos los 2 primeros lw del bucle por dos sw, para que el primero haga un *write miss* en la primera palabra del bloque (escribe un 1) y el segundo un *write hit* en la segunda palabra (escribe un 8).

Cada vez que se haga un sw se tiene que escribir el número correspondiente en la memoria de datos.

Memorias que se usan:

Memoria instrucciones: TEST WT (PRUEBAS CON SW)

Memoria datos: DATOS ORIGINALES

Memoria Scratch: SCRATCH

Código:

```

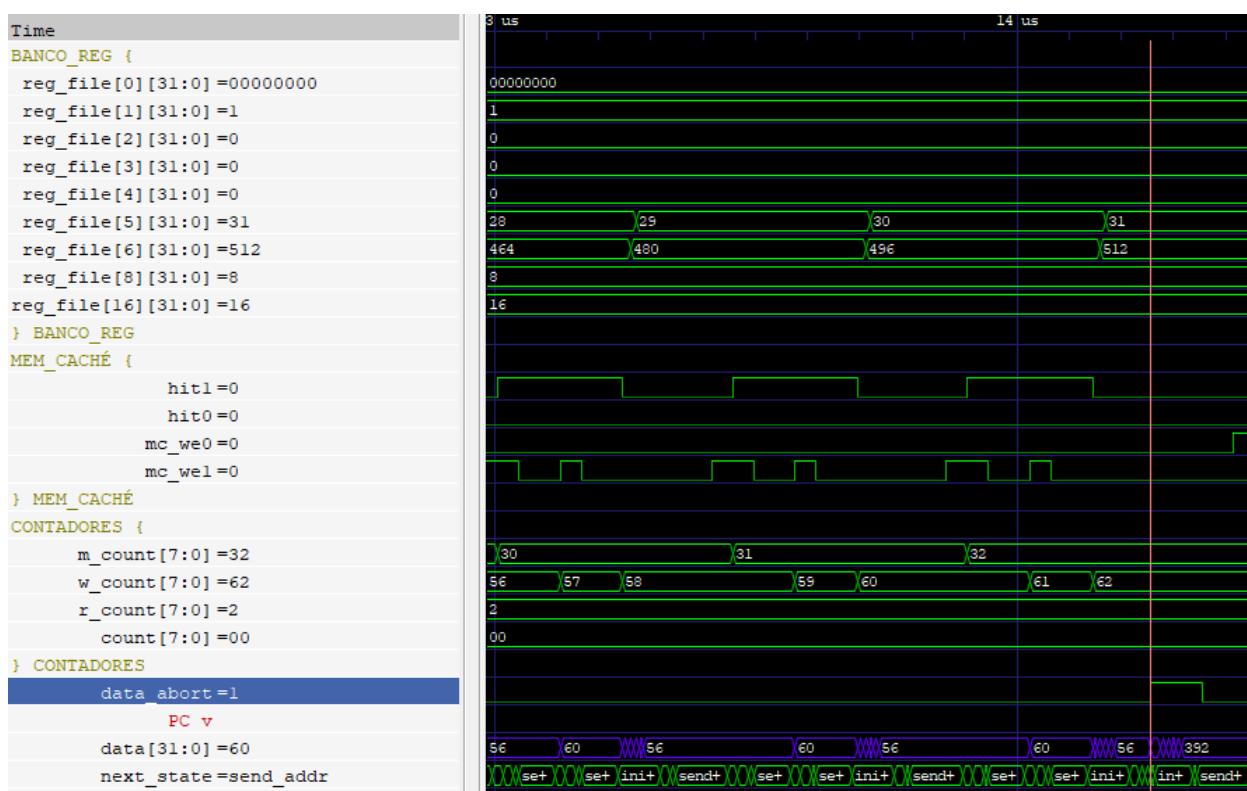
10210003 -- beq R1, R1, INI
1021003E -- beq R1, R1, RTI
1021005D -- beq R1, R1, RT_Abort
1021006C -- beq R1, R1, RT_UNDEF

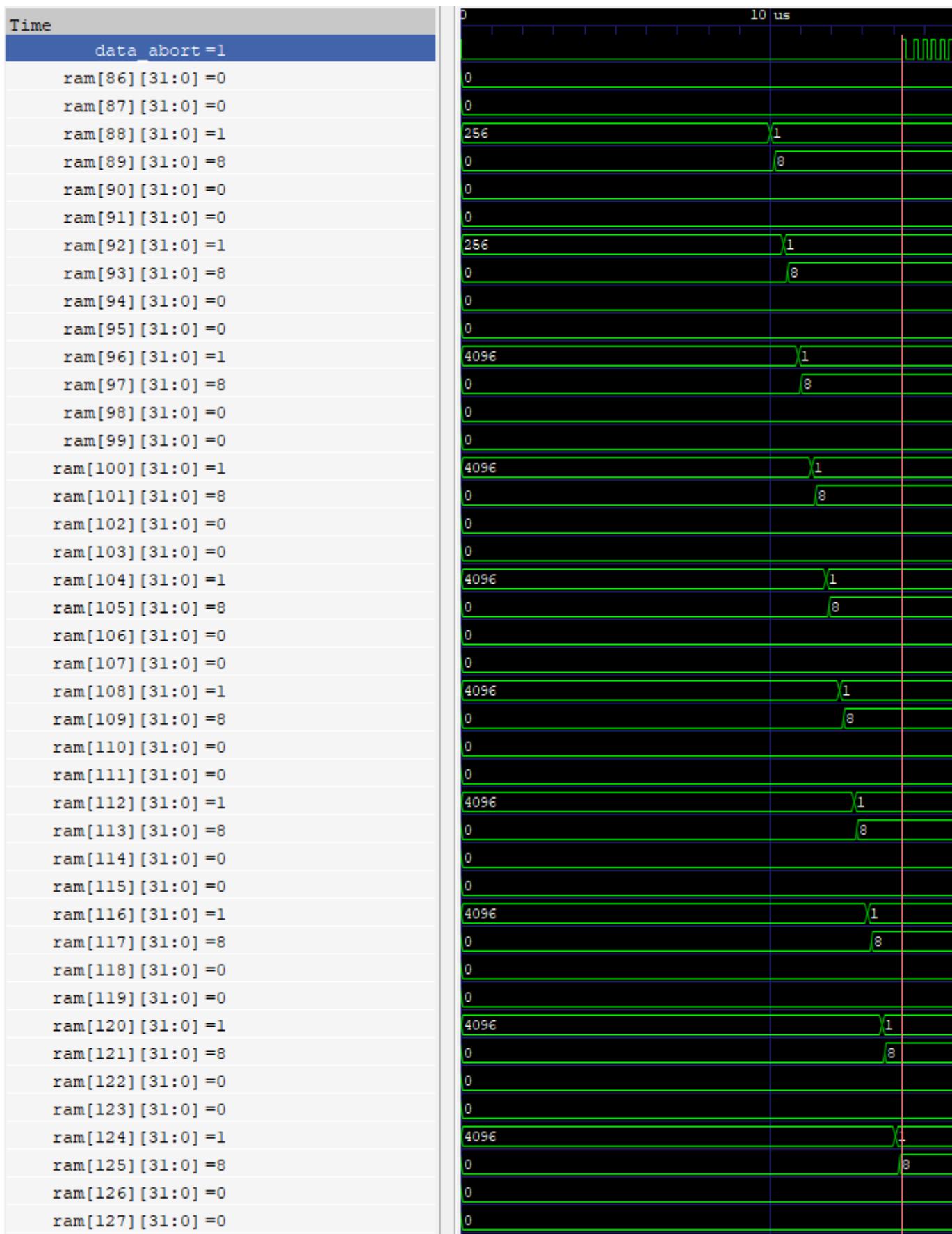
INI 08010000 -- Lw R1, 0(r0)
      04212000 -- Add R4, R1,R1
      04842000 -- Add R4, R4,R4
      04844000 -- Add R8, R4,R4
      05088000 -- Add R16, R8,R8
      06003000 -- Add R6, R16,R0
      08040004 -- Lw R4, 4(r0)

bucle1 0CC10000 -- sw r1, 0(r6)
        0CC80004 -- sw r8, 4(r6)
        06063000 -- Add R6, R16,R6
        04252800 -- Add R5, R1,R5
        1000FFFFB -- BEQ R0,R0, bucle1

```

Capturas de GTKWave:





Test de Fech-inc:

En el test original, cambiamos los 2 primeros lw del bucle por un lw y 3 lw_inc:

- El primer lw sirve para hacer miss y traer el bloque a memoria caché.
- El primer lw_inc hará hit, por lo que tendrá que poner a 1 el invalidate_bit para invalidar el bloque en caché.
- El segundo lw_inc hará miss, ya que el bloque en caché ha sido invalidado en la instrucción anterior. La instrucción lw_inc no trae el bloque nuevo al hacer miss.
- El tercer lw_inc nos servirá para ver que no se trajo el bloque a caché en la instrucción anterior, se hace miss y por ello no se pone invalidate_bit a 1.

Contadores:

-m_count = 1(lw del principio) + 31(iteraciones)*1(lw en bucle) = 32

-w_count = 0 (no hay)

-r_count = 2(lw's del principio) + 1(lw en el bucle)*31(iteraciones) = 33

-inv_count = 31 (una invalidación por iteración)

Memorias que se usan:

Memoria instrucciones: **TEST FECH-INC**

Memoria datos: **DATOS ORIGINALES**

Memoria Scratch: **SCRATCH**

Código:

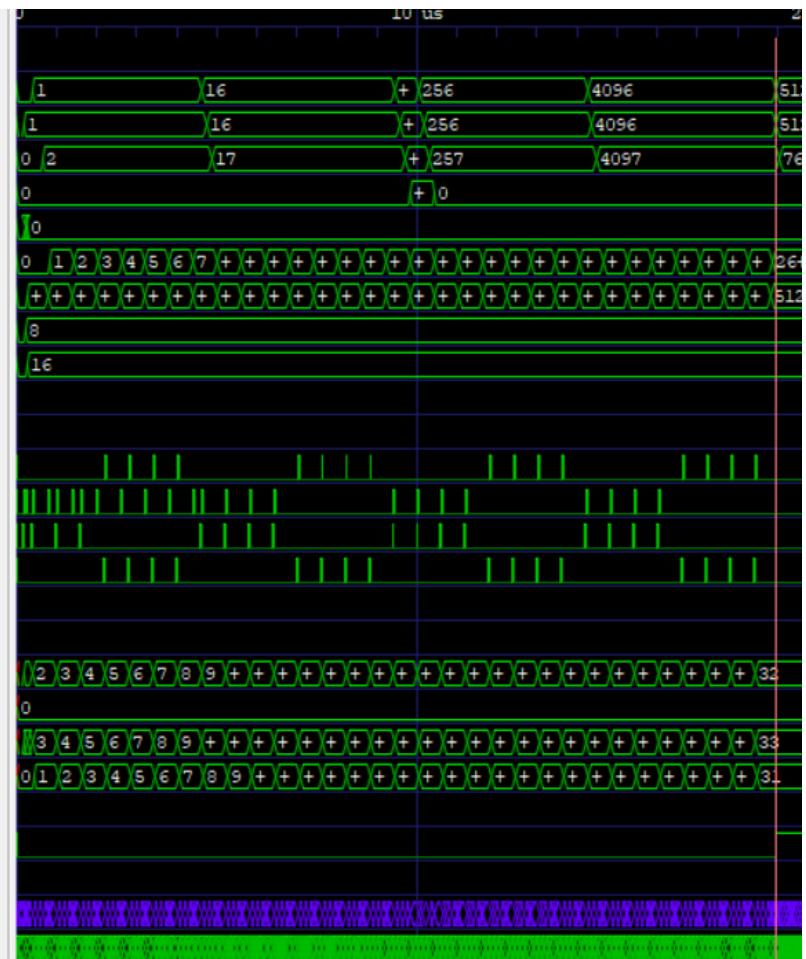
```
10210003 -- beq R1, R1,INI
10210040 -- beq R1, R1,RTI
1021005F -- beq R1, R1,RT_Abort
1021006E -- beq R1, R1,RT_UNDEF
INI 08010000 -- Lw R1, 0(r0)
      04212000 -- Add R4, R1,R1
      04842000 -- Add R4, R4,R4
      04844000 -- Add R8, R4,R4
      05088000 -- Add R16, R8,R8
      06003000 -- Add R6, R16,R0
      08040004 -- Lw R4, 4(r0)
bucle1 08C00000 -- lw r0, 0(r6)
        40C10000 -- lw_inc r1, 0(r6)
        40C20000 -- lw_inc r2, 0(r6)
        40C30004 -- lw_inc r3, 4(r6)
        06063000 -- Add R6, R16,R6
        04252800 -- Add R5, R1,R5
        1000FFF9 -- BEQ R0,R0, bucle1
```

Capturas de GTKWave:

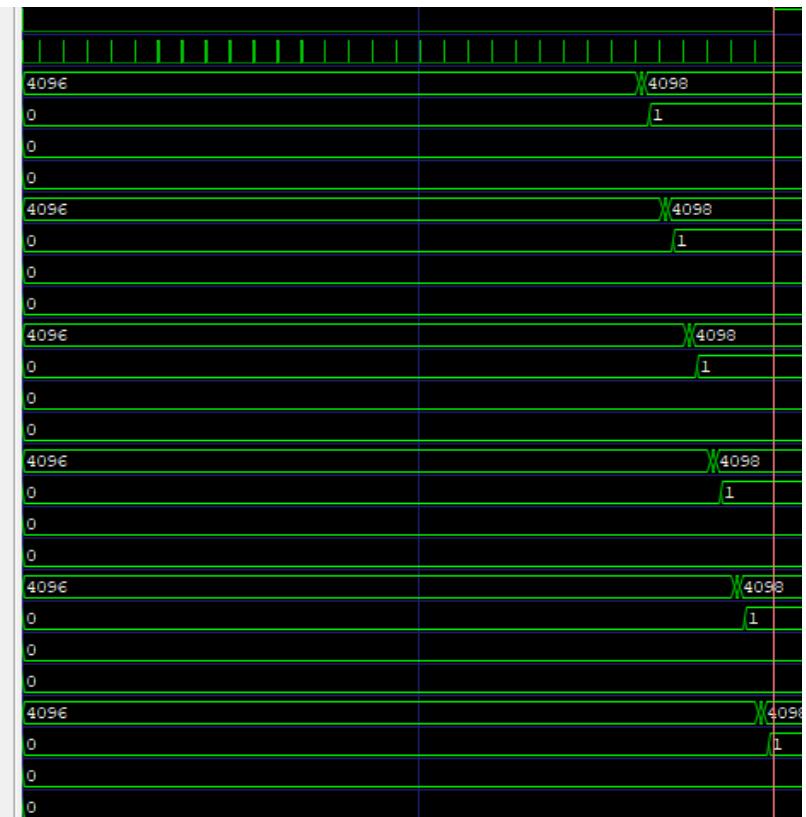
```

Time
BANCO_REG {
    reg_file[0][31:0] = 4096
    reg_file[1][31:0] = 4096
    reg_file[2][31:0] = 4097
    reg_file[3][31:0] = 0
    reg_file[4][31:0] = 0
    reg_file[5][31:0] = 268470151
    reg_file[6][31:0] = 512
    reg_file[8][31:0] = 8
    reg_file[16][31:0] = 16
} BANCO_REG
MEM_CACHÉ {
    hit1=0
    hit0=0
    mc_we0=0
    mc_wel=0
} MEM_CACHÉ
CONTADORES {
    m_count[7:0] = 32
    w_count[7:0] = 0
    r_count[7:0] = 33
    count[7:0] = 31
} CONTADORES
        data_abort=1
        PC v
        data[31:0] = 60
        next_state = send_addr

```



```
    data_abort=1  
invalidate_bit=0  
ram[104][31:0]=4098  
ram[105][31:0]=1  
ram[106][31:0]=0  
ram[107][31:0]=0  
ram[108][31:0]=4098  
ram[109][31:0]=1  
ram[110][31:0]=0  
ram[111][31:0]=0  
ram[112][31:0]=4098  
ram[113][31:0]=1  
ram[114][31:0]=0  
ram[115][31:0]=0  
ram[116][31:0]=4098  
ram[117][31:0]=1  
ram[118][31:0]=0  
ram[119][31:0]=0  
ram[120][31:0]=4098  
ram[121][31:0]=1  
ram[122][31:0]=0  
ram[123][31:0]=0  
ram[124][31:0]=4098  
ram[125][31:0]=1  
ram[126][31:0]=0  
ram[127][31:0]=0
```



Test de lectura con lw en memoria scratch:

En este pequeño test guardamos en r0 la primera dirección de memoria scratch, después guardamos en r1 la segunda palabra de memoria scratch. Como el acceso a scratch es no cacheable (no se hace ni hit ni miss), no traemos ningún bloque a caché en el segundo lw, tan solo nuestra UC se encarga únicamente de traer la palabra de scratch.

Contadores:

-m_count = 1 (el primer lw accede a memoria de datos normal)
 -r_count = 1 (el primer lw accede a memoria de datos normal)

Memorias que se usan:

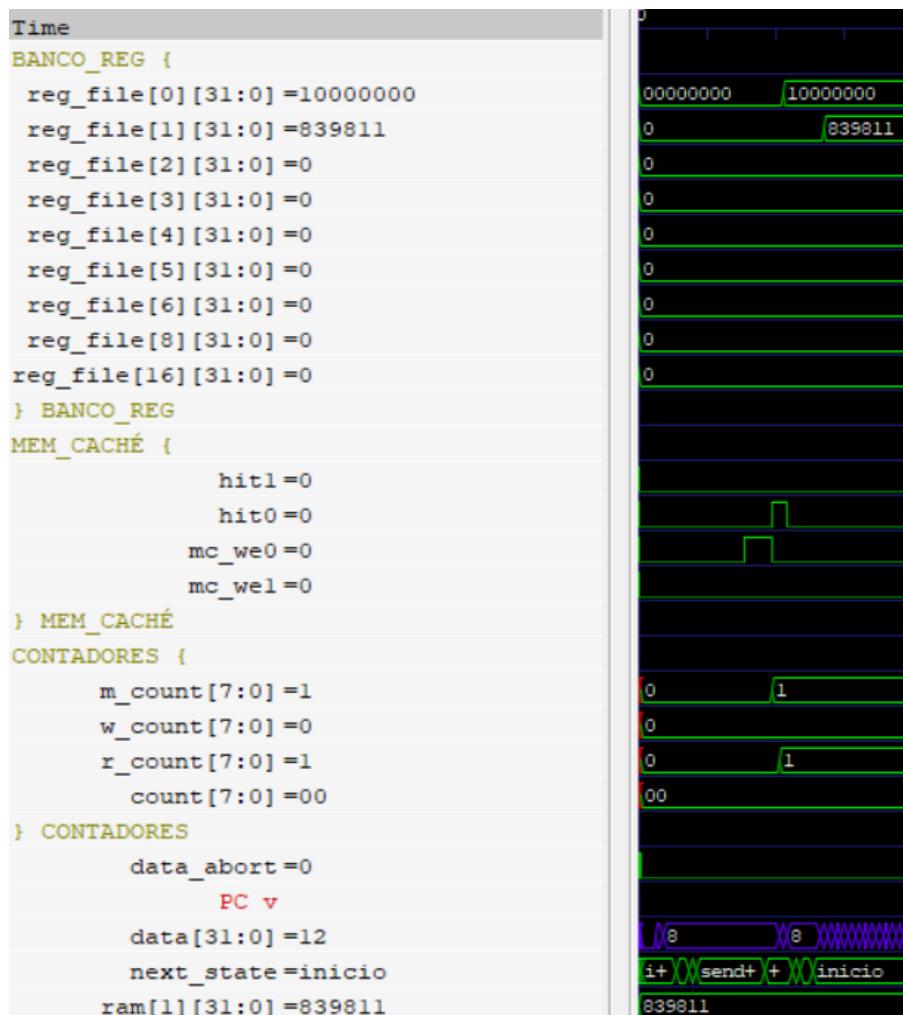
Memoria instrucciones: **TEST SCRATCH (PRUEBAS CON LW)**

Memoria datos: **DATOS ORIGINALES**

Memoria Scratch: **SCRATCH**

Código:

```
08000100 -- lw r0, 256(r0)
08010004 -- lw r1, 4(r0)
1000FFFF -- beq fin
```

Captura de GTKWave:

Test de escritura con sw en memoria scratch:

Cargamos en r1 la primera dirección de la memoria scratch, después en r2 leemos la quinta palabra de la memoria de datos y la guardamos en la primera palabra de memoria scratch. De nuevo, el sw no generará ni miss ni hit, al acceder a memoria scratch será tomado como instrucción no cacheable y la UC se limitará a escribir la palabra en la dirección adecuada.

Contadores:

- m_count = 2 (los 2 primeros lw acceden a distintos bloques en memoria de datos normal)
- w_count = 0 (el sw en memoria scratch es no cacheable)
- r_count = 2 (los dos lw en memoria de datos normal)

Memorias que se usan:

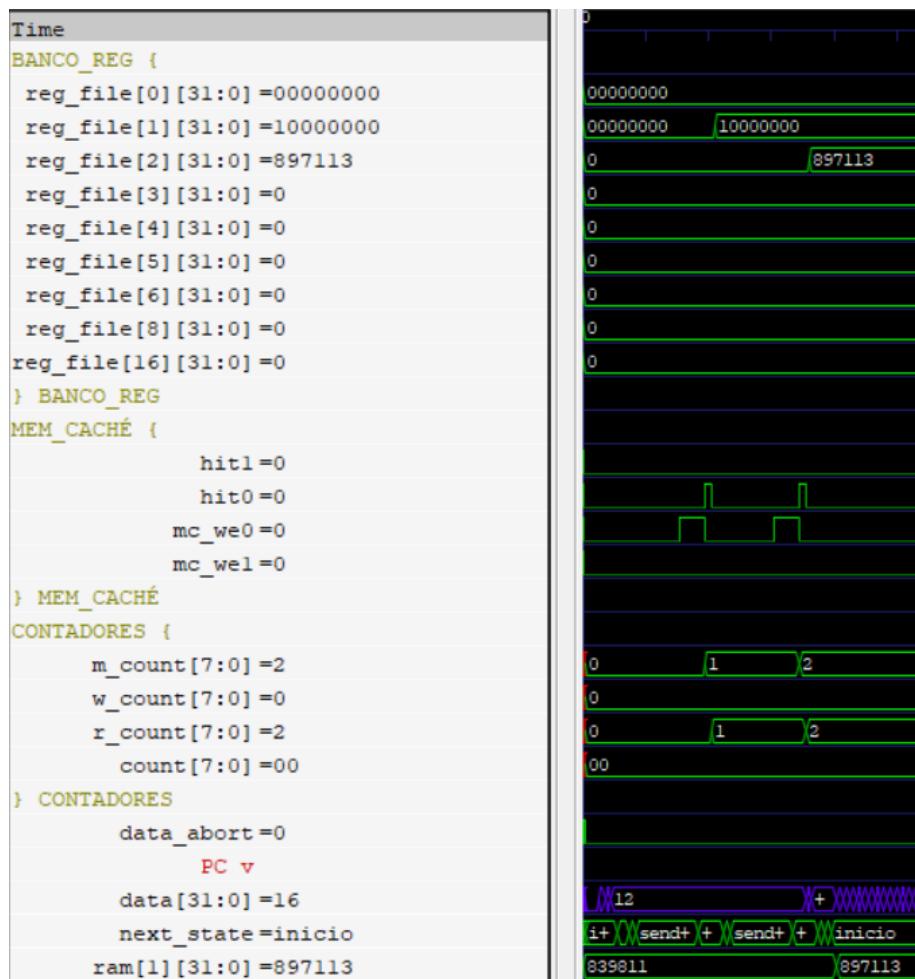
Memoria instrucciones: **TEST SCRATCH (PRUEBAS CON SW)**

Memoria datos: **DATOS SCRATCH sw y SCRATCH lw_inc**

Memoria Scratch: **SCRATCH**

Código:

```
08010100 -- lw r1, 256(r0)
08020014 -- lw r2, 20(r0)
0C220004 -- sw r2, 4(r1)
1000FFFF -- beq fin
```

Captura de GTKWave:

Test de Fech-inc en memoria scratch:

Los lw_inc en scratch se tienen que comportar como lw normales. Por ello, si la instrucción hace referencia a una dirección en memoria scratch, este lw_inc se tomará como una instrucción no cacheable.

En este test probamos 4 lw_inc y comprobamos que efectivamente funcionan como lw normales.

Contadores:

m_count = 1 (el del primer lw)

r_count = 1 (el del primer lw)

Memorias que se usan:

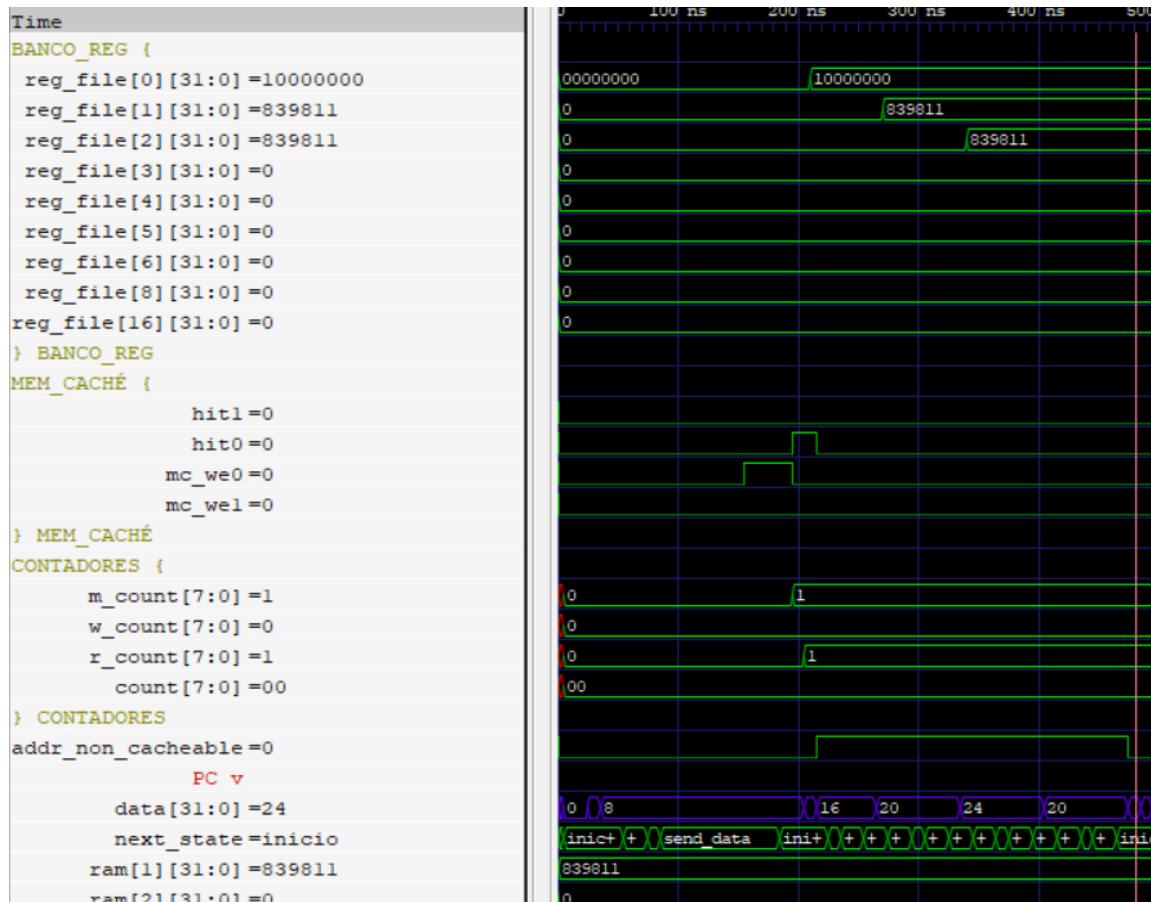
Memoria instrucciones: **TEST SCRATCH (PRUEBAS CON LW_INC)**

Memoria datos: **DATOS SCRATCH sw y SCRATCH lw_inc**

Memoria Scratch: **SCRATCH**

Código:

```
08000100 -- lw r0, 256(r0)
40010004 -- lw_inc r1, 4(r0)
40020004 -- lw_inc r2, 4(r0)
40030008 -- lw_inc r3, 8(r0)
40040008 -- lw_inc r4, 8(r0)
1000FFFF -- beq fin
```

Captura de GTKWave:

Test general:

En el test original, ponemos un lw que cargue en r0 la primera dirección de la memoria scratch y cambiamos los 2 primeros lw del bucle por:

- Un add que ayuda a recorrer la memoria scratch por bloques.
- 2 sw que apuntan a la primera palabra de un bloque de memoria de datos y un bloque de memoria scratch respectivamente. El primero será un miss y traerá a caché el bloque al que se referencia. Ambos escribirán en su respectiva palabra un 8.
- 2 lw que apuntan a la primera palabra de un bloque de memoria de datos y un bloque de memoria scratch respectivamente. El primero será un hit. Se deberá mantener en r1 y r2 un 8.
- 2 lw_inc que apuntan a la primera palabra de un bloque de memoria de datos, el primero será un hit y deberá poner invalidate_bit a 1, el segundo será miss. Además, ambos deberán escribir primero un 8 y luego un 9 en r3 y aumentar la palabra a la que se referencia en memoria de datos primero a 9 y luego a 10.
- 2 lw_inc que apuntan a la primera palabra de un bloque de memoria scratch, ambos deberán actuar como lw's normales, manteniendo en r4 un 8 y en la palabra de memoria scratch otro 8.

El programa “muere” por un data_abort provocado al intentar acceder a una dirección de memoria más allá de la asignada a la memoria scratch (10000100>100000FF).

Contadores:

(iteraciones = (r5 en el data_abort)/8 = 15)

-m_count = 2(lw's anteriores al bucle) + 15(iteraciones)*1(miss de primer sw) + 1 (primer sw antes de que el programa “explote”) = 18

-w_count = 1(sw cacheable en el bucle)*15(iteraciones) + 1(primer sw antes de que el programa “explote”) = 16

-r_count = 3(lw's anteriores al bucle) + 1(lw cachable en el bucle)*15(iteraciones) = 18

-inv_count = 1(lw_inc que provoca invalidate_bit=1 en el bucle)*15(iteraciones) = 15

Memorias que se usan:

Memoria instrucciones: **TEST GENERAL**

Memoria datos: **DATOS ORIGINALES**

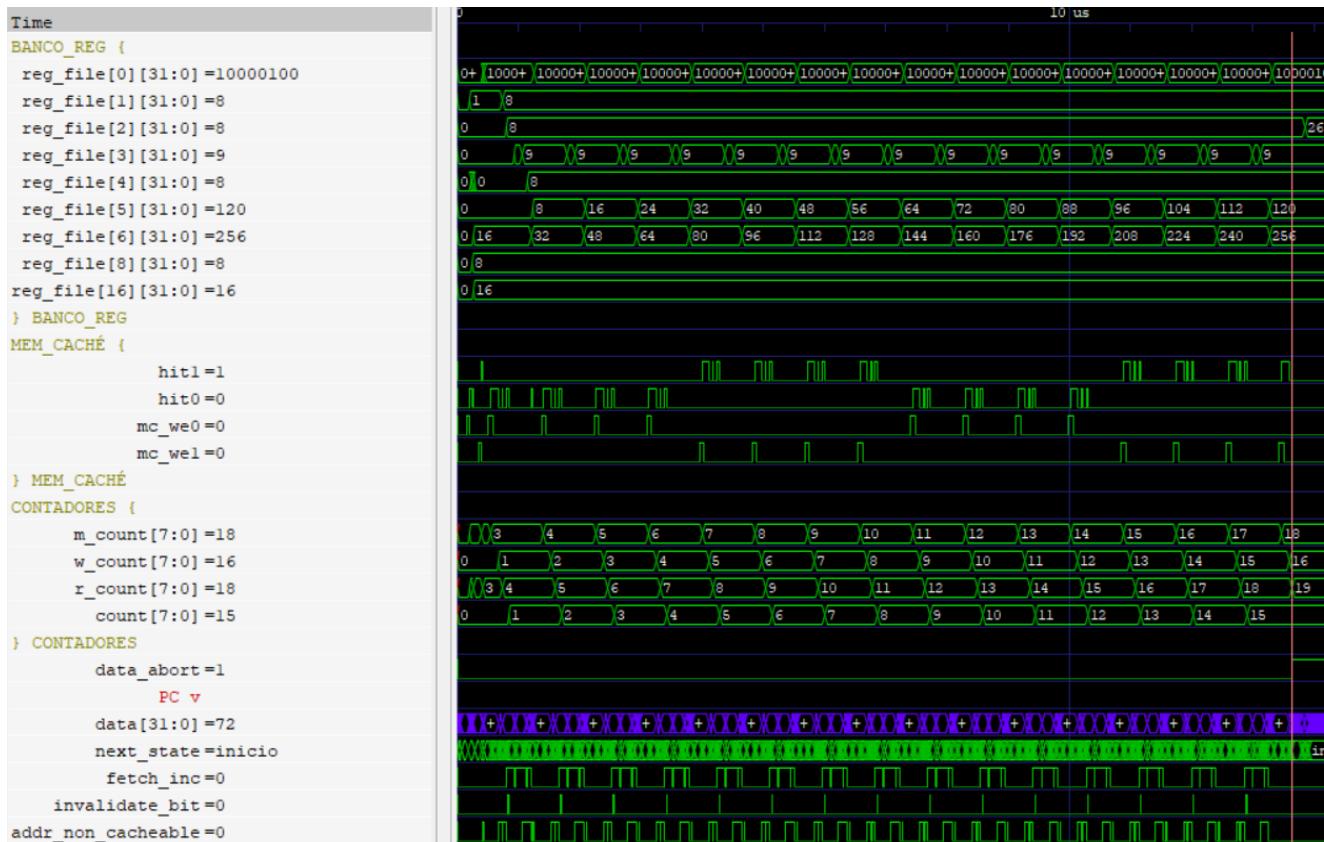
Memoria Scratch: **SCRATCH**

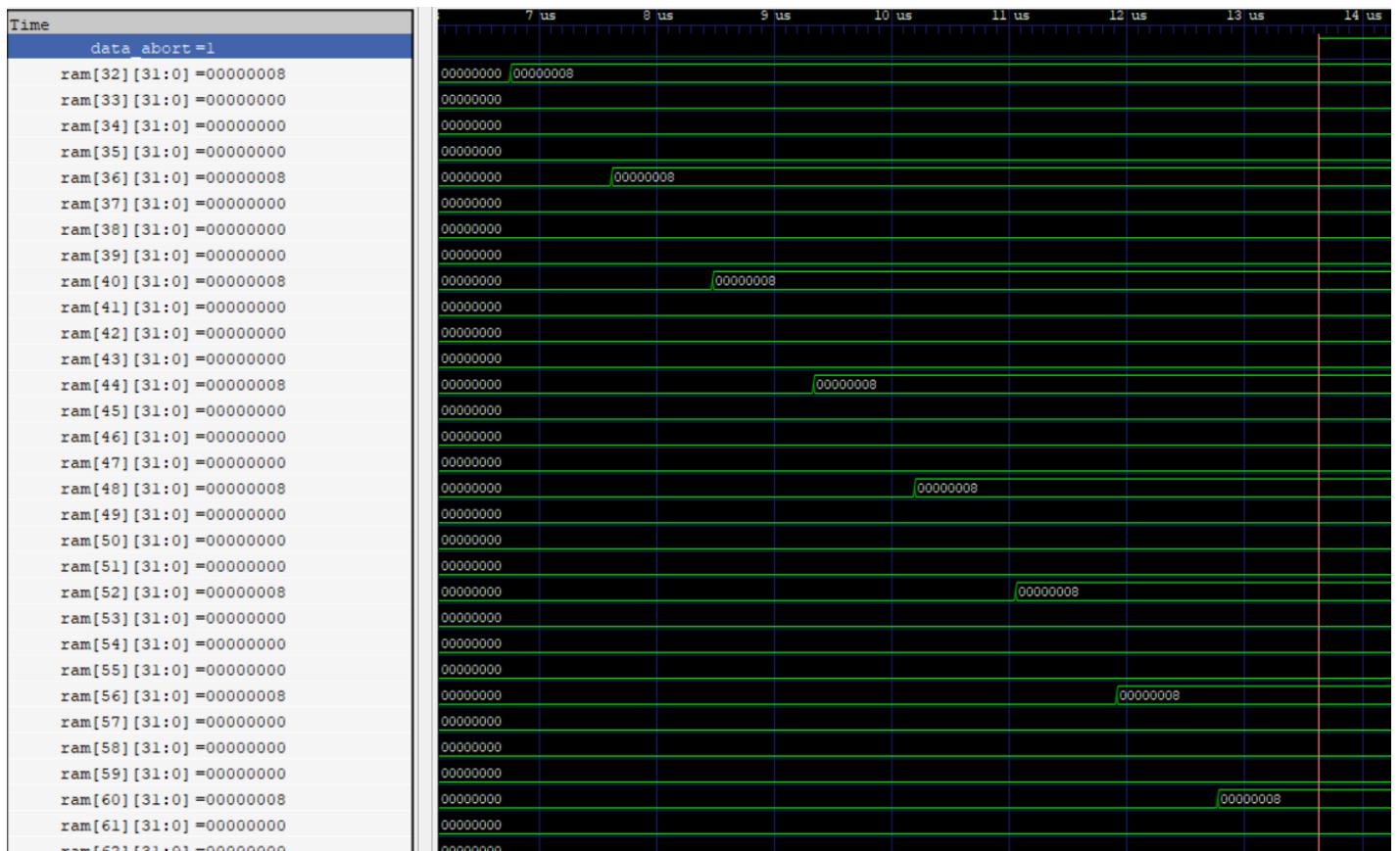
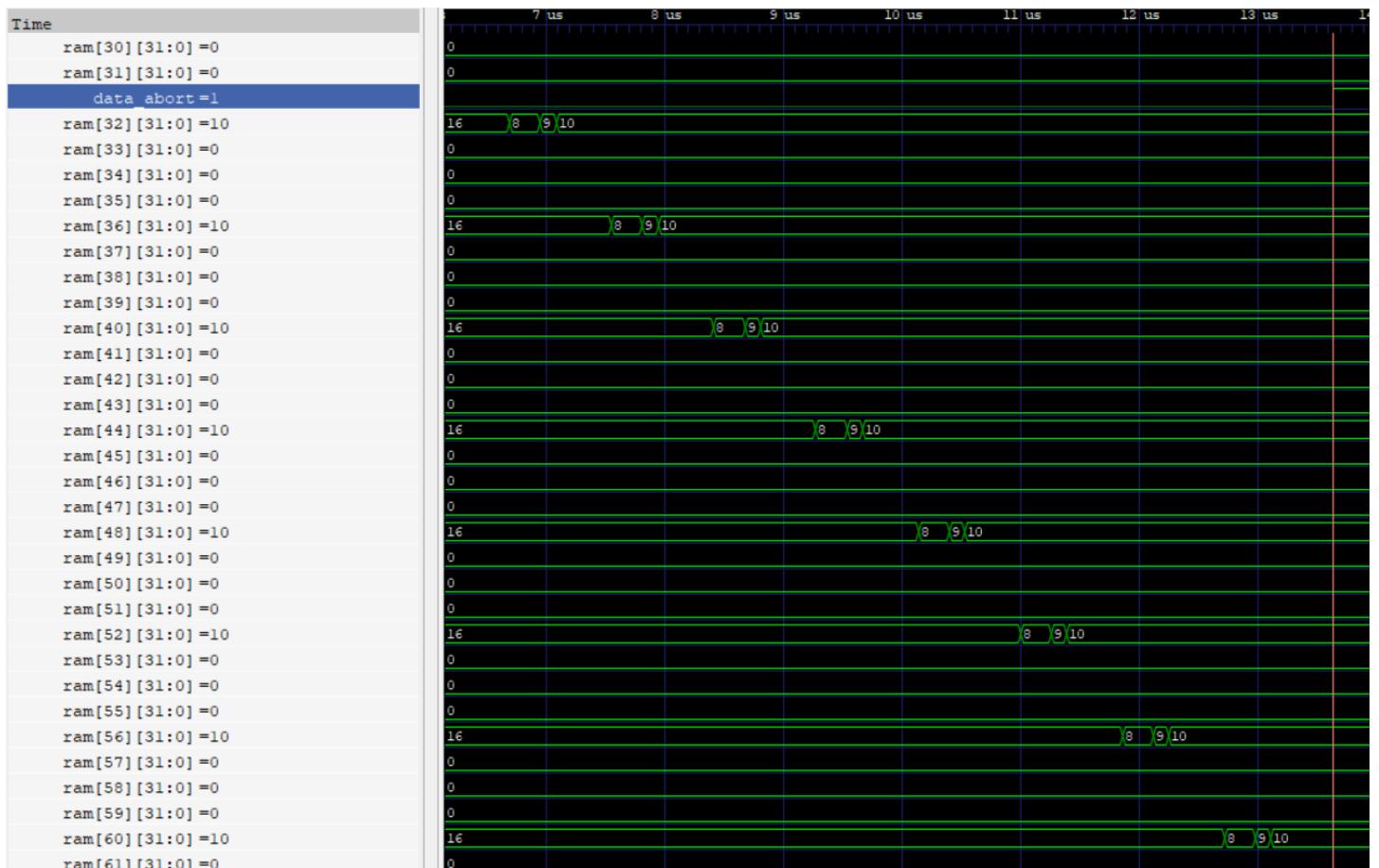
Código:

```

10210003 -- beq R1, R1,INI
10210046 -- beq R1, R1,RTI
10210065 -- beq R1, R1,RT_Abort
10210074 -- beq R1, R1,RT_UNDEF
INI 08010000 -- Lw R1, 0(r0)
04212000 -- Add R4, R1,R1
04842000 -- Add R4, R4,R4
04844000 -- Add R8, R4,R4
05088000 -- Add R16, R8,R8
06003000 -- Add R6, R16,R0
08040004 -- Lw R4, 4(r0)
08000100 -- lw r0, 256(r0)
bucle1 06000000 -- add r0, r16, r0
0CC80000 -- sw r8, 0(r6)
0C080000 -- sw r8, 0(r0)
08C10000 -- lw r1, 0(r6)
08020000 -- lw r2, 0(r0)
40C30000 -- lw_inc r3, 0(r6)
40C30000 -- lw_inc r3, 0(r6)
40040000 -- lw_inc r4, 0(r0)
40040000 -- lw_inc r4, 0(r0)
06063000 -- Add R6, R16,R6
04252800 -- Add R5, R1,R5
1000FFF4 -- BEQ R0,R0, bucle1

```

Capturas de GTKWave:



Test reg interno MC -> Addr error reg:

Intentamos hacer una escritura y una lectura del registro interno de la MC. La escritura produce un error de memory_error.

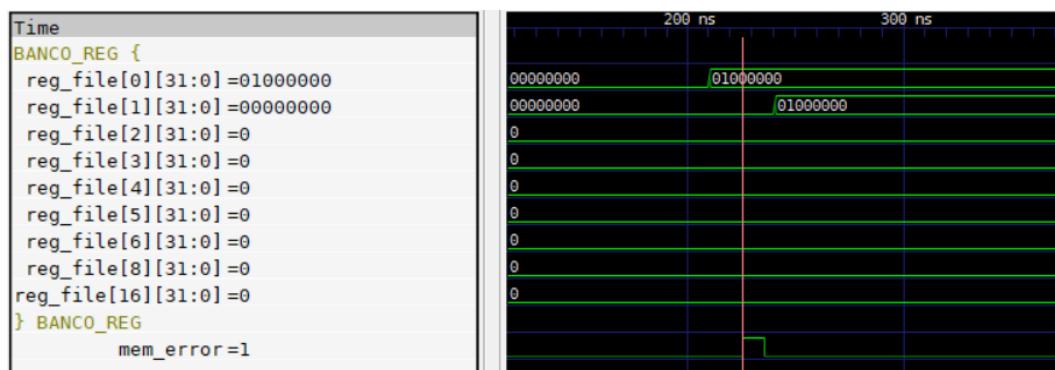
Código:

```
lw r0, 264(r0) 08000108
sw r1, 0(r0)   0C010000
lw r1, 0(r0)   08010000
```

Memorias que se usan:

Memoria instrucciones: **TEST ESCRITURA REGISTRO**

Memoria datos: **DATOS ORIGINALES**

Captura GTKwave:**Test registros Addr_input y Addr_output:**

En este pequeño test intentamos hacer escrituras tanto en el registro addr_input como en el addr_output. La escritura en el registro addr_input no se produce.

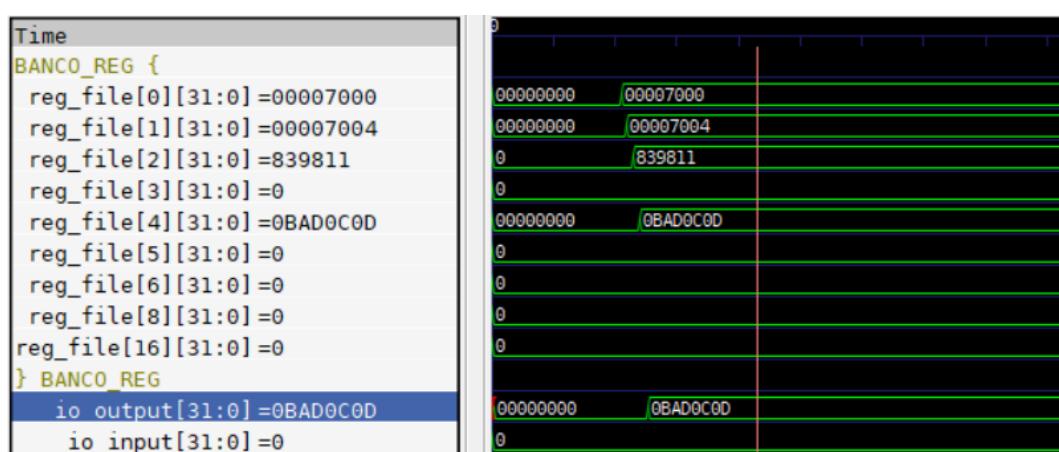
Código:

```
LW R0, 0(R0)-----08000000
LW R1, 4(R5)-----08A10004
LW R2, 8(R3)-----08620008
LW R4, 12(R3)-----0864000C
SW R2, 0(R0)-----0C020000
SW R4, 0 (R1)-----0C240000
```

Memorias que se usan:

Memoria instrucciones: **TEST ADD_INP AND ADD_OUT**

Memoria datos: **DATOS REGISTROS IO**

Captura GTKwave:

Prueba de integración

Como prueba de integración utilizamos nuestro test general.

$$S_{up} = \frac{T_{ex}(Sin\ Cache)}{T_{ex}(Cache)} = \frac{I \cdot CPI(Sin\ Cache) \cdot T_c}{I \cdot CPI(Cache) \cdot T_c} = \frac{CPI(Sin\ Cache)}{CPI(Cache)} = \frac{C_{eff}(Sin\ Cache)}{C_{eff}(Cache)}$$

► Ceff sin Caché:

$$Cef(no\ cache) = Cbus + L = 3 + 1.5 = 4.5$$

► Ceff con Caché:

$$Cef = 1 + \frac{\frac{Nrh + Nr.int + Nw.int + Nr.md.miss(Crb.md + 1 + 1.5) + Nw.md.hit(CwW.md + 1.5) + Nw.md.miss(CrB.md + CwW.md + 2 \cdot 1.5) + Nr.scratch(CrW.mds + 1.5) + Nw.scratch(CwW.mds + 1.5) + Nlwi(Clwi + 1.5)}{\Sigma inst}}$$

Consideramos los siguientes valores:

$$Cef = 1 + \frac{\frac{16(Nrh) + 0 + 0 + 2(Nr.md.miss) \cdot (6 + 1 + 1.5) + 0 + 16(Nw.md.miss) \cdot (6 + 3 + 2 \cdot 1.5) + 15(Nr.scratch)(3 + 1.5) + 15(Nw.scratch)(3 + 1.5) + 60}{(3 + 1.5)}}{194}$$

Finalmente nuestro valor de speed up es:

$$Cef = 4.24$$

Por lo tanto nuestro speedup es:

$$Speedup = \frac{4.5}{4.24} = 1.0613$$

Una mejora de un 6% no parece gran cosa, sin embargo, hay que tener en cuenta que el programa probado para calcular los Cef es un programa muy exigente para caché, ya que aprovecha muy poco sus cualidades (muchos write-miss y muy pocos hits en proporción). Por ello, aunque el test va en contra de las virtudes de la caché, es un rasgo muy positivo que aún se muestre mejora al calcular el speedup.

Por falta de tiempo, no hemos podido crear un programa que pruebe varias características de caché y que aprovechará más el poder del componente. Sin embargo, prevemos que el speedup con un programa de estas características tendría mucho mejor porcentaje de mejora.

Cuantificación de horas dedicadas:

	Ana	Fernando
Estudio de los componentes proporcionados	3	3
Diseño de la unidad de control	8	8
Arreglo de errores	3	3
Depuración, verificación y programas de prueba	20	20
Memoria	10	10
TOTAL	44	44

El trabajo ha sido realizado por los 2 miembros del equipo a la vez, por lo tanto, hemos realizado las mismas horas de trabajo. Diseñar la unidad de control nos ha llevado bastante tiempo, ya que había un gran número de señales a tener en cuenta. La depuración ha sido la parte más costosa, ya no solo por la verificación del correcto funcionamiento sino también por el planteamiento de nuestros propios test. La memoria ha sido un trabajo largo ya que hemos querido explicar correctamente todos los pasos.

Conclusiones y autoevaluación:

	CONCLUSIONES	AUTOEVALUACIÓN:
Ana	Trabajar con Fernando ha sido de nuevo una gran experiencia ya que hemos compartido conocimientos y nos hemos ayudado el uno al otro. Este trabajo me ha ayudado a entender mejor los conceptos adquiridos.	7,5
Fernando	Para mí, hacer este proyecto y el anterior han sido una manera muy eficiente para comprender las partes más complejas de los contenidos de la asignatura. Me ha ayudado mucho trabajar en un entorno interactivo y tener compañeros que me apoyaron para entender ciertos conceptos.	7,5

Ya que no nos ha dado tiempo a realizar las tareas optativas, creemos que nuestra calificación sería un 7,5. Creemos que hemos hecho un buen trabajo.

Agradecimientos:

Un gran agradecimiento a Juan José Muñoz Lahoz (902677) y Marcos San Julián Fuertes (899849), han sido indispensables ya que hemos podido poner ideas en común y comparar resultados.