Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad de Zaragoza

Facultad de Ingeniería

in Memoria Proyecto 2

AOC II

Data Memory Hierarchy

Eduardo Sánchez Sarsa- 901813

Athanasios Usero Samarás - 839543

Mayo de 2025

Índice

1. Resumen **3**

2. GRAFO DE ESTADOS DE LA UNIDAD DE CONTROL **5**
2.1. Descripción 5
2.2. Grafo de estados para el controlador principal 6
2.2.1. Tabla de comportamiento de estado (Moore) 7
2.2.2. Tabla de transiciones (Mealy) 7
2.2.3. Decisiones de diseño 10

3. DESCOMPOSICIÓN DE LA DIRECCIÓN **15**
3.1. Descripción 15
3.2. Descomposición 15

4. ANÁLISIS DE LATENCIA EN TRANSFERENCIAS DE BUS **17**
4.1. Descripción 17
4.2. Cálculo y simulación para obtener latencias 17

5. FORMULACIÓN MATEMÁTICA DE LOS CICLOS EFECTIVOS **22**
5.1. Descripción 22
5.2. Fórmula general 22

6. TEST UNITARIOS **25**
6.1. Descripción 25
6.2. Tests sobre MD 26
6.2.1. Read Miss 26
6.2.2. Read Hit 29
6.2.3. Write Miss 31
6.2.4. Write Hit 33
6.2.5. lw_inc 35
6.3. Tests sobre MD_Scratch 38
6.3.1. Read 38
6.3.2. Write 39
6.3.3. Lw_inc 40
6.4. Tests varios 41
6.4.1. Lw_inc Miss - Read Hit 41
6.4.2. LW sobre un bloque recién expulsado 42

7. TEST DE INTEGRACIÓN **44**
7.1. Explicación 44
7.2. Speedup 45

8. CUANTIFICACIÓN DE HORAS DEDICADAS **46**

9. EVALUACIÓN INDIVIDUAL **47**

10. AGRADECIMIENTOS **48**

11. ANEXO 1. PROGRAMAS DE PRUEBA PARA TEST UNITARIOS **49**
11.1. TestMD 49
11.2. TestMD_Scratch 51
11.3. Test_Errores 52
11.4. Test Integrado 55

12. ANEXO 2. APARTADO OPCIONAL : WRITE BUFFERING **58**
12.1. Explicación 58
12.2. Pruebas 62
12.3. Casos de utilidad 64

13. ANEXO 3. APARTADO OPCIONAL : ETHICAL HACKING **66**
13.1. Preludio: Acerca del planteamiento para el apartado en la sección de incidencias 66
13.2. Descripción 66
13.3. Programa de *Ethical Hacking* 67
13.4. ¿Cómo paliar el problema? 72
13.4.1. Contramedidas llevadas a cabo por el equipo 73
13.4.2. Resultados 80

1. Resumen

La presente memoria recoge el trabajo del segundo proyecto de la asignatura, consistente en el **diseño de un controlador de memoria cache conectado a un procesador y a un bus semi-síncrono**. Para ello, se ha elaborado un **grafo de estados** desde el punto de vista de los diferentes tipos de transferencias que se pueden producir: lectura/escritura de bloque/palabra. Además, se han tenido en cuenta las particularidades de la instrucción *lw_inc*, la cual solo interactuará con la MC para invalidar bloques. El grafo de estados viene acompañado de una tabla de comportamiento de estado y de transiciones, donde se explorarán los diferentes situaciones de ejecución en la arquitectura con la que se trabaja.

Tras la elaboración del autómata, se realizarán una serie de estudios sobre el sistema implementado, como la **descomposición de la dirección**, lo cual es muy útil en la fase de *testing*. Además, se extraeran las **latencias de las diferentes transferencias de bus** tanto empírica (programas) como matemáticamente (análisis del código de las componentes). La obtención de las latencias será fundamental para el posterior cálculo de los **ciclos efectivos**, tomando como medida el sumatorio de los ciclos que toman los diferentes eventos que provocan accesos a memoria por la tasa respecto al total de referencias con la que se producen.

Por último, el equipo puso a prueba el correcto funcionamiento de la cache a través de una fase adicional de **testing**. Por una parte, se elaboró una tabla de casos unitarios, con diferentes comportamientos específicos con los que poner a prueba la cache. Por otra, se elaboró un test de integración, un programa que realice operaciones con matrices en un flujo normal de uso para poder compararlo posteriormente con un sistema sin MC (sin vías) y calcular el speedup.

Adicionalmente, el equipo trabajó en la resolución de **2 apartados opcionales**. En 12 se puede encontrar la extensión de una ***basic look-up free cache***, que introduce un buffer en escritura para poder aceptar operaciones que no requieran del bus mientras se realice la escritura en MD. En 13 se puede encontrar la elaboración de un programa de **hacking ético**, que satura la cache con un número excesivo de reemplazamientos, degradando el *performance* del programa. En este mismo apartado se propone una solución para el mismo, con un mecanismo para inhibir/desinhibir manteniendo el sistema de tags para la toma de métricas. **Ambas extensiones se encuentran en fuentes diferentes adicionales**, para no colisionar con los requisitos funcionales de la exigencias obligatorias ni entre ellos (pues su puesta en marcha fue en paralelo).

NOTA: El apartado opcional *hacking ético* se elaboró siguiendo una idea parecida a la del guión proporcionado, pero trabajando con un número excesivo

de *misses* en vez de reemplazamientos sucios. Esto se debe a una omisión de la lectura del apartado *hacking ético* en el apartado de *incidencias* de la asignaturas por confusión. Esto se comenta también en el correspondiente anexo.

2. GRAFO DE ESTADOS DE LA UNIDAD DE CONTROL

2.1. Descripción

En el presente apartado se presentará el núcleo del proyecto, puesto que el grafo de estados define el comportamiento del controlador de la *Memoria Cache* (no es un grafo de estados de bloque, sino del controlador mismo), y el objetivo del presente proyecto es proporcionar una cache que se comporte correctamente.

Se presenta la especificación de dos diagramas de estados: uno para el **controlador principal de la cache**, que además acciona a otra **máquina de estados que gestiona sencillamente las situaciones de error** en la memoria cache ya implementada.

Para no saturar con notación, en el grafo tan solo se incluyen las **etiquetas** para cada transición. Además, se adjunta una tabla con el comportamiento fijo que presenta el controlador en un estado dado (**Moore**), así como una tabla con las condiciones que disparan las transiciones y las señales que se activan (**Mealy**). Nótese que no puede haber inconsistencias entre el comportamiento de estado con el de una posible transición (esto debería definir una nueva transición).

2.2. Grafo de estados para el controlador principal

A continuación, se presenta el autómata (mealy) para el controlador principal de la caché. Este se compone de *6 estados*, y los arcos están etiquetados como T_i. Las señales de entrada del autómata son: **state, error_state, RE, WE, Fetch_inc, unaligned, internal_addr, Bus_grant, hit, addr_non_cacheable, Bus_DevSel, hit0, hit1, bus_TRDY, via_2_rpl y last_word_block**. Para más información, consultar código fuente.

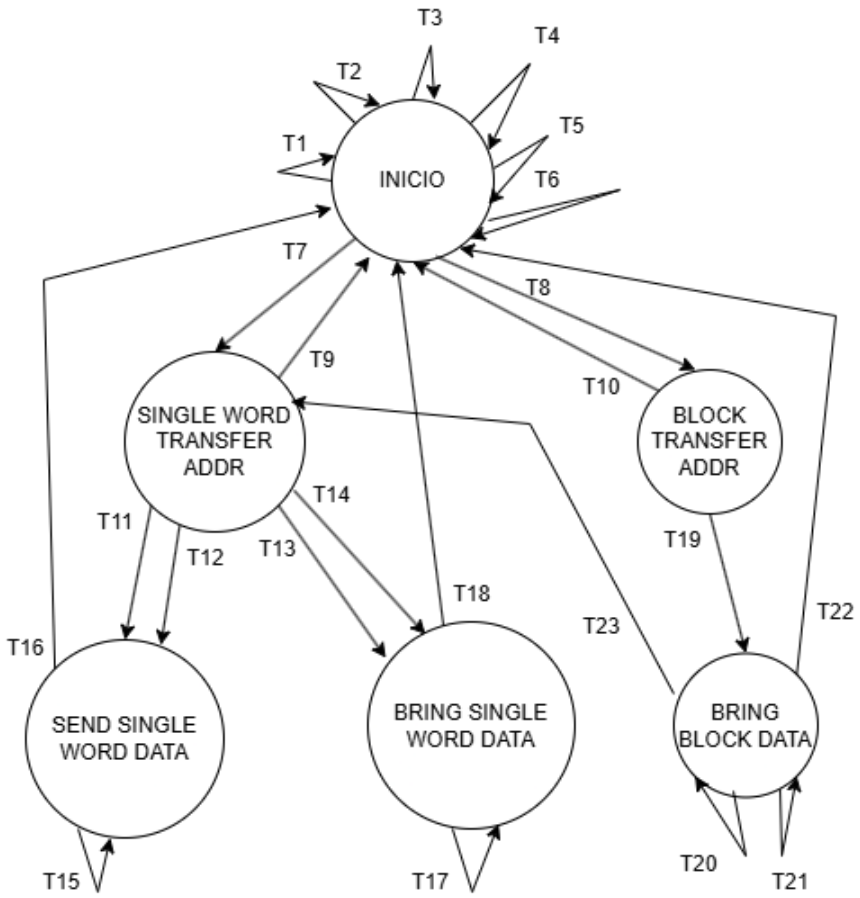

Figura 1: GRAFO DE ESTADOS PARA EL AUTÓMATA DE LA UC de la MC.

Como se puede observar, el diseño de la máquina de estados viene dirigido por dos aspectos en el flujo de un acceso a memoria: la **fase de la transferencia bus** (si la hay) y el **tipo de elemento** (palabra/bloque) que se transfiere.

En muchos casos, las diferencias de una transferencia (si es un WE a scratch o un WE con *Writethrough* a MD) vienen dada por la secuencia de transiciones disparadas. Las decisiones conceptuales serán esclarecidas posteriormente.

2.2.1. Tabla de comportamiento de estado (Moore)

La siguiente tabla determina las señales que se activarán cuando la máquina de estados se encuentre en un estado dado, **siempre e independientemente** de las transiciones disparadas y las señales de entrada. Esto permite representar algunos comportamientos genéricos de cualquier transferencia de bus, como el levantamiento de la señal *Frame* en toda fase de una transferencia bus.

ESTADO	SEÑALES ACTIVADAS
INICIO	NINGUNA
SINGLE WORD TRANSFER ADDR	Frame, MC_send_addr_ctrl; If (RE OR (Fetch_Inc and addr_non_cacheable) then MC_bus_Read else if(WB) then MC_bus_Write else MC_bus_Fetch_inc
BLOCK TRANSFER ADDR	Frame, MC_send_addr_ctrl, MC_bus_Read, block_addr
SEND SINGLE WORD DATA	Frame, MC_send_data, Last_word
BRING SINGLE WORD DATA	Frame, Last_word
BRING BLOCK DATA	Frame, mux_origen, Last_word = Last_word_block

Cuadro 1: Tabla de señales de comportamiento de estado

Véase que el autómata de la caché es **Mealy**. Por ejemplo, en el estado *SINGLE WORD TRANSFER ADDR*, el valor que toma cada señal de control de tipo de operación que se propagará por el bus depende del tipo de instrucción del MIPS que provocó el acceso a memoria.

2.2.2. Tabla de transiciones (Mealy)

La tabla de transiciones refleja el **comportamiento dinámico y variable** del sistema de memoria debido a distintos tipos de accesos y en diferentes situaciones. La etiqueta de las transiciones tiene un mapeo directo en el diagrama presentado. En cuanto a las condiciones (valores de señales de entrada), y considerando que solo puede dispararse una transición por estado, se resolverán ambiguedades por orden de aparición y especificidad. A continuación, se adjunta la tabla de transiciones del controlador:

TRANSICIÓN	CONDICIONES	SEÑALES ACTIVADAS
T1	RE = '0'AND WE = '0' AND Fetch_inc= '0'	ready
T2	RE = '1' AND internal_addr = '1'	ready, mux_output = '10', next_error_state = no_error
T3	(RE = '1' OR WE = '1' OR Fetch_inc = '1') AND unaligned = '1'	ready, load_addr_error, next_error_state = memory_error
T4	(WE = '1' OR Fetch_inc = '1') AND internal_addr = '1'	ready, load_addr_error, next_error_state = memory_error
T5	RE = '1' AND hit = '1'	ready, inc_r
T6	(addr_non_cacheable = '0' and ((RE = '1' AND hit = '0') OR Fetch_inc = '1' OR WE)) OR (addr_non_cacheable = '1' AND (RE = '1' OR WE = '1' OR Fetch_inc = '1')) AND Bus_grant = '0'	Bus_req
T7	(addr_non_cacheable = '0' and (Fetch_inc = '1' OR (WE = '1' AND hit = '1'))) OR (addr_non_cacheable = '1' AND (RE = '1' OR WE = '1' OR Fetch_inc = '1')) AND Bus_grant = '1'	Bus_req
T8	addr_non_cacheable = '0' AND ((RE = '1' OR WE = '1') AND hit = '0') AND Bus_grant = '1'	Bus_req
T9	Bus_DevSel = '0'	ready, load_addr_error, next_error_state = memory_error
T10	Bus_DevSel = '0'	ready, load_addr_error, next_error_state = memory_error
T11	Bus_DevSel = '1' AND WE = '1' AND addr_non_cacheable = '1'	NINGUNA
T12	Bus_DevSel = '1' AND WE = '1' AND hit = '1' AND addr_non_cacheable = '0'	inc_w, MC_WE0 = hit0, MC_WE1 = hit1
T13	Bus_DevSel = '1' AND ((addr_non_cacheable = '1' AND (RE = '1' OR Fetch_inc = '1')) OR (addr_non_cacheable = '0' AND Fetch_inc = '1' AND hit = '0'))	NINGUNA
T14	Bus_DevSel = '1' AND Fetch_inc = '1' AND hit = '1' AND addr_non_cacheable = '0'	inc_inv, invalidate_bit
T15	Bus_TRDY = '0'	NINGUNA
T16	Bus_TRDY = '1'	ready
T17	Bus_TRDY = '0'	NINGUNA
T18	Bus_TRDY = '1'	ready, mux_output = '01'
T19	Bus_Devsel = '1'	inc_m
T20	Bus_TRDY = '0'	NINGUNA
T21	Bus_TRDY = '1' AND last_word_block = '0'	inc_w, count_enable, mux_origen = '1'; if(via_2_rpl = '1') then MC_WE1 else MC_WE0
T22	Bus_TRDY = '1' AND last_word_block = '1' AND RE = '1'	inc_w, count_enable, mux_origen = '1', MC_tags_WE; if(via_2_rpl = '1') then MC_WE1 else MC_WE0
T23	Bus_TRDY = '1' AND last_word_block = '1' AND WE = '1'	inc_w, count_enable, mux_origen = '1', MC_tags_WE ; if(via_2_rpl = '1') then MC_WE1 else MC_WE0

Cuadro 2: Tabla de transiciones para la máquina de estados correspondiente al controlapor de la MC

- **NOTA:** Como se puede observar en las transiciones *T6*, *T7*, *T8*, se ha destacado la señal *Bus_grant* respecto al resto. Esto es así para destacar un aspecto importante de la fase de arbitración, y es que **la respuesta del árbitro se resuelve combinacionalmente**. De este modo, siempre que se cumpla una de las condiciones que desencadenen una transferencia en bus (un acceso a *MD Scratch*, un *miss* en *MC*, etc.), se levantará la señal *Bus_req* y en el mismo ciclo se comprobará si la señal *Bus_grant* ha sido activada. Lo único que cambia de manera secuencial es la prioridad, y solo sucede cuando una transferencia finaliza. De este modo, esperar un ciclo para tomar el control no perjudicaría el resultado de la transferencia, pero sí que redundaría en una latencia peor (mayor incluso de un ciclo si la prioridad pertence al *IO* y envía una petición justo al siguiente ciclo).

  En *vhd*, este comportamiento se resuelve fácilmente teniendo en cuenta que las **sentencias en procesos son secuenciales**:

```
1
2   elsif ((((RE = '1' and hit = '0') or (Fetch_inc = '1')
        or (WE = '1')) AND addr_non_cacheable = '0') OR
       (((RE = '1') or (Fetch_inc = '1') or (WE = '1'))
       and addr_non_cacheable = '1')) then -- si
       involucra alguna operaci n que tenga que hacer
       uso del bus
3     Bus_req <= '1';
4     if(Bus_grant = '1') then -- Bus_grant puede ser
          levantado en el mismo ciclo
5       if ( (RE = '1' or WE = '1') and hit = '0' and
            addr_non_cacheable = '0'  ) then -- CASO de
            miss: Se debe traer un BLOQUE (lw_inc tiene
            tratamiento especial, de ah  que no se
            incluya)
6         next_state <= block_transfer_addr;
7       else  -- En otro caso, ser  una tranferencia de
            palabra, bien sea con MD_scratch o con MD (y
            lectura o escritura)
8         next_state <= single_word_transfer_addr;
9       end if;
10    else -- Si el  rbitro  no ha concedido permiso para
          emplear el bus, se sigue esperando a que lo haga
           (n tese que seguir  cumpliendo la misma
          cl usula inicial al permanecer mem_ready bajado
          )
11      next_state <= Inicio;
12    end if;
```

Listing 1: COMPLETAR_UC_MC_2025_WT_ciclo_arb.vhd

2.2.3. Decisiones de diseño

En la sección que acontece, se pretenden exponer lo más exhaustiva y concisamente posible las razones que llevaron al diseño peresentado, tanto para demostrar su corrección general como los puntos más intrincados en el universo del problema. Esta tarea se abordará explicando el significado conceptual de cada estado del autómata, así como las señales más determinantes en su comportamiento inherente y las transiciones que se disparan partiendo desde él:

1. **INICIO**: Este estado representa el estado en el cual la memoria cache '*vive*' sin necesidad de interaccionar con el sistema de bus multiplexado. **Es el origen de toda operación**, bien conlleve una transferencia de bus o no (de ahí que no contenga señales levantadas por defecto), por lo que puede desembocar en diversas transiciones:

   a.) **T1**: Corresponde al caso de que la instrucción en la etapa *MEM* del MIPS no acceda a memoria, importante tenerla en cuenta para no obstruir abruptamente al procesador.

   b.) **T2**: Corresponde al caso de intentar **leer de un registro interno** de la *MC* (en el presente caso, solo puede ser el registro *ADDR_Error_Reg*). En este caso, es importante que **mux_output = '10'**, para enviar correctamente la salida del registro interno.

   c.) **T3**: Caso de intentar acceder a una dirección desalineada (no múltiplo de 4). En este punto, se debe destacar que **en toda situación de error**, debe activarse indistinamente la señal *ready*, pues en caso contrario el procesador no podrá gestionar la situación de error (recuérdese del anterior proyecto que las excepciones no se gestionan si el procesador está parado, por lo que no se saltaría en ningún caso a la rutina de servicio dedicada a *Data Abort*)

   d.) **T4**: Caso de intentar escribir en un registro interno. Cuando se tenga una instrucción ***lw_inc*** que intente acceder a un registro interno se mantendrá su semántica completa (al contrario que con el acceso a MD_Scratch), por lo que constituye también un acceso erróneo al conllevar un postincremento del registro.

   e.) **T5**: Caso de **lectura sobre memoria cache debido a un *hit***.

   f.) **T6, T7 Y T8**: Caso en el que **una instrucción precise de acceso al bus**. Como se ha mencionado anteriormente, se activa la señal **Bus_req** y comprueba en el mismo ciclo si el árbitro concede permiso para emplear el bus (*Bus_grant*). En caso de obtener permiso del árbitro para emplear al bus, como las líneas para datos y dirección están multiplexadas, se salta a un estado previo de **control y dirección**. En este punto, se diferencia si la transferencia será de **palabra** (acceso a MD_Scratch, caso de la política *Write-through*, lw_inc sobre MD) o de **bloque** (*sw* o *lw* con miss sobre caché).

Llegados a este punto, resulta de extrema importancia explicar la **POLÍTICA DE LW_INC** seguida, tanto con respecto a MD_Scratch como con MD. Para el caso de la **MD_scratch**, la instrucción **lw_inc** se tratará del mismo modo que la instrucción lw. Y es que esta componente de memoria de acceso rápido ni siquiera dispone de una entrada para la señal. Por otra parte, para el caso de la ***MD***, se ha decidido '*ignorar*' las políticas de cache (pero no el efecto sobre la cache misma). De este modo, cuando una instrucción *lw_inc* quiera acceder sobre una dirección perteneciente a la *MD*, la petición será propagada (pero no capturada) directamente a la *MD*. En caso de que la palabra requerida se encuentre la MC (que se produzca un *hit*), será estrictamente necesario invalidar su bloque, para así mantener coherencia entre la *MC* y la *MD* (recuérdese que con *Write-through*, MC = MP). Además, nótese que simplemente no tiene sentido gestionar el lw_inc siguiendo las políticas de la MC (por ejemplo, en caso de *miss*, cargando el bloque de *MD*, para posteriormente invalidarlo y acceder a MD), siempre redundará en un incremento de la latencia completamente innecesario.

2. **SEND_SINGLE_WORD_ADDR**: Esta etapa corresponde a la **fase *Adress* en la transferencia de bus para el caso de escritura/lectura de palabra**. **En general**, todas las fases de escritura/lectura de una palabra se pueden condensar en un único estado pese a la heterogeneidad de estas. Y esto se puede llevar a cabo sin añadir una gran sobrecarga lógica (*Mealy*) por varias razones:

   - En primer lugar, las instrucciones que requieran del **acceso a la memoria cache** (por ejemplo un *lw_inc* que requiera invalidar el bloque la palabra a la que pretende acceder) pueden realizar acciones individuales en un solo ciclo. De este modo, este tipo de comportamientos específicos pueden incorporarse a nivel de transición (y una sola vez al solo implicar a una palabra).
   - Por otra parte, los accesos a *MD* y *MD_Scratch* (diferenciados por la señal de entrada *addr_non_cacheable*) requieren un tratamiento diferenciado, ya que en el segundo caso las direcciones son no cacheables. Ahora bien, excluyendo este aspecto (gestionado con múltiples transiciones), el **proceso de transferencia en el bus** se aborda idénticamente por el controlador en ambos casos. Esto es así porque el proceso de transferencia está determinado por el protocolo del **bus**, siendo en este sentido los dispositivos "transparentes". Véase, por ejemplo, que cuando se precise enviar al *MIPS* un dato procedente tanto de la *MD* o *MD_Scratch*, se debe establecer la señal *mux_output = '01'*; se debe enviar la salida del bus.

   Es de vital importancia levantar a partir de este ciclo la señal **Frame**, pues es en este momento cuando comienza la transferencia (y es la línea del bus *Frame* la que principalente inhibe al árbitro). Además, debe indicarse el

uso concreto del **bus multiplexado** (señal *MC_send_addr_ctrl*), así como el tipo de operación a realizar (tanto escritura/lectura como palabra/bloque). Nótese que es necesario únicamente en esta etapa, pues las componentes de memoria están configuradas para almacenar en registros internos las señales de control cuando procede (como el caso del registro *Read_Write_register* en el componente MD_cont_2025.vhd). Las **transiciones** disparables desde estado son las siguientes:

a.) **T9**: Caso correspondiente a **en el que ningún dispositivo reconoce la dirección como perteneciente a su espacio de direcciones**. Es importante remarcar que el reconocimiento de direcciones se realiza en el **mismo ciclo en el que se envía la dirección**. Lo contrario generaría problemas en el caso del *MD_Scratch*, pues el dispositivo baja la señal al siguiente ciclo, dando lugara a un falso 'error' (el mismo problema se ve ocultado en *MD* debido a que incorpora un estado intermedio *DETECTADO*, donde además se vuelve a activar *MC_bus_DEVsel*).

b.) **T11, T12**: Corresponde a los casos en los que se **envia un dato desde el MIPS hasta la *MD* o *MD_Scratch***. En el caso de que el destino sea la *MD*, se debe escribir en la vía de la cache correspondiente (dada por la señal *hit0* o *hit1*, teniendo en cuenta de que nunca podrán encontrarse activas ambas), y la palabra que además se enviará a memoria, como parte de la política *Write-Through*.

c.) **T13, T14**: Corresponde a los casos en los que **el MIPS recibe un dato desde la *MD* o *MD_Scratch*** (para el caso de la MD, solo se incluye el caso del *lw_inc*, donde tan solo se propaga la palabra al MIPS). Se debe considerar el caso del *lw_inc* donde se produce un *hit* en una de la vías de la cache (*T14*). Aquí, antes de pasar a la fase de transferencia de datos se activará la señal **invalidate_bit** para mantener la coherencia entre MC y MD.

3. **SEND SINGLE WORD DATA, BRING SINGLE WORD DATA**: Corresponde a la **fase de datos en en la escritura o lectura de una sola palabra en/de *MD* o *MD_Scratch***. En estos estados es, importante activar **siempre** la señal **Frame** (incluso en la última fase), pues el *esclavo* retirara inmediatamente los datos del bus cuando esta se baje. Además, para el caso de escritura de una palabra se debe **levantar la señal *MC_send_data***. Por último, se debe mencionar que la señal *mux_origen* puede permanecer a '0' sin riesgo alguno (pues la única palabra de la transferencia coincide con la solicitada por el *MIPS*), mientras que es necesario **mantener siempre levantada la señal *Last_word***. Esto se debe a que la cache activa la señal *Last_word* en el momento de enviar o solicitar la última (única) palabra, no cuando la palabra se escriba o reciba (cuando *bus_TRDY* = '1'). Esto se realizó así para trabajar acorde al diseño conceptual de la *MD*, que vuelve al estado de *Espera* una vez termine su trabajo (aunque tampoco pueda la arquitectura actual

aprovecharse de esta particularidad, al estar conectada la *MD* a un unico bus y ser este liberado cuando termine la transferencia). En cuanto a las **transiciones** que se disparan desde estos estados:

a.) **T15, T17**: Corresponden a casos **en los que el *target* no puede realizar la operación en el presente ciclo**. Importante tenerlos en cuenta, pues es seguro que se produzcan en prácticamente todas las operaciones (los dos ciclos del *lw_inc*, los retardos arbitrarios de 2-3 ciclos en MD_Scratch, etc).

b.) **T16, T18**: Corresponden al caso en el que **la única palabra ha sido efectivamente transferida**. Como en este caso solo hay una operación que realizar, esto se producirá la primera vez que se reciba la señal *Bus_TRDY* activada. Además, es importante que en el caso de **lectura** de palabra ***mux_output = "01"***, puesto que la salida del bus siempre tiene como destinatario al MIPS (de hecho, exclusivamente a él, la *MC* solo empleará la salida del bus para traer bloques a MC).

4. **BLOCK TRANSFER ADDR**: Este caso corresponde a la **fase *Address* en una transferencia de bloque**. Nótese que se ha asignado este nombre por generalidad, pero en la arquitectura de trabajo el envío de un bloque a MD no será necesario mientras se mantenga la política *Write-Through*; de ahí que siempre se active la señal **MC_bus_Read**. Además, es de vital importancia levantar la señal ***block_Addr***, para que la *MC* formatee correctamente la dirección envíada al bus (la *MD* realmente funciona con independencia de la señal, pues lo que hace es incrementar un contador de palabras *cont_palabras* cada vez que efectúa una operación. No tiene la noción de bloque que sí existe en la cache, simplente cuando le llegue la señal *bus_last_word* sabrá que se ha llegado al final de la transferencia). En cuanto a las transiciones disparables desde este estado, se quiere mencionar que para el caso de dirección reconocida por la MD (**T19**), se debe activar **inc_m**, pues siempre que se transfiera un bloque de memoria será porque se ha producido un *miss*.

5. **BRING BLOCK DATA**: Estado correspondiente a la **fase *Data* en la transferencia de lectura de un bloque de *MD***. Para este estado, es de nuevo importante levantar **last_word** cuando se vaya a solicitar la última palabra, así como que **mux_origen = '1'** (pues ahora es el controlador de la *MC* el que que irá generando las consecutivas direcciones para las palabras de un bloque). Respecto a esto último, conviene destacar que el controlador posee un registro interno *word_counter*, el cual se encargará de generar la diferentes direcciones de palabras a lo largo de la transferencia; contador gobernado con la señal **count_enable**, la cual deberá ser activada **siempre** que se reciba una palabra. Desde este estado se pueden disparar 4 **transciones**:

a.) **T20**: Representa el caso en el que **la *MD* no puede realizar la operación en el presente ciclo**.

b.) **T21**: Representa el caso en el que **se ha recibido una nueva palabra de la *MD*, pero no la última**. En este punto, se deja constancia de que se considera que *cont_w* considera operaciones de escritura sobre la MC, pero siempre a nivel de palabra (por lo que se activa para cada palabra recibida del bloque). Por otra parte, la escritura en una vía u otra se gestiona a través de **via_2_rpl**.

c.) **T22, T23**: Representa el caso en el que **se ha recibido la última palabra de un bloque**. Bien el reemplazamiento de un bloque se deba a un *miss* en escritura o lectura, es importante que **ready = '0'**, pues se ha finalizado la transferencia del bloque, pero no la operación que precisaba el *MIPS*. Además, es importante activar la señal ***MC_Tags_WE*** concretamente en esta fase. Esto se debe a que el componente de la MC *Info_FIFO* actualiza la vía a reemplazar cuando se actualizan los tags de un bloque, por lo que sobreescribir los tags al inicio de la transferencia llevaría a reemplazar en la vía opuesta.

   Por último, se debe mencionar que se sigue un camino diferente en función de si la instrucción que provocó el fallo fue de lectura o escritura. Así, para el caso de un ***sw miss***, no se retorna al estado inicial, sino que se transiciona al estado *SINGLE WORD TRANSFER ADDR*. Esto se realiza por dos razones:

   - Primero, porque el autómata que gobierna la *MD* no retorna a la estado de *Espera* solo cuando se baja *Bus_Frame*, sino cuando el *Master* le indica que la palabra involucrada es la última de la tarnsferencia. Esto permite que un mismo dispositivo realice dos transferencias consecutivas con la *MD* sin la de necesidad de volver a pasar por la fase de arbitración.
   - Cuando se sobreescriben los tags del bloque al final de la transferencia, es trivial afirmar que la operación solicitada por el MIPS supone un acierto en escritura.

3. DESCOMPOSICIÓN DE LA DIRECCIÓN

3.1. Descripción

La descomposición de la dirección es un proceso muy importante en el contexto del manejo de Memorias *Cache*, consistente en **dividir una dirección de memoria en diversos fragmentos para identificar subcomponentes lógicas de una memoria *Cache***. En concreto, una dirección de memoria se dividirá en tres campos:

1. **TAG**: Sirve para identificar si un bloque de datos (conjunto de palabras contiguas en memoria) se encuentra o no en la *MC*.
2. **SET**: Identifica el conjunto de la cache al que pertenece el bloque de la palabra. Es decir, la línea de la cache donde se debe situar el bloque en caso de escribirse en MC.
3. **OFFSET**: Indica el desplazamiento dentro de un bloque.

En este apartado, se describe la descomposición de la dirección. Aunque no afecte directamente a la comprensión del diseño del autómata de la cache, su lectura resulta importante a la hora de estudiar los *tests* proporcionados.

3.2. Descomposición

En primer lugar, es preciso tener en cuenta las características de la memoria Caché, proporcionada:

- La *MC* trabaja con **direcciones de 32 bits** (aunque realmente el rango de direcciones cacheables se encuentre entre '0x00000000' y '0x000001FF').
- La *MC* dispone de 128 bytes, es decir, 32 palabras (teniendo en cuenta que cada palabra tiene un tamaño de 4 *bytes*). Además, se trabaja con bloques de 4 palabras. Por tanto, se tiene que la memoria cache contiene en total **8 bloques de 4 palabras de 4 bytes**.
- Ahora bien, una propiedad importante de la MC es que es **2-asociativa**, de manera que contiene 2 vías, cada una con 4 bloques. De este modo, cada bloque de un conjunto puede ser asignado en el mismo conjunto de una vía u otra; a costa de disminuir el número de conjuntos a 4.

Teniendo en cuenta esto, discernir la descomposición de la dirección es trivial. Serán necesarios 4 bits para el campo *OFFSET* (2 para seleccionar byte dentro de palabra y otros 2 para seleccionar palabra dentro de bloque). Además, se necesitan otros 2 bits para el campo *SET*, dada las características de asociatividad por bloques de la cache. Los 26 bits restantes corresponden al campo *TAG* (identificar bloque dentro de conjunto). A continuación, se adjunta la descomposición:

$$|^{31}tttttttttttttttttttttttttt^{6}|^{5}ss^{4}|^{3}oooo^{0}|$$

Como se ha mencionado anteriormente, la componente *MC* porporcionada ya se encarga de descomponer adecuadamente la dirección envíada por el *MIPS*, así como reconocer si está dirección es cacheable:

```
1
2       addr_non_cacheable <= '1' when Addr(31 downto 8) = x"
            100000" else '0';
3       unaligned <= '1' when Addr(1 downto 0) /= "00" else
            '0';
4       tag <= ADDR(31 downto 6);
5       dir_word <= ADDR(3 downto 2) when (mux_origen='0') else
             palabra_UC;
6       dir_cjto <= ADDR(5 downto 4); -- associative placement
            (two-ways)
```

Listing 2: MC_DATOS_2025_WT.vhd

4. ANÁLISIS DE LATENCIA EN TRANSFERENCIAS DE BUS

4.1. Descripción

En el presente apartado, se realizará el estudio de la **latencia para los difrentes tipos de transferencia de bus** con lo que trabaja la Memoria Cache. Lo que se estudian son las transferencias en sí mismas, por lo que resulta razonable excluir los tiempos variables derivados de la arbitración del estudio de las mismas. Además, se quiere comentar que la estimación de los ciclos consumidos por cada tipo de transferencia parte de un estudio de la descripción de componentes, pero en todo caso deberán ser verificadas y extraidas definitivamente a partir de la simulación.

4.2. Cálculo y simulación para obtener latencias

1. **CrB(MD)** (ciclos requeridos para leer un bloque de la *MD*): En este caso, la latencia se verá afectada por el hecho de cargar reiteradamente palabras, pero se deben tener en cuenta ciertas particularidades del controlador de la *MD*. Para la obtención del *CrB(MD)*, se deben tener en cuenta los siguientes puntos:

   - En la *MD* proporcionada, las operaciones escritura y lectura en memoria (sobre la *RAM*) siempre ocupan 1 ciclo. Tan solo el *lw_inc* lleva más de un ciclo (recuérdese que era esta la única instrucción capaz de provocar un *stall* del *MIPS*).
   - El controlador de la *MD* introduce siempre un estado intermedio (entre que detecta una transferencia y la comienza) ***Detectado***, donde informa de nuevo que ha detectado una dirección perteneciente a su espacio de memoria.
   - El controlador, además añade una lógica para **introducir retardos artificiales**. De este modo, cuando en el estado *Espera* se reconoce que la dirección recibida no coincide con la última (almacenada en el registro *reg_last_addr*), resetea un contador que no permitirá realizar una operación sobre memoria hasta pasados los 5 ciclos en la fase de *Transferencia*. Este retardo solo afectará a la primera palabra del bloque, pues para el resto la escritura podrá realizarse en un solo ciclo. Además, conviene mencionar que una vez termina la transferencia, el contador para introducir retardos se resetea, por lo que la siguiente operación (aunque la dirección sea la misma que la última) tendrá asimismo 5 ciclos de retardo.

   Teniendo en cuenta esto, se simularon dos casos: una lectura de bloque como primera instrucción del programa (no hay última dirección almacenada en *MD*) y un caso en el que se produce un *miss* después de que un *lw_inc* invalide el bloque de memoria por escribir la primera palabra del

mismo (único caso en el que la última dirección accedida en la *MD* puede ser igual a la primera palabra de una lectura de bloque de la *Md*). En ambos casos, se obtuvo la misma latencia (11 ciclos):

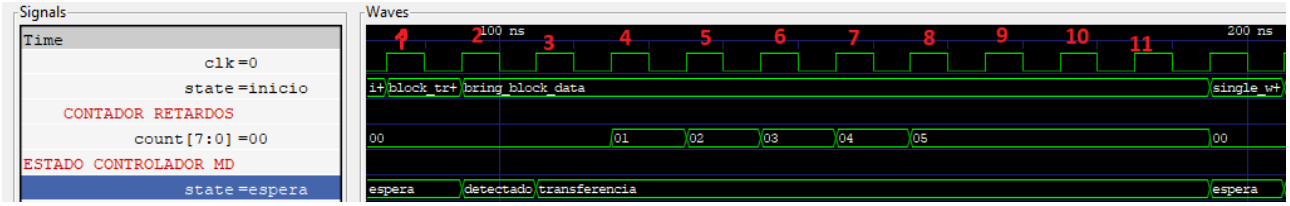

Figura 2: Latencia para una lectura de bloque de la MD

De este modo, se tiene que:

$$CrB(MD) = 1\ ciclo\ address + \ 1\ ciclo\ detectado\ + \ 5\ ciclos\ delay\ \ + \\ 4\ ciclos\ lectura\ palabras = 11\ ciclos$$

Considerando la fórmula $CrB(MD) = L + 3\ *\ R$, se tiene que $\mathbf{L = 8}$ y $\mathbf{R = 1}$.

2. **CwW(MD)** (ciclos requeridos para escribir una palabra a la *MD*): Para el caso de escribir una palabra, se debe tener en cuenta que la única palabra a escribir siempre conllevará el retardo inicial arbitrario. Se adjunta el caso de escritura de palabra despúes de un fallo en escritura, que como bien se puede observar arroja que **CwW(MD) = 8 ciclos**:

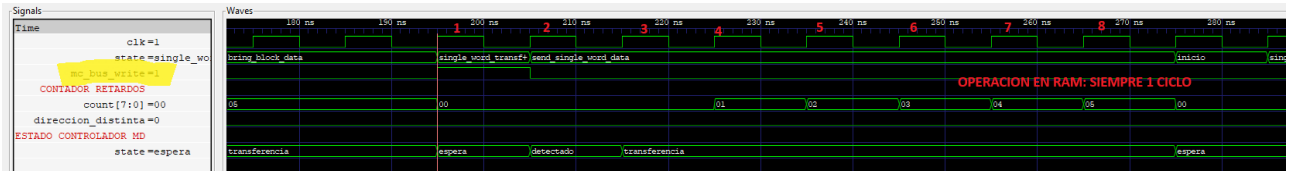

Figura 3: Latencia para una escritura de palabra sobre la MD

Así, se tiene que:

$$CwW(MD) = 1\ ciclo\ address + \ 1\ ciclo\ detectado\ + \ 5\ ciclos\ delay\ \ + \\ 1\ ciclo\ escritura = 8\ ciclos$$

3. **Clwi(MD)** (ciclos requeridos para realizar un *lwi* sobre la *MD*): La instrucción *lw_inc* presenta la particularidad de que conlleva una lógica interna (almacenar en registro intermedio e incrementar) en la MD que fuerza

a llevar la operación siempre en 2 ciclos. Más allá de ello, los resultados en las simulaciones se adecuan al comportamiento de las transferencias de bus y controlador de la *MD* ya mencionados. Se llevaron simualaciones escribiendo consecutivamente en el mismo registro y diferentes (además de tener en cuenta los casos en los que debe invalidar o no el bloque). En todos los casos, se obtuvo que **Clwi(MD) = 9 ciclos**. A continuación, se adjunta un fragmento de la simulación de dos *lw_inc* consecutivos, el segundo con dirección distinta de la última. Véase como la lógica subyacente a *direccion_distinta* no afecta a la latencia ya que al fin de toda transferencia se resetea el contador de ciclos.

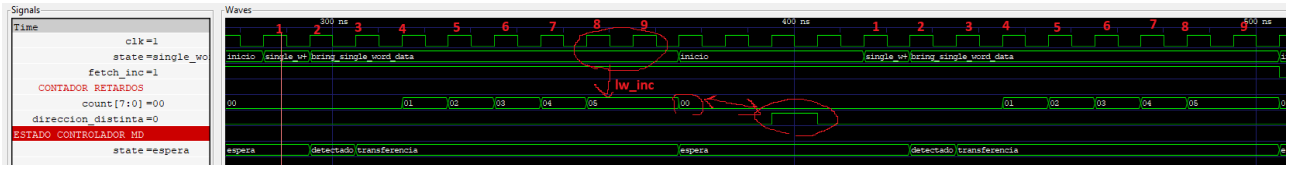

Figura 4: Latencia para una instrucción *lw_inc* sobre la MD

Así, se tiene que:

$$Clwi(MD) = 1\ ciclo\ address + \ 1\ ciclo\ detectado\ +\ 5\ ciclos\ delay\ \ + \\ 2\ ciclos\ lectura - escritura = 9\ ciclos$$

4. **CrW(MDscratch)** (ciclos requeridos para leer una palabra de la *MD_Scratch*): En cuanto a la *MD_Scratch*, se debe mencionar que trabaja con una *RAM* donde las escrituras y lecturas tardan siempre 1 ciclo. Ahora bien, la máquina de estados del controlador de la *MD_Scratch* introduce un estado adicional ***Envio***, de manera que cuando se produzca el acceso a la *MD_Scratch*, se cargará en un registro el dato solicitado; pero este no será enviado a la *MC* hasta pasado un ciclo. A partir de la simulación, se obtuvo que **CrW(MDsratch) = 3 ciclos**, tal como se muestra en la siguiente figura:

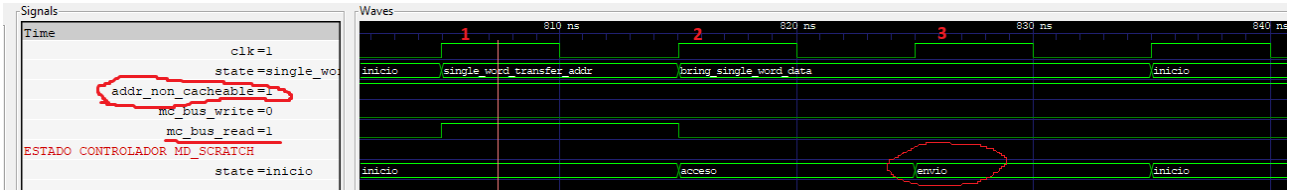

Figura 5: Latencia para una lectura de palabra sobre la MDscratch

Así, se tiene que:

$$CrW(MDscratch) = 1\ ciclo\ address + \ 1\ ciclo\ acceso\ + 1\ ciclo\ envio = 3\ ciclos$$

5. **CwW(MDscratch)** (ciclos requeridos para escribir una palabra en la *MD_Scratch*): El caso de escritura de una palabra sobre la *MD_Scratch* es más simple, ya que en este caso la escritura se produce en la misma fase de *Acceso* a la componente, tal como se muestra a continuación:

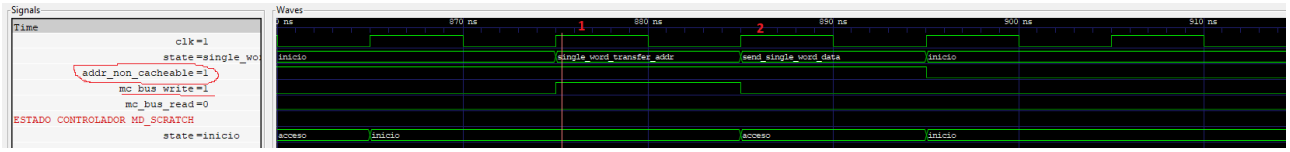

Figura 6: Latencia para una escritura de palabra sobre la MDscratch

Así, se tiene que:

$$CwW(MDscratch) = 1\ ciclo\ address + \ 1\ ciclo\ acceso\ = 2\ ciclos$$

6. **CrW o CwW (IO_REG)** (ciclos requeridos para leer o escribir una palabra en registros *I/O*): Para la escritura o lectura en registros de entrada y salida solo es necesario **1 ciclo**, puesto que se interacciona con el subsistema IO_MD directamente:

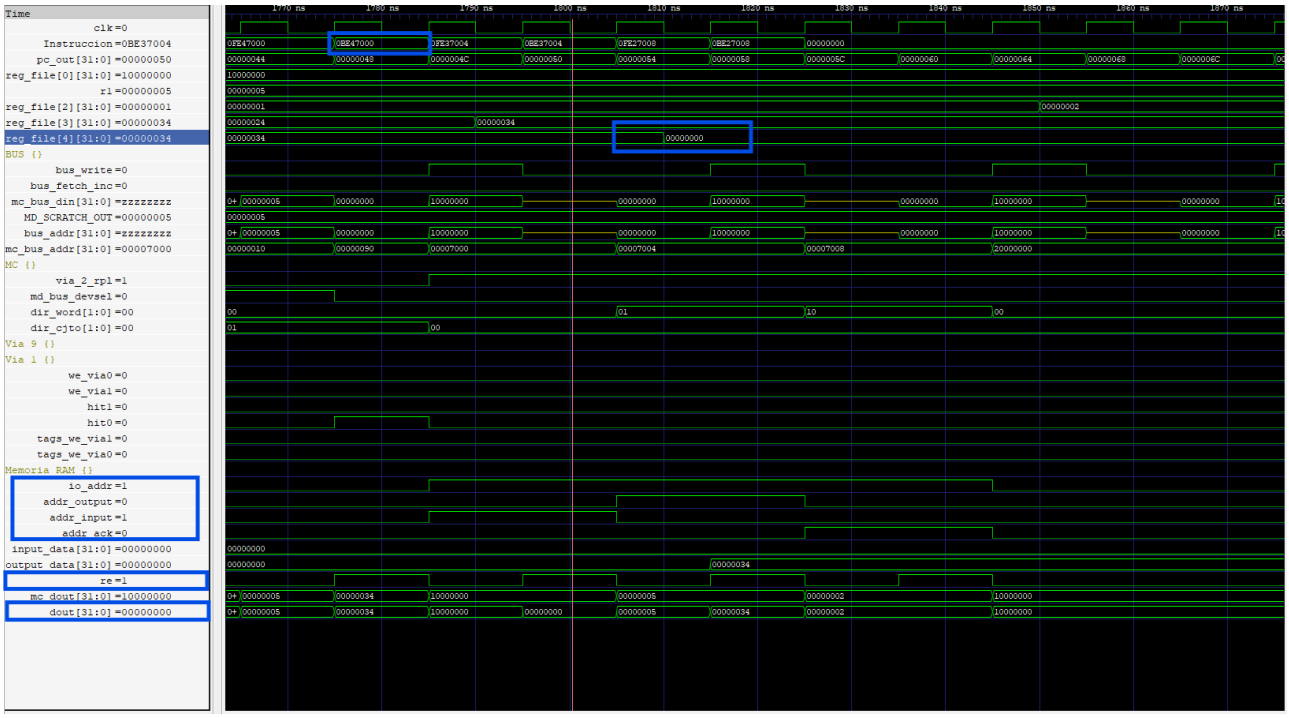

Figura 7: Caso de LW sobre Input Reg

,

5. FORMULACIÓN MATEMÁTICA DE LOS CICLOS EFECTIVOS

5.1. Descripción

La latencias calculadas en el apartado anterior servirán de base para calcular los **ciclos efectivos por acceso a memoria**, es decir, el promedio de ciclos de reloj que se necesitará para completar un *acceso completo* a memoria. Ahora bien, hay diversos *eventos* que pueden provocar un acceso a memoria: tanto escrituras/lecturas que involucren a las vías de la *MC* o sus registros internos, como transferencias que impliquen el bus (a *MD*, *MD_Scratch* o *I/O registers*). En este apartado, se presentará el razonamiento llevado a cabo para llegar a una formulación de ***Ceff***.

5.2. Fórmula general

Como se ha mencionado en la introducción a la sección, el cálculo de los ciclos efectivos debe calcularse como un promedio, y esta sujeto a los diversos eventos que pueden provocar accesos a memoria. Por ello, se presenta útil definir *Ceff* como:

$$C_{eff} = \frac{\sum ciclos\_evento_i * refs\_evento_i}{refs\_mem}$$

donde $ciclos\_evento_i$ hace referencia al número de ciclos que supone un acceso a memoria generado por el evento i (las latencias del apartado anterior). Nótese que $\sum refs\_evento_i = refs\_mem$.

Con esto, lo que se debe realizar ahora es desengranar los diferentes eventos que se pueden producir en la *MC*. En este punto, no se considera el *overhead* por arbitraje en transferencias de bus (se hará posteriormente):

1. **Escrituras/Lecturas sobre direcciones de memoria cacheables (MD)**:
   Estos son los accesos dentro del espacio de direcciones con el que trabaja la *MC*, que en el presente caso siempre siempre corresponden a la *MD*. Debe destacarse que en este punto no se consideran accesos a memoria debidos a lw_inc, pues estos puden provocar la invalidación de un bloque, pero en ningún caso la escritura/lectura sobre *MC*. Además, es necesario tener en cuenta que los eventos producidos dependerán de si se produce un *hit* o *miss*. De este modo, hay cuatro casos a considerar:

   a) *hit* en lectura: Tan solo requiere **1 ciclo**, puesto que no se producirá ninguna transferencia con la *MD*.

b) *hit* en escritura: En este caso, debido a la política *Write-Through*, además de escribir la palabra en la *MC*, deberá asimismo escribir la palabra en la *MD*. Esto requiere de un número de ciclos igual a **CwW(MD) = 8 ciclos**.

c) *miss* en lectura: Este caso requiere leer un bloque de la *MD*, para que posteriormente se produzca un *hit* y se lea la palabra de la *MC*. Esto supone un total de **CrB(MD) + 1 = 12 ciclos**.

d) *miss* en escritura: Un fallo en escritura implica la lectura del bloque correspondiente de la *MD*, para posteriormente escribir la palabra en la *MC y MD* siguiendo la política *Write-Through*. Esto conlleva un total de **CrB(MD) + CwW(MD) = 19 ciclos**.

2. **Accesos debidos a *lw_inc* efectivos**: Se menciona el término *efectivos* debido a que la *MD_Scratch* no soporta esta instrucción (se incluyen como caso de escritura sobre *MD_Scratch*), por lo que una lw_inc como tal solo se ejecutará efectivamente sobre *MD*. Nótese que tanto si se debe invalidar el bloque en *MC* como si no, el *lw_inc* solo supone la escritura de una palabra sobre *MD*, tal como se calculó en el apartado anterior. Esto supone un total de **Clwi(MD) = 9 ciclos**.

3. **Escrituras/Lecturas sobre la *MD_Scratch***: En este caso, al no interaccionar con la *MC* como tal, el número de ciclos requerido corresponde a **CrW(MD_Scratch) = 3 cilos** y **CwW(MD_Scratch) = 2 ciclos**, respectivamente.

4. **Accesos a registros internos de la *MC***: Tanto si se produce un intento de lectura como de escritura (en este caso, tan solo se generará un error), el acceso durará **1 ciclo**.

Con ello, y teniendo en cuenta el *delay* debido al arbitraje de **1,5 ciclos**, se obtiene la siguiente fórmula:

$$C_{eff} = \frac{C\_evento\_dirCacheable * refs\_Cache + C\_lw\_inc * refs\_lw\_inc}{refs\_mem}$$

$$\frac{+C\_evento\_MDscratch * refs\_MDscratch + C\_evento\_regInt * refs\_regInt}{refs\_mem} =$$

Desarrollando:

$$C\_evento\_dirCacheable = whc_{rate} \cdot C_{whc} + rhc_{rate} \cdot C_{rhc} + wmc_{rate} \cdot C_{wmc} + rmc_{rate} \cdot C_{rmc} =$$

$$= (delayArb + CwW(MD)) \cdot whc_{rate} + 1 \cdot rhc_{rate} + (delayArb + CrB(MD) + CwW(MD)) \cdot wmc_{rate}$$

$$+ (\text{delayArb} + \text{CrB(MD)} + 1) \cdot rmc_{rate}$$

$$= delayArb * (mc_{rate} + whc_{rate}) + CwW(MD) * (wc_{rate}) + CrB(MD) * (mc_{rate}) + (rc_{rate});$$

donde $\text{Xc}_{rate} = \frac{refs\_eventoCache\_X}{refs\_cache}$

$$C\_lw\_inc = delayArb + Clwi(MD)$$

$$C\_evento\_MDscratch = C_r \cdot rs_{rate} + C_w \cdot ws_{rate} =$$

$$delayArb * 1 + CrW(MDscratch) \cdot rs_{rate} + CwW(MDscratch) \cdot ws_{rate};$$

donde $\text{Xs}_{rate} = \frac{refs\_eventoSratch\_X}{refs\_MDscratch}$

De este modo, volviendo a la fórmula para **Ceff**:

$$C_{eff} = \frac{delayArb * (mc + whc + refs\_lw\_inc + refs\_MDscratch) + CwW(MD) * (wc) +}{refs\_mem}$$

$$\frac{+CrB(MD) * (mc) + (rc) + Clwi(MD) * refs\_lw\_inc + 1 \cdot \_regInt +}{refs\_mem}$$

$$\frac{+CrW(MDscratch) \cdot rs + CwW(MDscratch) \cdot ws}{refs\_mem}$$

Finalmente, sustituyendo por los valores de las latencias conocidos, se obtiene:

$$C_{eff} = \frac{1{,}5 * (mc + whc + refs\_lw\_inc + refs\_MDscratch) + 8 * (wc) +}{refs\_mem}$$

$$\frac{+11 * (mc) + (rc) + 9 \cdot refs\_lw\_inc + 1 \cdot refs\_regInt + 3 \cdot rs + 2 \cdot ws}{refs\_mem}$$

donde,

- mc : Número de misses sobre direcciones cacheables.
- whc : Número de write hits sobre direcciones cacheables.
- wc : Número de writes sobre direcciones cacheables.
- rc : Número de reads sobre direcciones cacheables.
- rs : Número de reads sobre MD_scratch.
- ws : Número de reads sobre MD_Scratch.
- refs_X: Número de referencias a un componente de memoria (o memoria en general)

6. TEST UNITARIOS

6.1. Descripción

Una vez implementado el autómata del controlador de la *MC*, se procedió a definir **casos de prueba unitarios**, que pusiesen a prueba comportamientos específicos de la MC. Para ello, se planteó exhaustivamente la ejecución de diferentes operaciones en los diferentes contextos donde se pueden dar, lo que dió lugar a las siguiente **tabla de requisitos unitarios** y *tests* donde se pueden validar:

CASO DE PRUEBA	UNIT TEST MD	UNIT TEST MD_SCRATCH	UNIT TEST ERRORES
operación sin interacción con sistema de memoria	**X**		
Read hit sencillo en Vía 0	**X**		
Read hit tras read miss en Vía 0	**X**		
Read hit tras write miss en Vía 0	**X**		
Read hit tras un lw_inc sobre bloque no cargado en memoria (no invalida) en Via 0	**X**		
Read hit sencillo en Vía 1	**X**		
Read Hit tras un read miss en Via 1	**X**		
Read Hit tras un write miss en Via 1	**X**		
Read Hit tras un lw_inc sobre bloque no cargado en memoria cache (no invalida) en Via 1	**X**		
Read hit después de operación scratch (mostrar independencia)		**X**	
Read hit de palabra que acaba de ser escrita en cache por un write miss + write through (se escribe palabra tras cargar el bloque)	**X**		
Read hit en ráfaga de todas las palabras en cada vías después de reemplazamiento	**X**		
Read Miss obligatorio en Via 0	**X**		
Read Miss por conflicto en Via 0	**X**		
Read Miss Obligatorio en Via 1	**X**		
Read Miss por conflicto en Via 1	**X**		
Read Miss debido a que se ha invalidado bloque por lw_inc	**X**		
Read Miss debido a que se acaba de explusar el bloque de la vía (se cargará en la otra vía)	**X**		
Read Miss tras un wrtie miss		**X**	
write hit sencillo en Vía 0	**X**		
write hit sencillo en Vía 1	**X**		
write hit tras un write miss	**X**		
write hit tras un read miss		**X**	
varios write hits sobre distintas vías consecutivos	**X**		
write Miss por conflicto en Vía 0	**X**		
write Miss obligatorio en Vía 0	**X**		
write Miss por conflicto en Vía 1	**X**		
write Miss obligatorio en Vía 1	**X**		
varios write miss y read miss seguidos sobre el mismo set		**X**	
varios write seguidos sobre la misma dirección		**X**	

Cuadro 3: Tabla de señales de comportamiento de estado

CASO DE PRUEBA	UNIT TEST MD	UNIT TEST MD_SCRATCH	UNIT TEST ERRORES
lw_inc con 'hit', que invalide	**X**		
lw_inc sobre bloque que acaba de ser traído de la cache, invalida	**X**		
lw_inc con 'miss', que no invalide	**X**		
lw_inc con sobre bloque que acaba de ser expulsado de la cache , que no invalide	**X**		
Reemplazamiento de un bloque debido a que ha sido invalidado por lw_inc	**X**		
lw en MD_Sratch		**X**	
sw en MD_Sratch		**X**	
lw_inc en MD_Sratch (tratado como lw)		**X**	
write en I/O register		**X**	
read en I/O register		**X**	
gestión de error al escribir en internal MC registers			**X**
read en internal MC registers			**X**
pasar a estado normal después de leer de registro interno			**X**
gestión de error debido a un acceso desalineado			**X**
gestión de error debido a que no se reconoce dirección en transferencia de bloque			**X**
gestión de error debido a que no se reconoce dirección en transferencia de palabra			**X**

Cuadro 4: Tabla de señales de comportamiento de estado

Los programas de prueba se encuentra **adjuntos** en el anexo 11.

6.2. Tests sobre MD

6.2.1. Read Miss

Para verificar el correcto funcionamiento de la acción *Read miss* se han hecho varias comprobaciones que se podrán ver en el *TEST_MD.asm*. La primera de ellas es mediante un *Read miss* de fallo *obligatorio*. Cuya instrucción es: *lw r1,96(r0)* y se encuentra en la quinta línea del test *TestMD.asm*

En esta primera imagen podemos ver como estando en una instrucción LW sobre un bloque reconocido por MD, hasta que la señal *BUS_GRANT* no se pone a 1, no se cambia al estado *BLOCK_TRANSFER_ADDR*, en el cual vemos que se activa la señal *BUS_DEVSEL*. Más abajo en la imagen, penúltima línea, podemos ver que *MD_BUS_DEVSEL* se pone a 1, con lo que podemos ver que MD reconoce la dirección de memoria que MC envía por el bus. También podemos ver que se activan las señales pertinentes para el envío de la dirección del bloque a traer durante el estado *BLOCK_TRANSFER_ADDR*, siendo estas: *MC_SEND_ADDR_CTRL* y también *FRAME*, que indica que se está realizando una transferencia y por lo tanto se está usando el bus. De ahí, se pasa al siguiente estado, que es el de *BRING_BLOCK_DATA*, en el cual permaneceremos hasta que consigamos leer todas las palabras del bloque que están siendo enviadas por el bus.

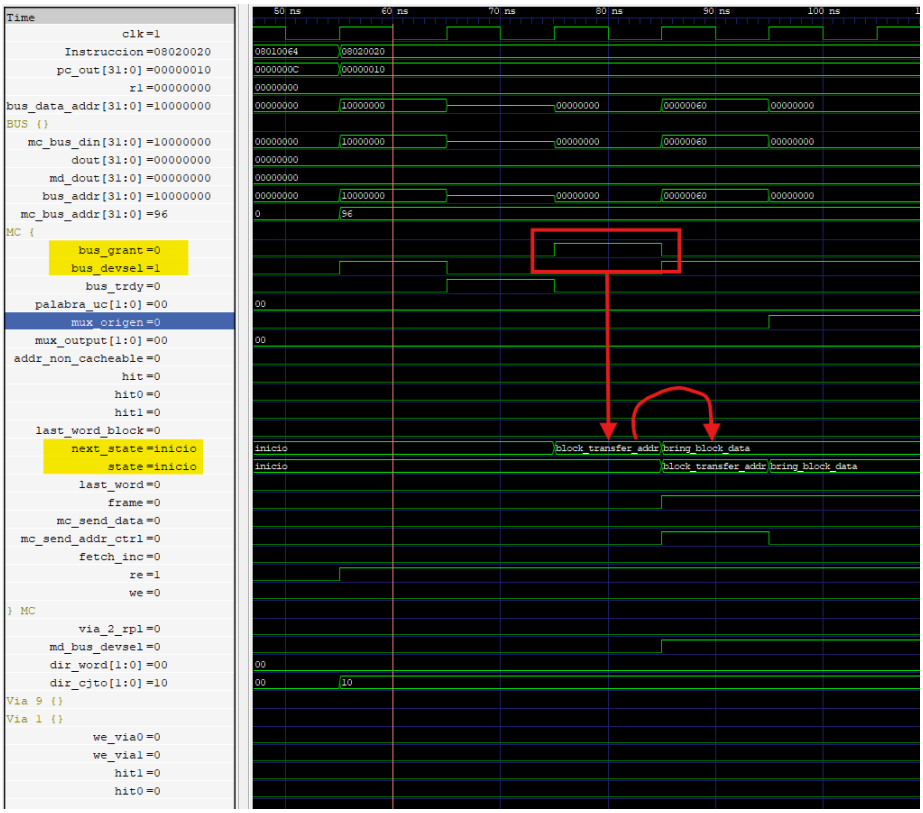

Figura 8: Primer set de señales de Read Miss de fallo obligatorio

En esta segunda imagen podemos ver el final de la transferencia de datos, que empieza con la señal *BUS_TRDY*, cuando el bus indica que está listo para transferir, podemos ver como avanza el contador de palabra y a su vez como se van guardando en la vía correspondiente, en este caso la vía 0, ya que es un Read Miss sobre un set vacío, es decir un Read Miss de fallo obligatorio.

Por último, en la última palabra se activa la señal *LAST_WORD*, que nos indica que se ha terminado la transferencia de palabras en el bus, con lo que vamos a volver al estado *INICIO*, en el que bajaremos *FRAME*, indicando al bus que hemos terminado.

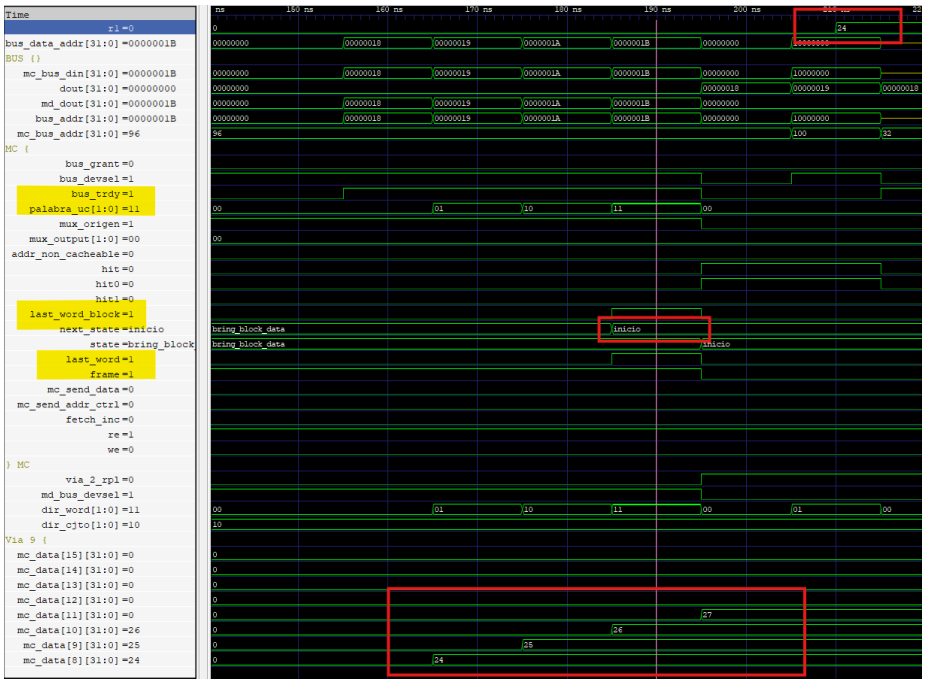

Figura 9: Segundo set de señales de Read Miss de fallo obligatorio

Tambien se ha comprobado a hacer un Read Miss de fallo de carácter de conflicto, es decir reemplazando a un bloque de una de las 2 vías. Las señales son idénticas a la prueba anterior, lo que si que se puede apreciar en estas imágenes, es que abajo en los datos de las vías, se puede ver como había valores previamente en la vía. En el test anterior, los valores eran de 0, es decir, que no estaban inicializados.

La instrucción es: *lw r5,160(r0)*, que se encuentra en la línea número 28 del test *TestMD.asm*.

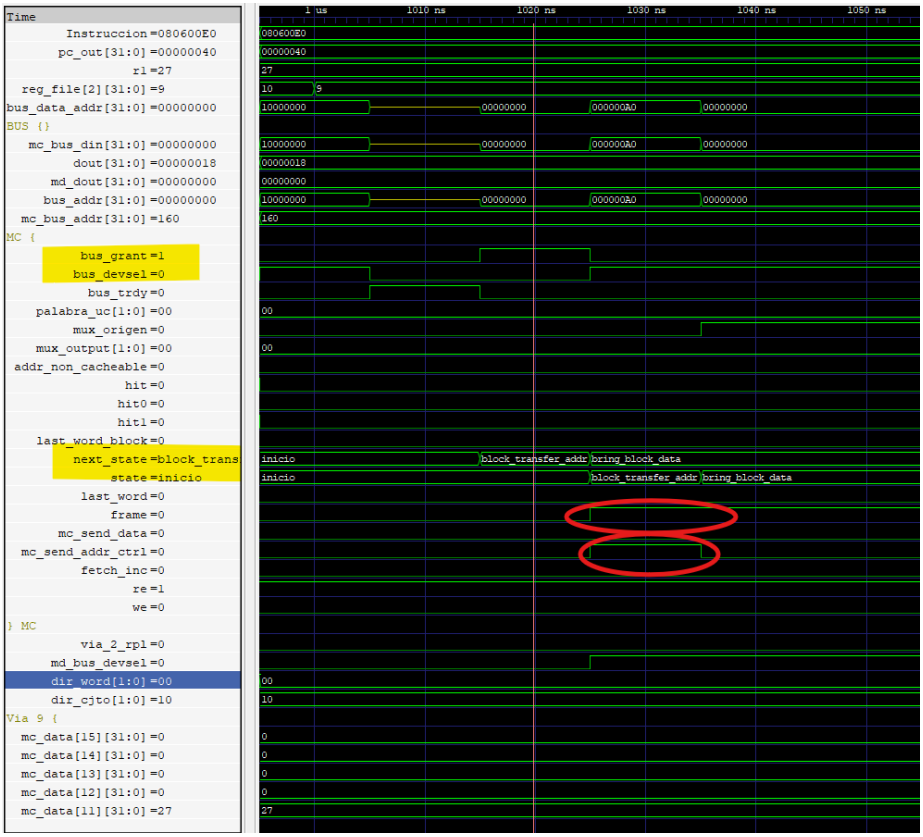

Figura 10: Your caption here

Estos 2 ejemplos guardan los bloques en memoria cache en la Vía 0, pero tambien se han probado los mismos tests para la vía 1, que se pueden encontrar en el *TEST_MD.asm*, pero que por motivos de redundancia y para que la memoria no sea de un tamaño infinito, se han decido no poner la imágenes en la misma. A efectos de señales, lo que cambia es la señal *via_2_rpl*, que se encuentra a 1 en vez de a 0, que se usa para indicarle a la memoria cache en que vía debe escribir.

6.2.2. Read Hit

Para verificar el correcto funcionamiento de un *Read Hit*, se han probado varios casos, que incluyen hacer lecturas directamente detrás de un Read Miss, Write Miss, varios Read Hit seguidos... en los que no se han observado diferencia a la hora de su comportamiento. Para este caso vamos a usar la representación de las señales de la instrucción: *lw r1,100(r0)* ,que aparece en la sexta línea del test *TestMD.asm*

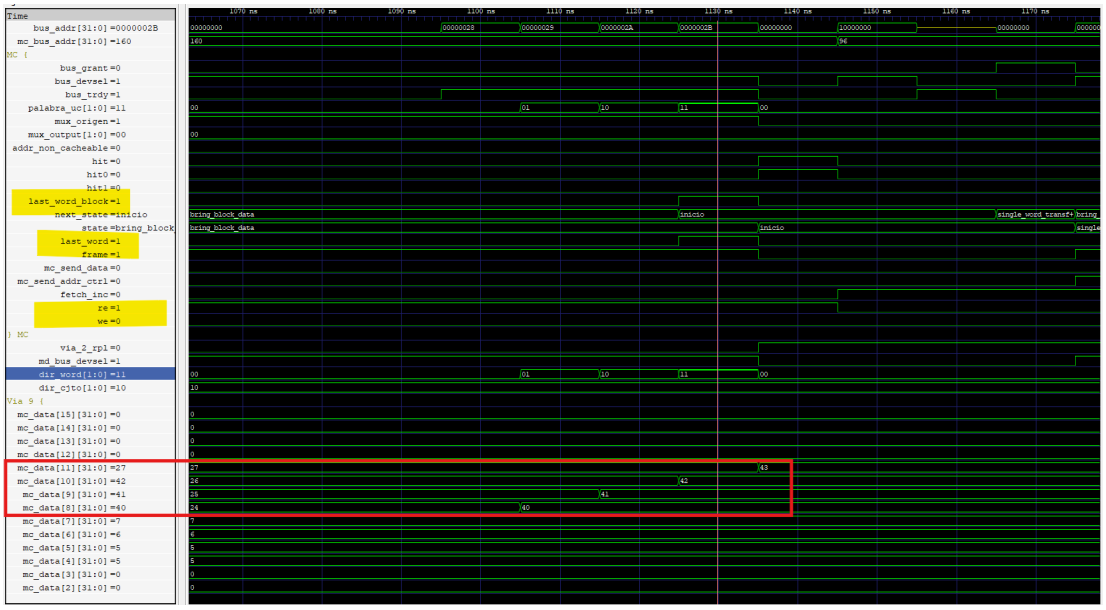

Figura 11: Your caption here

Figura 12: Set de señales de un Read Hit tras un Read Miss

Vemos como el acceso a memoria cache se produce en 1 ciclo, estando al ciclo siguiente ya escrito en el banco de registros, vemos que se produce ese Hit en la vía 0, y también que tanto el estado actual y siguiente de la memoria cache es *INICIO*, aunque la señal *BUS_DEVSEL* se encuentre remarcada con fosforito, no efecta ni tiene nada que ver con el estado de ejecución de nuestra instrucción que produce el Read Hit.

6.2.3. Write Miss

Para comprobar el correcto funcionamiento de un Write Miss se han probado varios casos, tanto escrituras del bloque tanto en vía 0 como en vía 1, con el miss generado por un bloque invalidado en memoria, para todos estos casos, la única señal que varía es *via_2_rpl*, por lo que simplemente se va a enseñar el funcionamiento de la instrucción: *sw r2,80(r0)*, que tambíen se encuentra en el test *TestMD.asm*, en la línea número 17

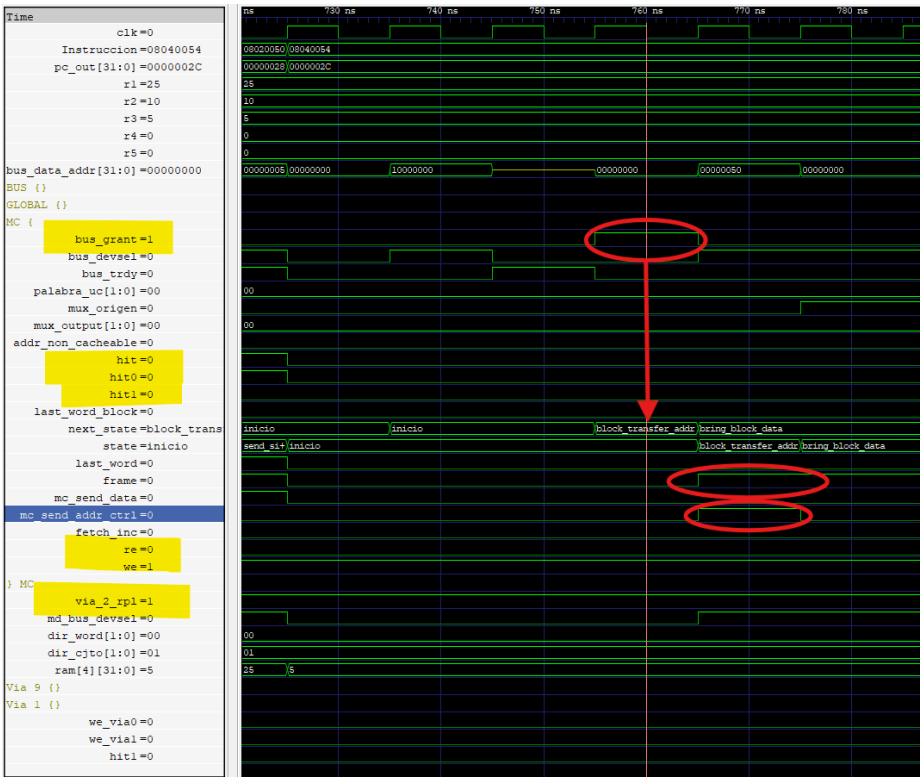

Figura 13: Primer set de señales de un Write Miss generado por: sw r2,80(r0)

En esta primera parte, podemos observar la política AF de nuestra memoria cache, que hace un Fetch del bloque para traerlo a memoria para posteriormente escribirlo mediante WT (Write Through), podemos ver en las señales

como la señal *Hit* es 0, de manera que el siguiente estado una vez se activa la señal *BUS_GRANT* va a ser el de *BLOCK_TRANSFER_ADDR*; en el que tras enviar la dirección del bloque a traer de MD, activando las señales pertinentes *MC_SEND_ADDR_CTRL* y *FRAME*, y tras recibir la señal *BUS_DEVSEL* igual a 1, se irá al estado *BRING_BLOCK_DATA*, el cual veremos en la siguiente imagen. Cabe destacar también que la escritura del bloque se hará sobre la via 1, dato que se observa dado el valor de la señal *via_2_rpl* que se encuentra a 1.

Figura 14: Segundo set de señales de un Write Miss generado por: sw r2,80(r0)

En esta segunda imagen podemos ver como la memoria cache guarda de manera correcta las palabras traídas de la MD, guardándolas en la vía 1, como está indicado anteriormente. Se va guardando las palabras en el estado *BRING_BLOCK_DATA*, hasta llegar a la última palabra, que es cuando tanto *BUS_TRDY* y *LAST_WORD* están activadas, momento de pasar al siguiente estado, que será el de *SINGLE_WORD_TRANSFER_ADDR*. En este, se mandará en por el bus la dirección de la palabra que quiere escribir, activando las señales pertinentes para la transferencia de la dirección por el bus, *MC_SEND_ADDR_CTRL*, y pasará al siguiente estado, el cual es *SEND_SINGLE_WORD_DATA*, estado en el que la MC actualizará el valor sobre la vía y palabra correspondiente, como podemos ver en *MC_DATA[4]*, aunque la actualización en MD no sucederá hasta que la MC no reciba la señal *BUS_TRDY*, momento en el que transferirá la palabra al bus. Hecho que se puede apreciar en la siguiente imagen:

Figura 15: Tercer y último set de señales de un Write Miss generado por: sw r2,80(r0)

6.2.4. Write Hit

Para la comprobación de los Write Hit, se han probado también varios casos, en ambas vías, después de diversas instrucciones... Para estos casos, la única señal que varía es *Hit0/Hit1*, dependiendo de la vía en la que se encuentra la palabra almacenada en memoria cache.+ Se van a analizar las señales de la instrucción: *sw r3,16(r0)*, que genera un Write Hit tras una instrucción que genera un Write Miss, estas instrucciones se encuentran en las líneas 14-15 del test *TestMD.asm*

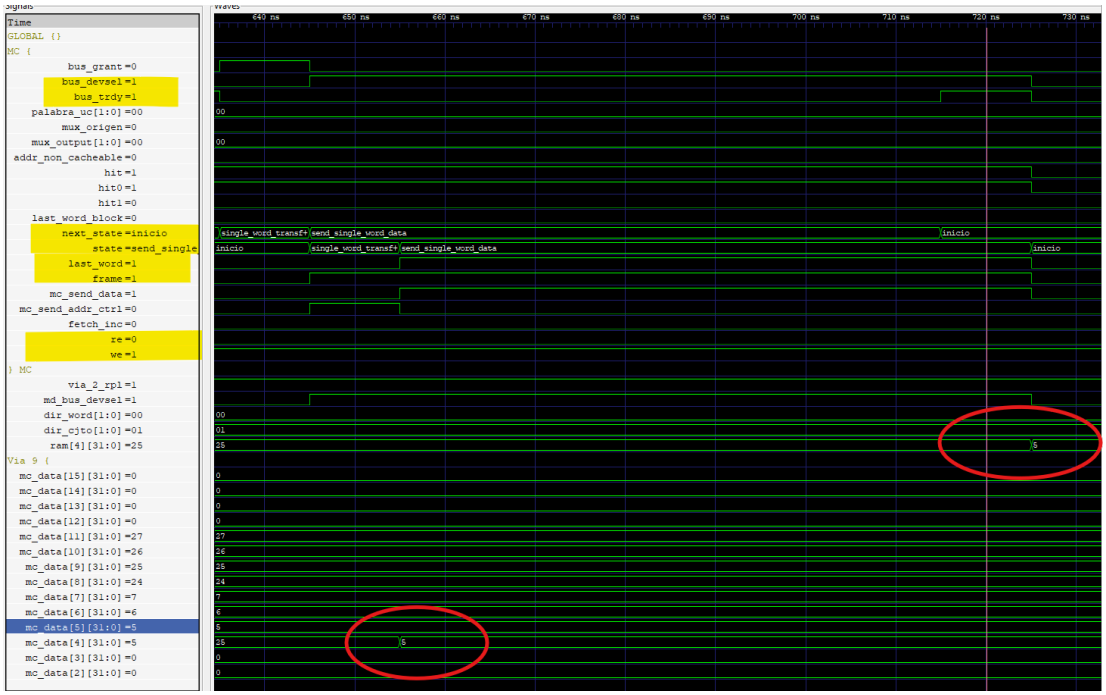

Figura 16: Set de señales de un Write Hit generado por: sw r3,16(r0)

Se puede ver en la imagen, como tras llegar la señal *BUS_GRANT*, se pasa al estado *SINGLE_WORD_TRANSFER_ADDR*, estado en el que se activa la señal *MC_SEND_ADDR_CTRL* y también se activa *FRAME*. Como se activa la señal *BUS_DEVSEL* el siguiente estado será *SEND_SINGLE_WORD_DATA*, estado en el cual MC actualiza el valor de la palabra en su vía correspondiente y estado en el que permanecerá hasta que el bus esté listo para la transferencia y así se lo indique mediante la activación de la señal *BUS_TRDY*. En ese ciclo, enviará la palabra por el bus y terminará la transferencia siendo su siguiente estado el estado inicial *INICIO*, estado en el que se bajará *FRAME* liberando así el bus.

6.2.5. lw_inc

Para los casos de *lw_inc* sobre MD, por la gestión que se ha decicido hacer del lw_inc, es decir, por diseño, se van a invalidar los bloques cuando se haga un Hit y se propagará al MIPS directamente; y, en caso de Miss, se propagará la palabra traída al MIPS directamente, sin guardar nada en MC. Por lo que se puede distinguir de esta manera los 2 claros casos que debemos de comprobar. Otras pruebas que incluyan una combinación de instrucciones, entre ellas lw_inc serán descritas más adelante.

Para el primer caso, vamos a probar el Hit, que lo haremos con la instrucción: *lw_inc r6,224(r0)*, que se encuentra en la línea 34 del test *TestMD.asm*

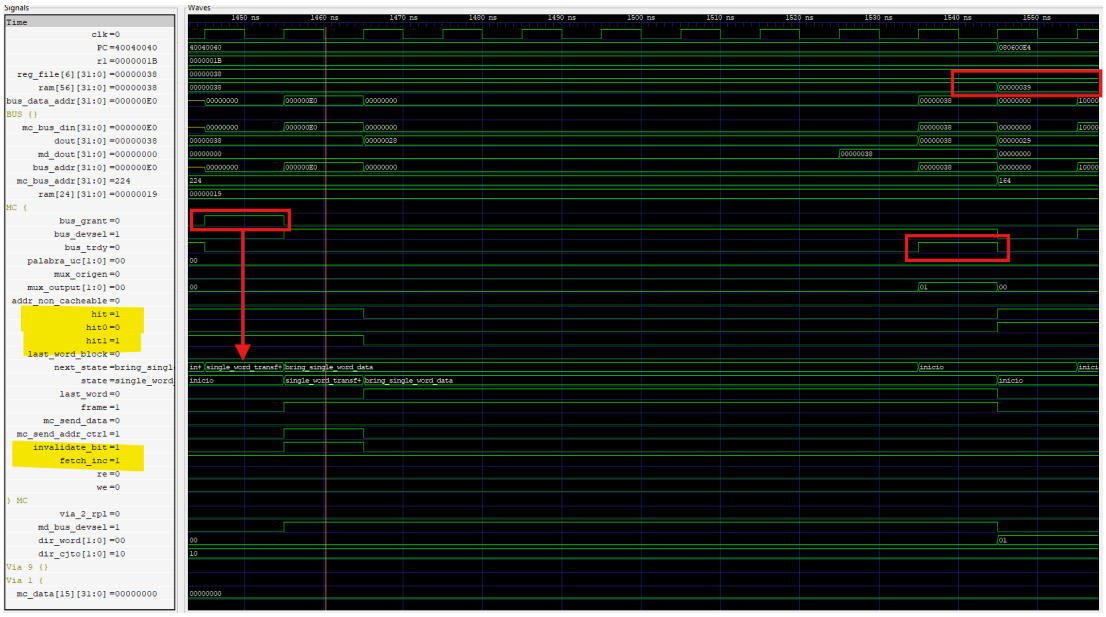

Figura 17: Set de señales de un Fetch Inc Hit generado por: lw_inc r6,224(r0)

En este set de señales podemos ver como las señales de un *lw_inc* está activadas, siendo estas las señales: *FETCH_INC*, y también al estar Hit a 1, la señal *INVALIDATE_BIT*. Cuando llega la señal *BUS_GRANT*, entonces se pasa a mandar la dirección de la palabra, al estado *SINGLE_WORD_TRANSFER_ADDR*, en el que se activan las señales de *MC_SEND_ADDR_CTRL* y *FRAME*. Al pasar al siguiente estado, *BRING_SINGLE_WORD_DATA*, se puede ver, a diferencia de todas las instrucciones que se han visto y estudiado hasta ahora que la señal de *Hit* baja a 0, hecho que se produce porque se ha invalidado correctamente el bloque en MC. Al igual que en el resto de casos, la UC de la MC esperará hasta que le llegué la señal *BUS_TRDY* para leer la palabra del *BUS*. Podemos ver que después del ciclo de INICIO se incrementa en 1 el valor de la palabra guardada en la @224 de MD.

Vamos a estudiar ahora el siguiente caso, el caso del *lw_inc* con Miss, para el cual se va a usar la instrucción: *lw_inc r8,96(r0)*, que se encuentra en la línea 29 del test *TestMD.asm*.

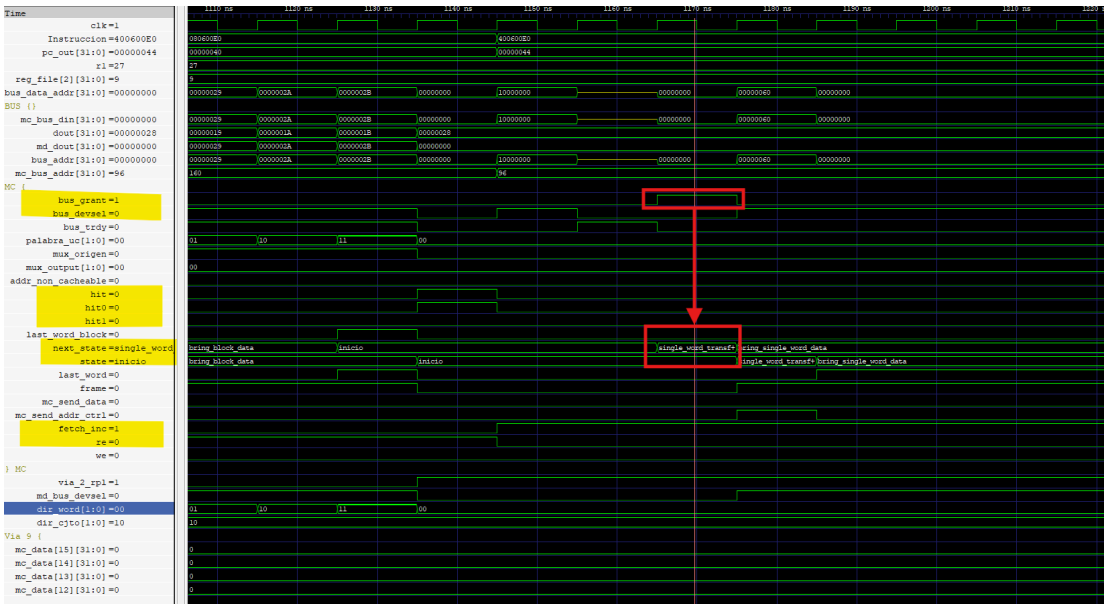

Figura 18: Primer set de señales de un Fetch Inc Miss generado por: lw_inc r8,96(r0)

En este set de señales se puede observar las señales identificadores del *lw_inc*, que es la señal *FETCH_INC*. También vemos que la señal *Hit* es 0, por lo que se "propagará"la palabra al MIPS directamente sin traer ningún bloque a MC, se espera a la señal *BUS_GRANT* para empezar la transferencia de la dirección de la palabra al bus, que se realiza en el estado *SINGLE_WORD_TRANSFER_ADDR*, con las señales pertinentes ya nombradas en el primer caso de prueba anterior, y se avanza al siguiente estado en el que se trae la palabra.

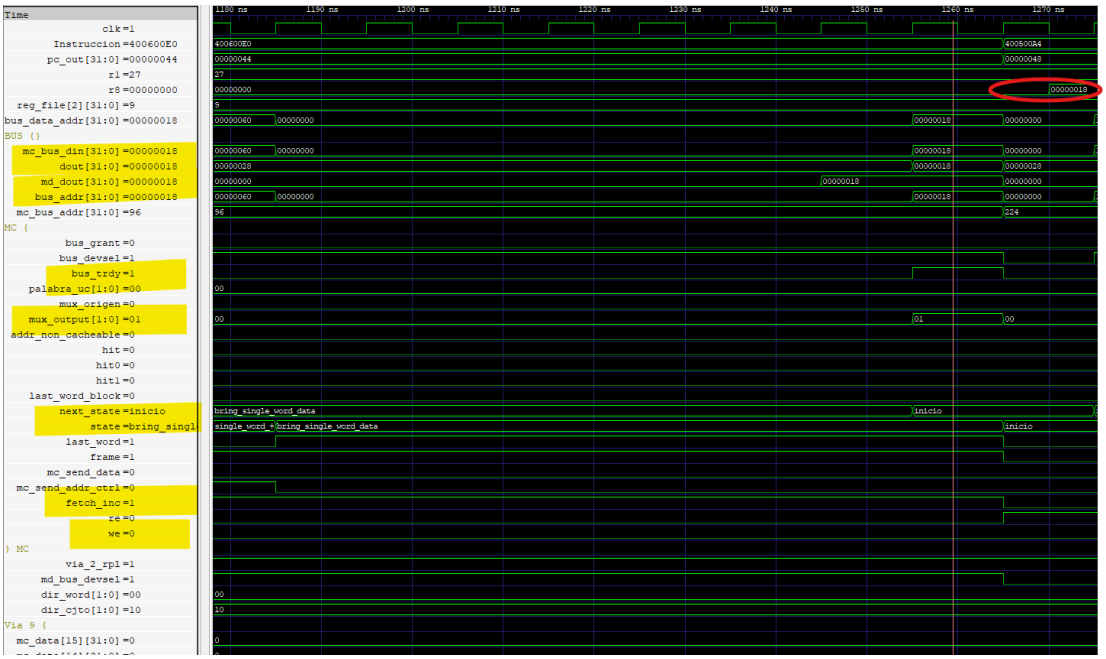

Figura 19: Segundo set de señales de un Fetch Inc Miss generado por: lw_inc r8,96(r0)

En este segundo set de señales se puede observar como se trae la palabra del bus cuando llega la señal *BUS_TRDY*. En ese momento, se pone la señal *MUX_OUTPUT* a "01", que hace que la salida de la MC sea *MC_BUS_DIN* por lo que la salida de la MC que está conectada al MIPS saque la palabra recién traída del bus.

Podemos ver como en el siguiente ciclo, en el flanco de bajada se escribe dicha palabra en registro.

6.3. Tests sobre MD_Scratch

Sobre MD_Scratch se han realizado varias pruebas, que están unitariamente algo más limitadas que las de MD, esto es debido al hecho de que al ser una memoria sustancialmente más rápida que MD, no se cachean las palabras traídas / escritas de esta memoria sobre MC. Vamos a ver en todas las pruebas, la señal *ADDR_NON_CACHEABLE*, que nos indica que la dirección sobre la que estamos trabajando se encuentra en *MD_Scratch*.

6.3.1. Read

Cada vez que se haga un lw sobre una dirección 'no cacheable', es decir de MD_Scratch, nunca se va a producir un Hit, y se irá iniciará la transferencia, por bus en cuanto el árbitro lo permita.

Para este caso vamos a usar la instrucción: *lw r1,12(r0)*, que se encuentra en la línea 4 del test *TestMD_Scratch.asm*

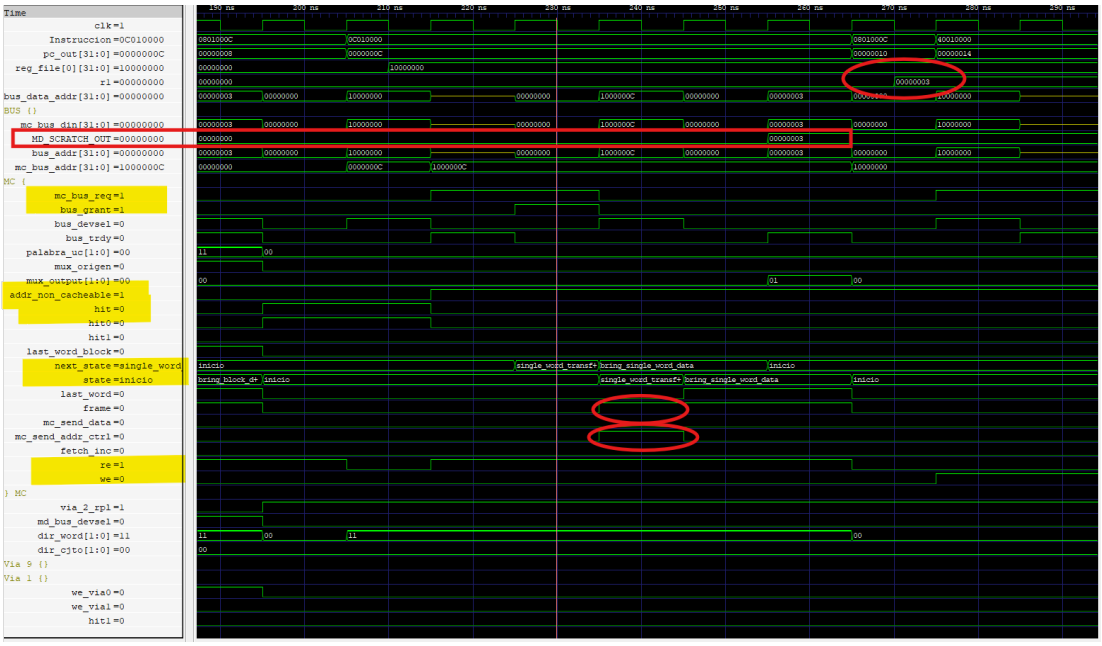

Figura 20

En este set de señales se puede observar como dicho anteriormente, la señal *ADDR_NON_CACHEABLE*, se puede observar también como, en vez de traer un bloque de memoria, el siguiente estado a la señal *BUS_GRANT* es *SINGLE_WORD_TRANSFER_ADDR* y posteriormente al activar las señales *MC_SEND_ADDR_CTRL* y *FRAME* es *BRING_SINGLE_WORD_DATA*, en el que se puede apreciar una gran diferencia entre la duración de ciclos de este estado de *BRING_SINGLE_WORD_DATA*

y el de la prueba anterior con el *lw_inc*, apreciándose claramente que en el caso de *MD_Scratch* dura unos cuantos ciclos menos. Se ha puesto la salida de *MD_Scratch* en el set de señales, en la que vemos que al ciclo siguiente de llegar *BUS_TRDY* y propagarse la palabra recién traída, se escribe de manera correcta en el banco de registros en el flanco de bajada.

6.3.2. Write

A diferencia de MD, las escrituras sobre MD_Scratch no se realizan en MC.

Se va a estudiar el Write mediante la instrucción: *sw r1,16(r0)*, que se encuentra en la línea 5 del test *TestMD_Scratch.asm*.

Figura 21: Set de señales de un Write sobre MD_Scratch generado por: sw r1,16(r0)

En este test de señales podemos ver otra vez la señal: *ADDR_NON_CACHEABLE*, al igual que las señales que se activan al llegar una instrucción de escritura como lo es *WE*. Al llegar la señal *BUS_GRANT*, se pasa al estado *SINGLE_WORD_TRANSFER_ADDR*, en el que se activan las señales *FRAME* y

MC_SEND_ADDR_CTRL y al siguiente ciclo se pasa al estado *SEND_SINGLE_WORD_DATA*, el cual se puede apreciar que solo dura un ciclo, al estar activado *BUS_TRDY* en ese mismo ciclo, consiguiéndose una escritura en apenas 2 ciclos de BUS.

6.3.3. Lw_inc

En este test se ha probado que la instrucción *LW_INC* al realizarse sobre MD_Scratch, se comporte como un LW "normal"sobre MD_Scratch, es decir que trae la palabra sin activar ninguna señal adicional.

Figura 22

En el siguiente set de señales, se puede observar que las señales son las mismas que en el apartado anterior de Read sobre MD_Scratch, podemos ver que la señal *BUS_FETCH_INC* no está activada, pese a que la señal *FETCH_INC* sí que lo esté. El resto de señales de los ciclos ya están explicados anteriormente.

6.4. Tests varios

6.4.1. Lw_inc Miss - Read Hit

Se ha probado el caso de la ejecución de un lw_inc que hace Miss, sobre el set '00', por lo que no invalida ningún bloque de MC, y después se realiza un LW sobre uno de los bloques que está guardado en el set '00' y que si el LW_INC funciona correctamente será un Read Hit limpio.

El Set de instrucciones que se prueba es el siguiente:

lw r1,0(r0)

lw r2,64(r0)

lw_inc r1,128(r0)

lw r1,0(r0)

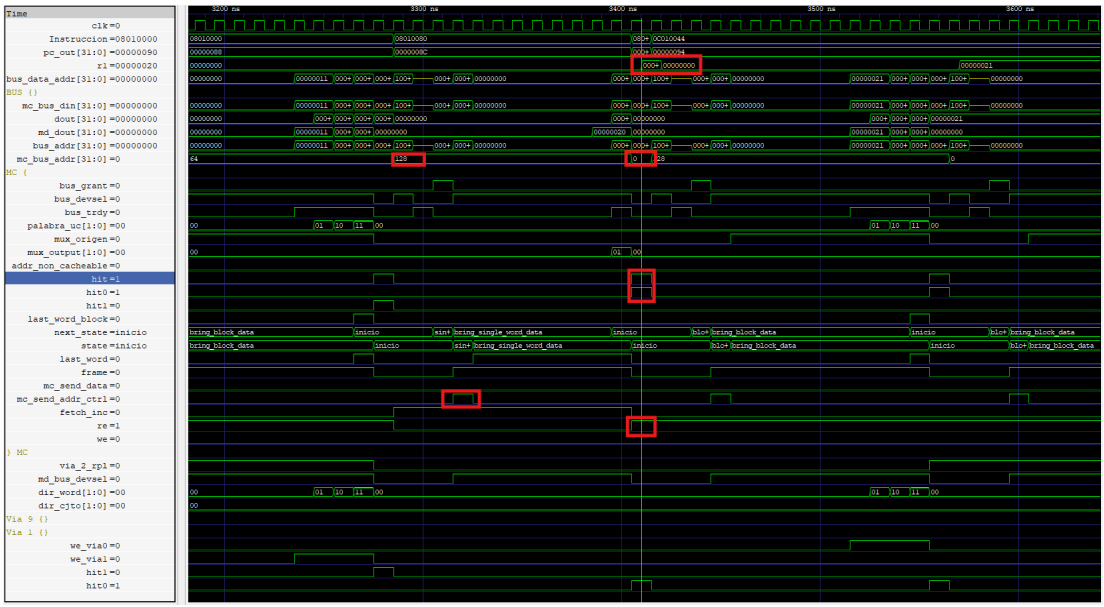

Figura 23

En el siguiente test de señales podemos ver, varias instrucciones, de las cuales solo nos interesan las 2 centrales. Vemos que la primera dirección es @128, que está dentro del set '00', como se puede apreciar también en el valor de *dir_cjto*, que es un *LW_INC*, indicado por la señal *FETCH_INC* con valor 1, y que carga el registro r1 con un valor al llegar al estado de inicio. En este, entra la instrucción *lw r1,0(r0)* al ciclo de memoria, en el cual, como se puede ver, se produce un Hit en la via 0 y se realiza la escritura en banco de registro en el ciclo siguiente, que se puede apreciar por la escritura del 0 en el registro *r1* del ciclo siguiente.

6.4.2. LW sobre un bloque recién expulsado

El siguiente test que se ha decidido hacer ha sido el de la lectura de un palabra perteneciente a un bloque que acaba de ser expulsado de la MC.

Las instrucciones han sido:

lw r1,128(r0)

lw r1,0(r0)

Cabe nombrar, que el test ha sido llevado a cabo en el *TestMD.asm* y que antes había sido ejecutada también la instrucción lw r2,64(r0), que carga un bloque en la via 1, quedando asi ocupadas las 2 vías del set '00'.

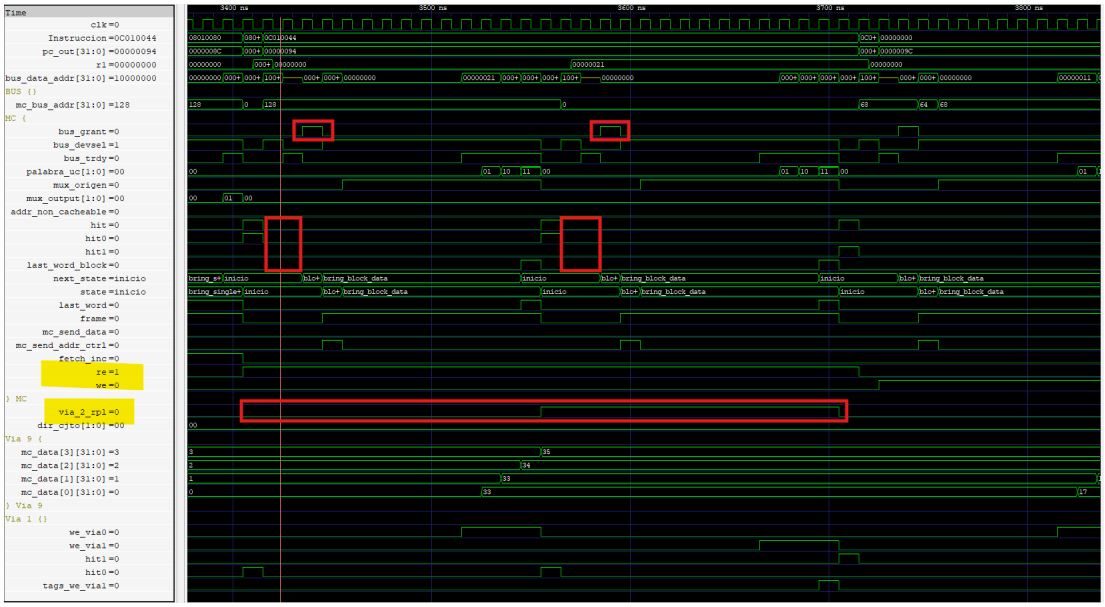

Figura 24

En el set de señales, podemos ver las señales de las 2 instrucciones. La primera de ellas, *lw r1,128(r0)*, intenta leer, pero eno se produce un Hit, por lo que va a memoria a traer el bloque de la forma vista anteriormente ya explicada de los Read Misses en MD, y trae el bloque a MC, escribiendo los TAGS e indirectamente cambiando la Vía a reemplazar, *via_2_rpl*. Al llegar la siguiente instrucción, antes en la vía 0, que intenta leer del bloque que acaba de ser reemplazado, se encuentra que Hit = 0. Por lo tanto ha producido un miss, y traerá el bloque de memoria con el comportamiento descrito anteriormente y los escribirá en la Vía 1, escribiendo también los TAGS y volviendo a dejar via_2_rpl a 0. En los contenidos de Vía 9, (el grupo debería de llamarse vía 0), vemos el contenido de la vía 0, en concreto las 4 primeras palabras, correspondientes al set 0, que cambian de valor de 0,1,2,3 a las correspondientes con el bloque

cargado por el primer lw.

7. TEST DE INTEGRACIÓN

7.1. Explicación

Para el Test de Integración se ha diseñado un programa en ASM que lo que hace es sumar 2 matrices 4x4, guardar en MD la matriz resultado y multiplicar dicha matriz por 2. Se han realizado la carga de variables de modo que se intenta dejar cada matriz en una vía para mejor representación. Se ha escogido el tamaño de 4x4 ya que es el tamaño perfecto para que quepa cada matriz en la misma vía, es decir que ocupe los 4 sets de memoria cache.

En este test se hacen varios bucles. El primero de ellos carga por orden componente a componente de cada fila de cada matriz, las suma y al llegar a la última las guarda en memoria. Se aprovecha al máximo la memoria cache, ya que como cada fila ocupa 4 palabras en memoria, la memoria cache trae todo el bloque de MD, teniendo solo 1 miss para cargar las 4 componentes de la fila. Lo mismo sucede para guardar la matriz resultado, en el primer sw se produce un miss, pero al traer el bloque de memoria los 3 siguientes son hit. De esta manera se optimiza al máximo el uso de la memoria cache, y el bucle acaba al realizar las 4 iteraciones, 1 por fila.

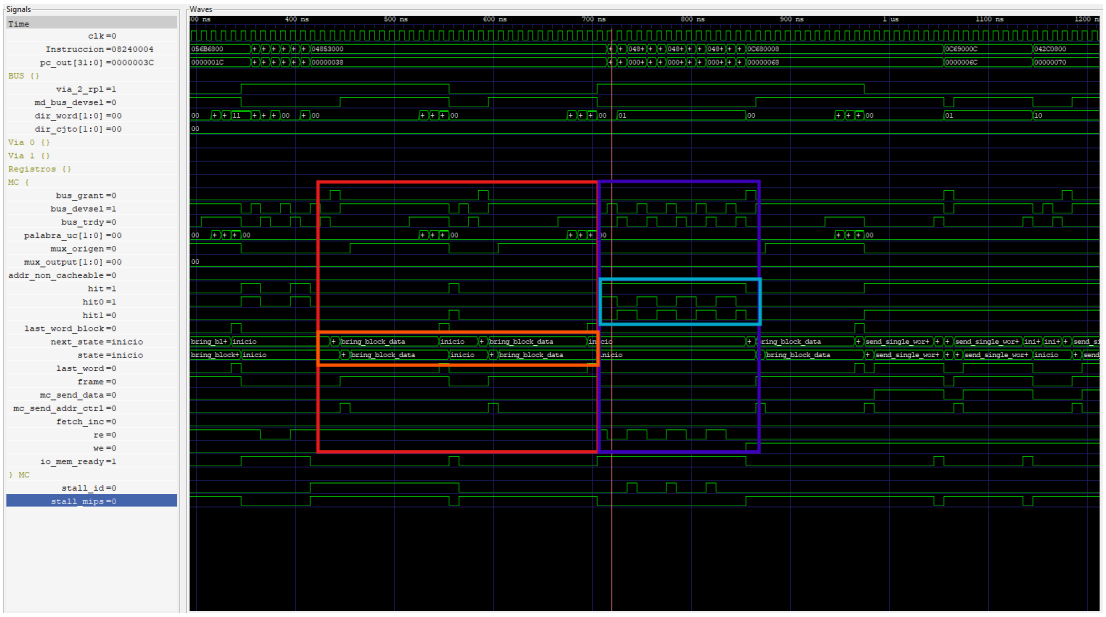

Figura 25

El siguiente bucle carga en registro fila por fila la matriz resultado, la multiplica por 2 y la vuelve a guardar. Como al hacer sw antes para guardar el resultado se traen los bloques y se guardan en MC, los lw ya dán Hit desde el principio, por lo que el lw de la matriz se produce en 1 ciclo por componente,

ahorrándonos el primer read miss que produciría leer cada fila sin tener nada en la cache antes. Todos los lw y sw producirán hits en este bucle.

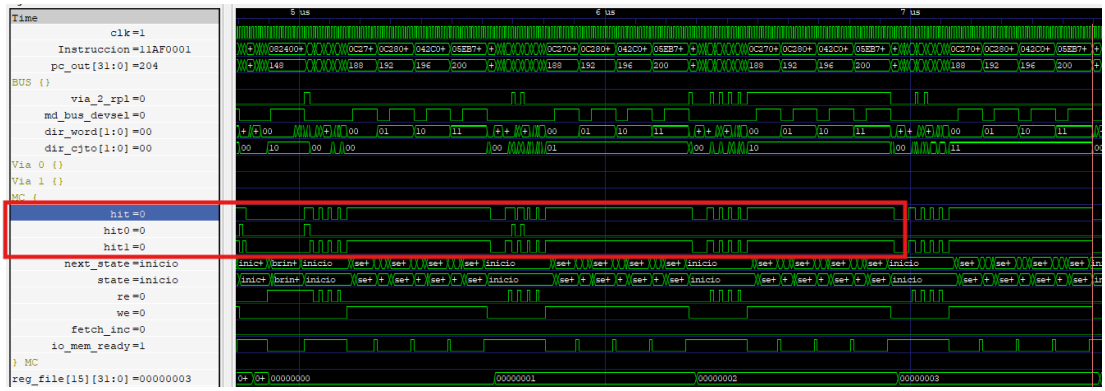

Figura 26

La ultima parte del test consiste en invalidar los remnants"de las matrices restantes en memoria cache y volver a realizar el primer bucle, en el que vemos que los ciclos son los mismos, ya que tiene que se producen los mismos misses que en el primer bucle.

7.2. Speedup

Para el cálculo del speedup de este test, lo que se ha hecho ha sido coger el tiempo de ejecución del programa ejecutado en el MIPS con MC de funcionamiento 'normal', es decir, con la UC diseñada para este proyecto. Pero también se ha ejecutado en con una **UC modificada** para que, siempre que haya read o write, se pida request en el bus sin usar la MC y siempre traiga de memoria.

Para el primer de los casos descritos, el programa ha tardado: 12315 ns

Para el segundo de los casos descritos, el programa ha tardado: 14875 ns

$$Speedup = \frac{14875ns}{12315ns} = 1,20787$$

8. CUANTIFICACIÓN DE HORAS DEDICADAS

	ATHANASIOS USERO	**EDUARDO SÁNCHEZ**
Estudio previo (MC, etc)	6 h	3 h
Elaboración del autómata de la MC	6 h	5 h
Descomposición de la dirección y latencias	4 h	3 h
Cálculo de *Ceff*	1 h	1 h
Tests unitarios	6h	12 h
Test de integración	1h	6 h
Memoria	15h	8 h
Apartado opcional: Write buffering	1h	8 h
Apartado opcional: Ethical Hacking	8h	1 h
TOTAL	48h	47 h

Cuadro 5: Cuantificación de horas dedicadas

9. EVALUACIÓN INDIVIDUAL

Las sensaciones tras la terminación del segundo proyecto son **positivas**. El equipo considera que ha mantenido las decisiones que funcionaron bien en el primer proyecto, como tener paciencia y estudiar pausadamente el sistema proporcionado antes de comenzar con las implementaciones. Además, trataron de corregirse algunos aspectos mejorables del primer proyecto, como con la documentación más clara de la fase de pruebas a través de una tabla de casos.

Más allá de ello, también se encuentra satisfecho respecto a los objetivos del mismo. Han necesitado estudiarse varios aspectos específicos de las caches y buses, y en muchos puntos el trabajo previo en las clases de problemas fue de mucha ayuda. Además, el trabajo de los apartados opcionales, donde había menos documentación y donde se modificó con *libertad* la arquitectura proporcionada permitió investigar y experimentar con los contenidos de la asignatura.

En conclusión, el equipo valora el trabajo con optimismo dado el esfuerzo e interés invertidos, aunque se quedase con las ganas de elaborar el último apartado opcional por las limitaciones de tiempo.

Eduardo Sánchez Sarsa (901813):

- Do you believe you achieved the course objectives?

Yo creo que sí que he conseguido aprender lo que este asignatura pretende enseñar. Creo que he sido capaz de pensar, desarrollar y resolver de manera adecuada y correcta los problemas y dilemas que esta asignatura plantea.

- What grade would you give yourself?

Yo me daría una nota de entre 8,5 y 9,5. Creo que no hubiese sido posible el magnífico desarrollo y trabajo de este proyecto de no ser por mi compañero Athanasios, y si bién ha sido un esfuerzo de equipo, creo que gracias ha él he sido capaz de seguir trabajando y esforzándome por la realización de los proyectos optativos. Que igual no hubiese intentado de no haber hecho el trabajo con él.

Athanasios Usero Samarás (839543) :

- Do you believe you achieved the course objectives?

A mi parecer, creo que he intentado y conseguido obtener las trazas de algunas de la ideas principales de la asignatura. Es cierto que, para este proyecto, se ha trabajado un escenario concreto (MC 2-asociativa, bus semisíncrono, etc); pero la resolución de los problemas planteados no puede ser correcta sin algunos conocimientos generales. Ocurre así con la descomposición de la dirección, con el hecho de una comunicación *Master-Slave* o de lo que conlleva invalidar un bloque de una cache, por ejemplo.

- What grade would you give yourself?

Por el esfuerzo realizado, pondría una buena nota (el valor exacto es más complicado de determinar). También me gustaría agradecer a mi compañero Edu, por siempre mostrar ayuda cuando se ha necesitado y por su asertividad y comprensibilidad a la hora de resolver conflictos.

10. AGRADECIMIENTOS

Al equipo le gustaría agredecer a varias personas que han ayudado directa o indirectamente con la realización de este proyecto. La primera de ellas es Antonio Jose Antonio Secadura del Olmo, que con su script se han podido codificar y realizar multiples tests de manera sencilla y rápida sin tener que usar otra herramienta de codificación más lenta.

11. ANEXO 1. PROGRAMAS DE PRUEBA PARA TEST UNITARIOS

11.1. TestMD

```
1  add r0,r0,r0  # Add que no afecta a MD
2
3  #Set: "10" -> VIA 0: 96,100,104,108 ; VIA 1: 32,36,40,44
4
5  lw r1,96(r0)  # Read miss sobre via 0
6  lw r1,100(r0) # Read hit sobre via 0
7
8  lw r2,32(r0)  # Read miss sobre via 1
9  lw r2,40(r0)  # Read hit sobre via 1
10
11 #Set: "01" -> VIA 0: 16,20,24,28 ; VIA 1: 80,84,88,92
12
13 sw r1,16(r0)  # Write miss sobre via 0
14 lw r3,20(r0)  # Write hit sobre via 0
15 sw r3,16(r0)  # Write hit sobre via 0
16
17 sw r2,80(r0)  # Write miss sobre via 1
18 lw r2,80(r0)  # Read hit sobre via 1
19 lw r4,84(r0)  # Read hit sobre via 1
20
21 #Vuelvo a Set "10" para tests "SENCILLOS"
22
23 lw r1,108(r0) # Read hit sobre via 0
24 lw r2,36(r0)  # Read hit sobre via 1
25
26 #Tests de conflicto Set "10"
27 #VIA 0: 160,164,168,172 ; VIA 1: 224,228,232,236
28
29 lw r5,160(r0) # Read miss que reemplaza sobre via 0
30 lw_inc r8,96(r0)# lw_inc miss que no invalida
31 lw r6,224(r0) # Read miss que reemplaza sobre via 1
32
33 #Tests de lw_inc invalidan: Set "10" ->
34 VIA 0: 160,164,168,172 y VIA 1: 224,228,232,236
35
36 lw_inc r6,224(r0)#lw_inc hit que invalida via 1
37 lw_inc r5,164(r0)#lw_inc hit que invalida via 0
38 lw_inc r4,64(r0) #lw_inc miss que no invalida nada.
39
40 #Invalidados. Set "01" -> VIA 0: 144,148,152,156 ; VIA 1:
       208,212,216,220
41
42 lw r6,228(r0) # Read miss que reemplaza via 0 Set "10"
43 sw r6,144(r0) # Write miss que escribe en via 0 Set "01"
```

```
44 sw r7,208(r0) # Write miss que escribe en via 1 Set "01"
45
46 #Varios seguidos.
47
48 sw r1,160(r0) # Write miss que reemplaza via 1 Set "10"
49 sw r1,224(r0) # Write hit
50 sw r2,148(r0) # Write hit
51 sw r2,216(r0) # Write hit
52
53 #Read hit seguidos
54
55 lw r1,144(r0) # Read hit
56 lw r1,148(r0) # Read hit
57 lw r1,152(r0) # Read hit
58 lw r1,156(r0) # Read hit
59
60 #Set: "00" -> VIA 0: 0,4,8,12 , VIA 1: 64,68,72,76
61
62 lw r1,0(r0)      # Read miss sobre via 0
63 lw r2,64(r0)     # Read miss sobre via 1
64 lw_inc r1,128(r0) # lw_inc miss que no invalida.
65 lw r1,0(r0)      # Read hit sobre via 0
66 lw r1,128(r0)    # Read miss que reemplaza sobre via 0
67 lw r1,0(r0)      # Read miss que reemplaza sobre via 1
68 sw r1,68(r0)     # Write miss que reemplaza sobre via 0
69
70 sw r1,144(r0)      # Write hit sobre Set "01" VIA 0
```

Listing 3: TestMD.asm

11.2. TestMD_Scratch

```
#Primero cargamos la direcci n de MD_Scratch
lw r0,0(r0)    # Read sobre MD_Scratch

lw r1,12(r0)   # Read sobre MD_Scratch
sw r1,16(r0)   # Write sobre MD_Scratch
lw r1,16(r0)   # Read sobre MD_Scratch

lw_inc r1,20(r0)#lw_inc sobre MD_Scratch (lw)

#Vuelta a MD

lw r2,4(r31)   # Read miss sobre Set "00" via 0
sw r1,4(r31)   # Write hit sobre Set "00" via 0

sw r1,16(r31) # Write miss sobre Set "01" via 0
sw r1,80(r31) # Write miss sobre Set "01" via 1
lw r3,144(r31)   # Read miss sobre Set "01" via 0
lw r4,208(r31)   # Read miss sobre Set "01" via 1

sw r1,144(r31)   # Write hit sobre Set "01" via 0
sw r3,144(r31)   # Write hit sobre Set "01" via 0
sw r4,144(r31)   # Write hit sobre Set "01" via 0

lw_inc r5,16(r31) # lw_inc miss que no invalida.
lw r3,144(r31)   # Read hit sobre Set "01" via 0


#I/O Registers ^^

sw r4,28672(r31)   # Intento de Write sobre Input Register
lw r4,28672(r31)   # Intento de Read sobre Input Register

sw r3,28676(r31)   # Intento de Write sobre Output Register
lw r4,28676(r31)   # Intento de Read sobre Output Register

sw r2,28680(r31)   # Intento de Write sobre ACK Register
lw r4,28680(r31)   # Intento de Read sobre ACK Register
```

Listing 4: TestMD$_{S}$*cratch.asm*

11.3. Test_Errores

```
;; TEST DE COMPROBACI N DE ERRORES. COMPRENDE TRES TIPOS DE ERRORES.
;; 1. Acceso desalineado a palabra
;; 2. Direcci n de bloque o palabra no reconocida por ning n dispositivo (devsel = 0)
;; 3. Error por intento de lectura de un registro interno
;; Adem s , se ver  como el sistema pasa a estado normal despu s de leer de un registro interno.

Reset: beq  R1, R1, INI ;@0x0 (Salta a INI en 0x10)
IRQ: beq  R1, R1, RTI ;@0x4 (Salta a IRQ en 0x14)
DAbort: beq R1, R1, RT_Abort;@0x8 (Salta a DAbort en 0x18)
UNDEF: beq R1, R1, RT_Undef;@0xC (Salta a UNDEF en 0x1C)

;; CONTENIDO MD: 1, 0x10000000, 0x00000AB0, 0x01000000, 0x0BAD0C0D;
;; CONTENIDO MD : ULTIMA PALABNRA CON UN 0x4
INI:
    LW R2, 4(r0) ;@0x10 (Cargar el valor del inicio de memoria scratch 0x10000000)
                ; READ MISS ->  VIA 0 SET 0
    LW, R3, 8(R0) ;@0x14 (Cargar el valor de error abort de la direcci n 0x0004)
                 ; READ HIT -> VIA 0 SET  0
    SW R3, 4(R2) ;@0x18 (Almacenar el valor de R2 en la direcci n 0x01000004)
                ; WRITE IN MDSCRATCH
    LW R3, 12(R0) ; @0x1C (Cargar el valor de direcci n de ADDR_ERROR_Register de la direcci n 0x0008)
                 ; READ HIT -> VIA 0 SET 0
    SW R3, 8(R2) ;@0x18 (Almacenar el valor de ADDR_Error_Register en la direcci n 0x01000008)
                ; WRITE IN MDSCRATCH

    ;;  Probar acceso desalineado a palabra

    LW_INC R4,65(R0); @0x20 (Cargar el valor de R3 en la direcci n 0x00000001)
                   ; ERROR DESALINEADO -> "SET 0", PERO NO HAY REEMPLAZO
    LW R5, 0(R3) ;@0x24 (Volvemos a leer el valor del registro de error, pero ya estaba en modo normal)
                ; READ HIT DE INTERNAL REGISTER
    SW R5, 12(R0); @0x28 (donde estaba el valor de registro de error se guarda direcci n de valor invalido)
                ; WRITE HIT -> VIA 0 SET 0 (porque instrucci n anterior era err nea)

```

```
35      ;; Probar Direcci n de bloque o palabra no reconocida
            por ning n dispositivo (devsel = 0)
36      LW R6, 512(R0); @0x2C (Intentar cargar el valor de una
            direcci n no mapeable 0x00000200)
37                      ; READ ERROR -> DISPOSITIVO NO
                            RECONOCIDO
38      LW r6, 508(R0); @0x30 (Cargar el valor de la direcci n
            de error de la direcci n 0x000001FC)
39                    ; READ MISS -> VIA 1 SET 3
40      SW r6, 32(R0) ; @0x34 (Almacenar el valor de R6 en la
            direcci n 0x00000020)
41                    ; WRITE MISS -> VIA 0 SET 2
42
43      ;; Probar escritura de registro interno
44      SW r6, 0(R3); @0x38 (Intentar escribir en el registro de
             error el valor 4)
45                    ; WRITE ERROR -> INTENTO DE ESCRITURA DE
                        REGISTRO INTERNO
46
47      beq R0, R0, 65535 ;@0x3C BUCLE INFINITO
48
49  RTI:
50      LW R1, 0(R0) ;@0x40 (Cargar en R1 el valor 1)
51                    ; READ HIT
52      SW R1, 28680(R0) ;@0x44 (Almacenar el valor de R1 en la
            direcci n 0x7008)
53      RTE ;@0x48 (Retornar de la interrupci n)
54
55  DAbort:
56      LW R1, 4(R2) ;@0x4C (Cargar en R1 el valor de error
            abort de registro de error de memoria scratch 0
            x10000004)
57      SW R1, 28676(R0) ;@0x50 (Almacenar el valor de R1 en la
            direcci n 0x7004)
58                        ; LECTURA DE IO -> NO INTERVIENE
                              MEMORIA CACHE
59      LW R1,  8(R2) ; @54LEER DE MEMORIA SCRATCH EL VALOR DE
            LA DIRECCI N DE ERROR INTERNO 0x10000008
60      LW r1, 0(R1) ;@58 (leer del registro ADDR_Error_Register
             el valor de la direcci n problem tica)
61                          ; PASA A ESTADO NO ERROR
62                          ; LECTURA DE REGISTRO INTERNO
63      SW R1, 48(R0) ;@0x5C (Almacenar el valor de R1 en la
            direcci n 0x0030)
64                    ; 1er ERROR: WRITE MISS -> VIA 0 SET 3
65                    ; 2o y 3er ERROR: WRITE HIT -> VIA 0 SET 3
66      RTE;@0X60 Volver a Ini
67
68
69  RT_Undef: LW R1, 16(R0) ;0x64 (cargar  0x0BAD0C0D)
```

```
70                          ; READ MISS la 1 vez (read hit las
                               siguientes) -> via 0 set 1
71          SW R1, 28676(R0) ;@0x68 (Almacenar el valor de R1 en
                la direcci n 0x7004)
72          beq R0, R0, 65532 ;@0x6C (#inm = -3)
```

Listing 5: Test$_{E}rrores.asm$

11.4. Test Integrado

```
# Programa que recorre una matriz 4x4 usando bucles
# Matriz A: posiciones 0-60 (16 palabras de 4 bytes)
# Matriz B: posiciones 64-124 (16 palabras)
# Matriz Resultado: posiciones 128-188 (16 palabras)
# Los datos se encuentran en la RAM de datos de "RAM PARA TEST INTEGRADO"


# Inicializaci n de registros
add r0, r0, r0      # r0 = 0 (direcci n base)
add r14, r0, r0     # r14 = 0
add r15, r0, r0     # r15 = 0 (contador de bucle)
lw r11, 272(r0)     # Cargar constante 1
lw r12, 264(r0)     # Cargar constante 16
#r13 para fin de bucle
add r13,r11,r11   # r13 = 1 + 1 = 2
add r13,r13,r13   # r13 = 2 + 2 = 4


# Inicializar punteros
add r1, r0, r0      # r1 apunta al inicio de matriz A
lw r2, 256(r0)      # r2 = 64 (inicio de matriz B, cargar desde memoria)
lw r3, 288(r0)      # r3 = 128 (inicio de matriz Resultado, cargar desde memoria)
# Cargar registro useless para ocupar bien una via entera.

lw r16,304(r0)

# Bucle principal para recorrer la matriz
bucle_inicio:
    lw r4, 0(r1)        # Cargar elemento de A
    lw r5, 0(r2)        # Cargar elemento de B
    add r6, r4, r5      # Sumar elementos

    lw r4,4(r1)
    lw r5,4(r2)
    add r7, r4, r5

    lw r4,4(r1)
    lw r5,4(r2)
    add r8, r4, r5

    lw r4,4(r1)
    lw r5,4(r2)
    add r9, r4, r5

    sw r6,0(r3)
```

```
46      sw r7,4(r3)
47      sw r8,8(r3)
48      sw r9,12(r3)
49
50      add r1,r1,r12    # Avanzar punteros...
51      add r2,r2,r12
52      add r3,r3,r12
53
54      add r15,r15,r11    # Avanzar contador
55
56      beq r15,r13, FIN_BUCLE1
57      beq r0,r0, INICIO_BUCLE1
58
59
60  fin_bucle1:
61
62  # Segunda fase: multiplicar cada elemento por 2
63  # Restablecemos contadores
64  add r15, r0, r0       # Reiniciar contador
65  lw r1, 288(r0)       # r1 = 128 (inicio de matriz Resultado)
66  lw r3, 292(r0)       # r3 = 192 (nueva ubicaci n para
        almacenar)
67
68  # Bucle de multiplicaci n por 2
69  bucle_mult:
70      lw r4, 0(r1)         # Cargar elemento de Resultado
71      add r5, r4, r4       # Multiplicar por 2 (sumando consigo
            mismo)
72      lw r4,4(r1)
73      add r6, r4, r4
74      lw r4,8(r1)
75      add r7, r4, r4
76      lw r4,12(r1)
77      add r8, r4, r4
78      sw r5, 0(r1)         # Guardar resultado
79      sw r6, 4(r1)
80      sw r7, 8(r1)
81      sw r8, 12(r1)
82
83      add r1,r1,r12    # Avanzar puntero
84      add r15, r15, r11      # Incrementar contador
85
86      beq r15, r13, fin_bucle2  # Si hemos terminado, salir
87      beq r0,r0,BUCLE_MULT       # Volver al inicio del bucle
88  fin_bucle2:
89
90  # Invalidaci n de cach  usando lw_inc
91  lw r2, 256(r0)       # r2 = 64 (inicio de matriz B)
92  lw_inc r4,0(r2)
93  lw_inc r4,16(r2)
```

```
94  lw_inc r4,32(r2)
95  lw_inc r4,48(r2)
96  # Esto de arriba deber a invalidar los 4 bloques que siguen
        en la v a 1.
97
98
99  # Repetir primer bucle para ver diferencia de rendimiento
100 add r15, r0, r0      # Reiniciar contador
101 add r1, r0, r0       # r1 apunta al inicio de matriz A
102 lw r2, 256(r0)       # r2 = 64 (inicio de matriz B)
103 lw r3, 276(r0)       # r3 = 320 (otra ubicaci n para
        almacenar)
104
105 # Repetir bucle principal tras invalidaci n
106 bucle_repetir:
107     lw r4, 0(r1)          # Cargar elemento de A
108     lw r5, 0(r2)          # Cargar elemento de B
109     add r6, r4, r5        # Sumar elementos
110
111     lw r4,4(r1)
112     lw r5,4(r2)
113     add r7, r4, r5
114
115     lw r4,4(r1)
116     lw r5,4(r2)
117     add r8, r4, r5
118
119     lw r4,4(r1)
120     lw r5,4(r2)
121     add r9, r4, r5
122
123     sw r6,0(r3)
124     sw r7,4(r3)
125     sw r8,8(r3)
126     sw r9,12(r3)
127
128     add r1,r1,r12    # Avanzar punteros...
129     add r2,r2,r12
130     add r3,r3,r12
131
132     add r15,r15,r11    # Avanzar contador
133
134     beq r15,r13, FIN
135     beq r0,r0, bucle_repetir
136 FIN:
```

Listing 6: TestIntegrado.asm

12. ANEXO 2. APARTADO OPCIONAL : WRITE BUFFERING

12.1. Explicación

Se plantea la resolución de un apartado opcional que requiere la siguiente funcionalidad: *Include a buffer for writes to MD. When a write to MD is required, the data and its address are stored in dedicated registers, and the processor is signaled to continue by asserting Mem Ready. The MC may continue handling hits while the Control Unit manages the write over the bus.* Es decir, tenemos que crear un **Buffer** para que la MC pueda responder a los Read Hit de manera 'paralela' a los writes en memoria. Para ello lo que se ha hecho es diseñar una extensión de la Unidad de Control previamente diseñada. Por lo que a partir de ahora, vamos a separar los estados que escriban en la MD, es decir los Write Hit y Write Miss a MD. Todo lo demás seguirá haciéndose en los estados anteriormente descritos.

Estos nuevos estados replican el funcionamiento de los estados anteriores, con la diferencia de que señalan que la instrucción está 'Bufferizada' mediante una señal añadida tanto a la UC de la memoria cache como a la memoria cache. Por supuesto, ha sido necesario añadir varios registros para almacenar la dirección, el dato a enviar y si la instrucción había producido un Hit o no. Este último registro si bien no está especificado en el enunciado, se ha visto necesario para la implementación debido a que al leventar la señal '*ready*', la nueva instrucción que venga modificará dicha señal. Y la señal de Hit es necesaria para saber si que hay que hacer Fetch del bloque antes de escribirlo o simplemente escribir en Cache y MD.

ESTADO	SEÑALES ACTIVADAS
WRITE MD	**Bus_req**; If RE = '1' and (hit = '1' or internal_addr = '1') then **ready**, **inc_r** If internal_addr = '1' then **mux_output = '10'** else **mux_output = '00'**
WRITE MD SEND BLOCK ADDR	**Frame**, **MC_send_addr_ctrl**, **bufferizado**, **MC_bus_Read**, **block_addr** If RE = '1' and (hit = '1' or internal_addr = '1') then **ready**, **inc_r** If internal_addr = '1' then **mux_output = '10'** else **mux_output = '00'**
WRITE MD SEND WORD ADDR	**Frame**, **MC_send_addr_ctrl**, **bufferizado**, **MC_bus_Read**, **block_addr = '0' MC_bus_Write** If RE = '1' and (hit = '1' or internal_addr = '1') then **ready**, **inc_r** If internal_addr = '1' then **mux_output = '10'** else **mux_output = '00'**
WRITE MD BRING BLOCK	**Frame**, **MC_send_data**, **Last_word = Last_word_block**, **mux_origen = "01"** If RE = '1' and (hit = '1' or internal_addr = '1') then **ready**, **inc_r** If internal_addr = '1' then **mux_output = '10'** else **mux_output = '00'**
WRITE MD SEND WORD	**Frame**, **Last_word**, **MC_send_data**, **mux_origen = '11'** If RE = '1' and (hit = '1' or internal_addr = '1') then **ready**, **inc_r** If internal_addr = '1' then **mux_output = '10'** else **mux_output = '00'**

Cuadro 6: Tabla de señales de comportamiento de estado

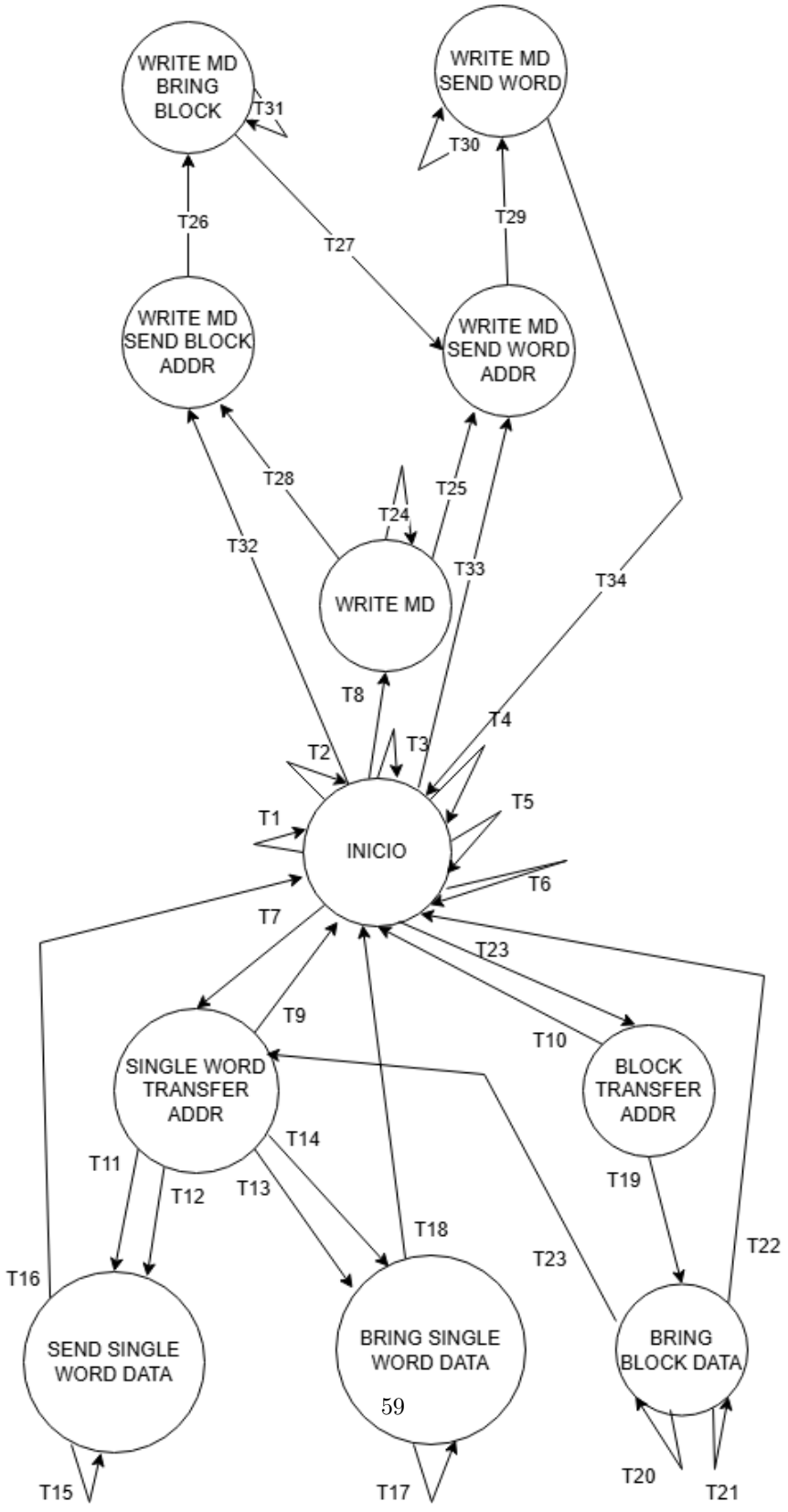

Figura 27: Autómata ampliado para el apartado opcional

TRANSICIÓN	CONDICIONES	SEÑALES ACTIVADAS
T8	**WE = '1' AND addr_non_cacheable= '0'**	**load_registros, Bus_Req,ready,**
T24	**Bus_Grant = 0**	**NINGUNA**
T25	**(Bus_Grant = 1 and registro_hit_output = '1')**	**NINGUNA**
T26	**Bus_DevSel = '1'**	**ready, load_addr_error, next_error_state = memory_error**
T27	**NINGUNA (Siempre se toma)**	**count_enable, inc_w, MC_tags_WE y MC_WE1 si via_2_rpl o MC_WE0 si via_2_rpl = 0**
T28	**(Bus_Grant = 1 and registro_hit_output = '0')**	**NINGUNA**
T29	**Bus_DevSel = '1'**	**NINGUNA**
T30	**Bus_TRDY = 0**	**NINGUNA**
T31	Bus_TRDY = 0 OR Bus_TRDY = '1' AND last_word_block = 0	En caso de Bus_TRDY = 1 **inc_w,count_enable** y **MC_WE1 si via_2_rpl** o **MC_WE0 si via_2_rpl = 0**
T32	**Bus_Grant = 1 and Hit = 0**	**load_registros, Bus_Req,ready**
T33	**Bus_Grant = 1 and Hit = 1**	**load_registros, Bus_Req,ready**
T34	**Bus_TRDY = 1**	**NINGUNA**

La manera en la que se va a tratar el problema es la siguiente. Se van a añadir 5 estados como extensión, el primero de ellos va a ser: ***write_md***, el cual es un estado de espera para esperar la señal *Bus_Grant*. ¿Por qué se ha considerado necesario este estado?. Bien, pues la respuesta a esa pregunta es: para poder empezar a procesar Read Hits. En caso de no hacer este estado, tendríamos que esperar en *Inicio* a que el árbitro nos diese el grant y entonces a partir de ahí, sí que podríamos activar *ready* y aceptar la siguiente instrucción. Con la decisión de crear este estado, viene la necesidad de la creación de el registro de 1 bit para guardar el valor de *Hit*, de esta manera, podemos hacer un único estado de espera, en el cual, cuando llegue el *Bus_Grant*, se pueda ir a un estado u a otro.

Las señales que se activan en estos estados nuevos, son casi idénticas a los otros estados ya creados que transfieren dirección de bloque / palabra y traen un bloque o mandan una palabra. Sin embargo, para el tratamiento de Read Hits, es necesaria la adición de nueva lógica que trate el problema planteado. Por ello se añaden al código la siguientes líneas:

```
1
2   if (RE = '1' and (hit = '1' or internal_addr = '1')) then
3         ready <= '1';
4         inc_r <= '1';
5         if (internal_addr = '1') then
6             mux_output <= "10";
7         else
8             mux_output <= "00";
9         end if;
10    end if;
```

Listing 7: COMPLETAR_UC_MC_2025_WT_ciclo_arb.vhd

De manera que, si hay un **Read Hit**, ya sea en registros internos o en memoria cache, se sacará el dato y se seguirá a la siguiente instrucción. En el momento en el que llegue algo que no sea un Read o que sea un Read que haga miss, no se tratará y hasta que no se llegue al estado de Inicio, esa instrucción seguirá esperando a que se vuelva a Inicio.

Otra modificación que se realiza es que al mandar la palabra, cuando llega *Bus_TRDY* en el estado de *write_md_send_word*, **no se activa la señal *ready***, porque sería un error intentar indicar que nuestra instrucción ha terminado cuando desde un primer momento el MIPS ya la daba por terminada ya que al transicionar al estado de espera *write_md*, se activa la señal *ready*. También se contempla el caso en el que pudiese llegar ***Bus_Grant*** en el mismo ciclo de la petición con lógica adicional en el estado de Inicio para ir directamente a *write_md_send_word_addr* o a *write_md_send_block_addr* dependiendo de si *Hit* es 0 o 1.

Para **recopilar**: se han añadido varias señales nuevas a la UC, *load_registros* y *bufferizado*, que salen de la UC a la MC; y *registro_hit_output*, que como su nombre indica, es el valor guardado en el nuevo registro Hit que guarda si ha habido un Hit o no en memoria al llegar una instruccion sw.

En la lógica de la MC, ha sido necesario cambiar el funcionamiento de ***MC_Bus_ADDR***, ya que necesitamos modificar de donde viene la dirección que se manda al bus. En nuestro nuevo caso, se mandará desde el output del nuevo registro que guarda la direcci por ello añadimos varias señales nuevas para manejar esta caso:

```
1       -- Modificacion
2   Internal_MC_Bus_ADDR_Registro <=registro_addr_output(31 downto 2)&"00"
3                                   when block_addr ='0' else
4                   registro_addr_output(31 downto 4)&"0000";
5
6   MC_Bus_ADDR <=Internal_MC_Bus_ADDR when ( bufferizado = '0' ) else
7              Internal_MC_Bus_ADDR_Registro;
8
9   MC_Bus_data_out <=  Din when (addr_non_cacheable = '1') else
10            registro_dato_output when (bufferizado = '1') else
11            MC_Dout; -- is used to send the data to be written
```

Listing 8: COMPLETAR_UC_MC_2025_WT_ciclo_arb.vhd

Para la resolución de este caso se usa la señal ***bufferizado***, que se activa en las etapas de *WRITE_SEND_...._ADDR*. Que le indicará a MC si la dirección que tiene que mandar sale del Registro de este apartado opcional o si proviene directamente del MIPS.

Por último, la señal *load_registros*, hace exactamente lo que el nombre sugiere, activa la señal *load* de los 3 registros añadidos.

```
    -- Modificacion
Registro_addr: reg generic map (size => 32)
          port map ( Din => ADDR, clk => clk, reset => reset
              , load => load_registros, Dout =>
              registro_addr_output);
hit(0) <= hit0 or hit1;
bit_hit_output <= registro_hit_output(0);
Registro_hit: reg generic map (size => 1)
          port map ( Din => hit, clk => clk, reset => reset,
               load => load_registros, Dout =>
              registro_hit_output);
Registro_dato: reg generic map (size => 32)
          port map ( Din => Din, clk => clk, reset => reset,
               load => load_registros, Dout =>
              registro_dato_output);
```

Listing 9: COMPLETAR_UC_MC_2025_WT_ciclo_arb.vhd

12.2. Pruebas

Para la realización de las pruebas, se ha aprovechado uno de los tests previamente hechos y descritos para comprobar el correcto funcionamiento de la memoria cache y su interacción con el bus. Vamos a analizar las siguientes instrucciones del *TestMD.asm* :

sw r1,16(r0) ========== 0C010010

lw r3,20(r0) ========== 08030014

sw r3,16(r0) ========== 0C030010

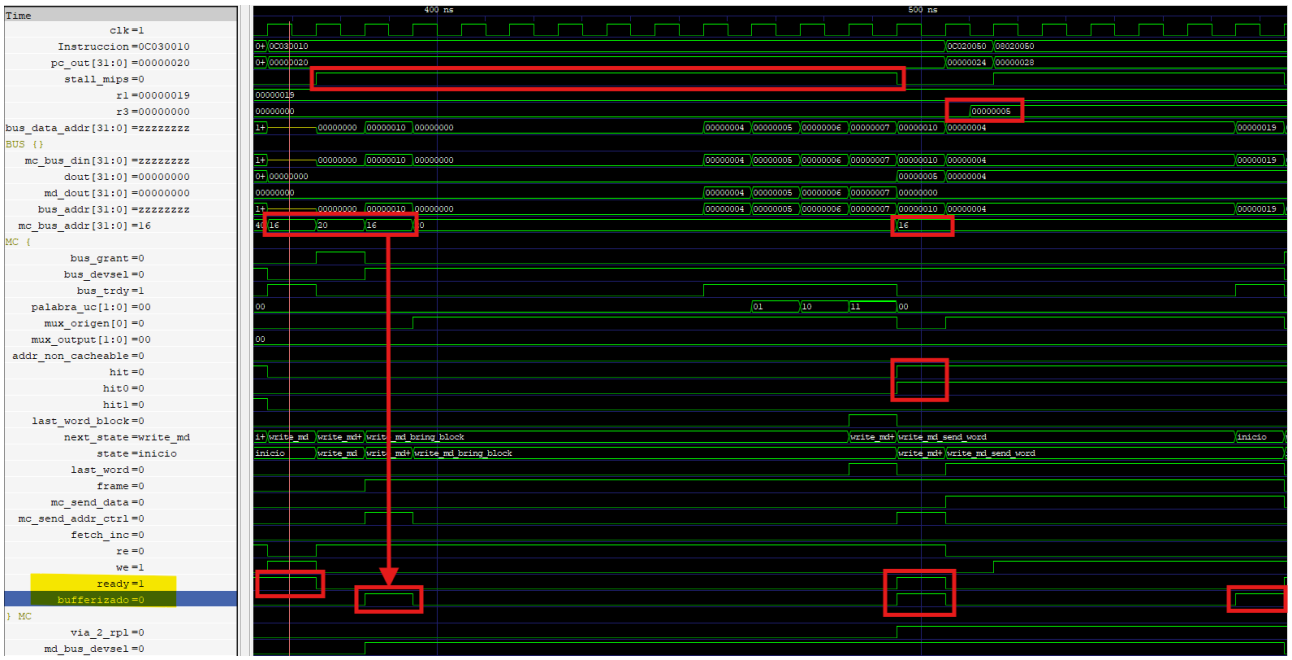

Figura 28: Primer set de señales para el test del apartado opcional

En este test podemos observar varias señales interesantes, las cuales son las 2 activaciones de la señal *ready*, la primera de ellas al transicionar al estado de *write_md*, con la que llega a la MC la señal *lw r3,20(r0)*, que podemos comprobar viendo que la *mc_bus_addr* cambia a 20, también podemos comprobar el correcto funcionamiento de la señal *bufferizado* ya que en el siguiente ciclo vuelve a estar a 16 la señal, que es cuando se manda la dirección por el bus. También se activa la señal a la hora de mandar la palabra en el último ciclo de *write_md_send_word*, para que el *MC_bus_data_out* tome el valor del registro en vez de el dato recién mandado por el MIPS. Algo interesante de este test es que al traer el bloque y escribir los tags se produce el *Hit* sobre la instrucción *lw r3,20(r0)*, por lo que al ciclo siguiente de escribir los tags ya se produce ese Read Hit que buscamos para este apartado. Podemos ver también en consecuencia la correcta escritura del registro r3 en el ciclo siguiente. Al producirse este Read Hit, se levanta la señal *ready*, que trae la siguiente instrucción *sw r3,16(r0)*, que produce un Hit, pero al ser un *Write*, se tendrá que esperar a que termine la instrucción de Write actual.

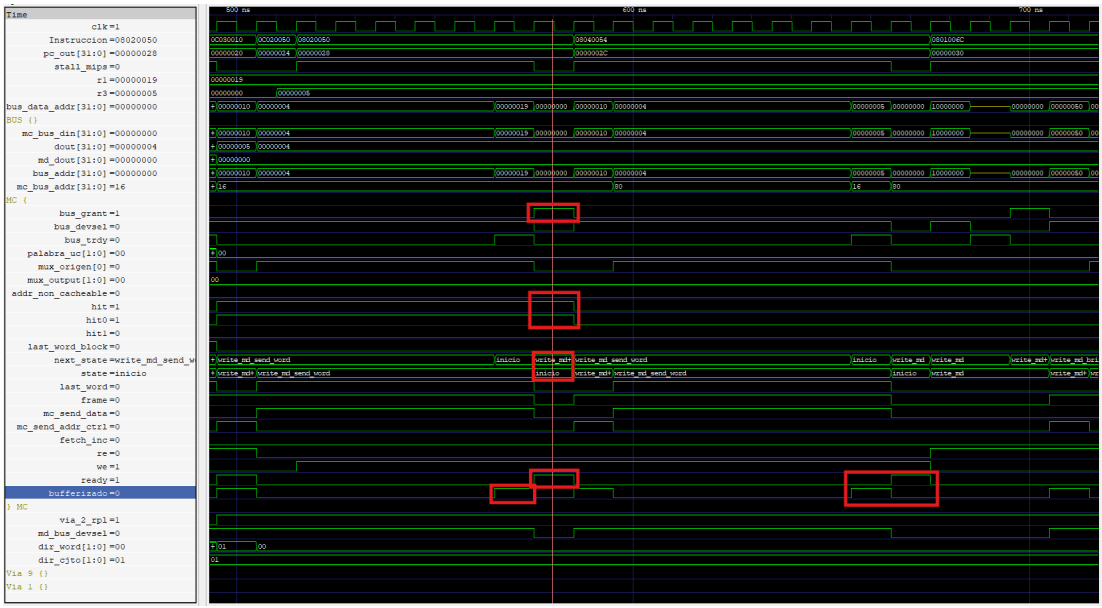

Figura 29: Segundo y último set de señales para el test del apartado opcional

En esta segunda parte sucede también un ejemplo que sigue comprobando el correcto funcionamiento de este apartado, y es que al terminar nuestra primera instrucción, *sw r1,16(r0)* sucede algo interesante y es que el árbitro nos concede *Bus_Grant* en el mismo ciclo en el que hacemos el Request, lo cual como podemos ver hace que desde *Inicio* se vaya directamente a *write_md_send_word_addr* debido a que la instrucción que se va a ejecutar *sw r3,16(r0)* produce un Hit. Se puede observar igualmente el correcto funcionamiento de la señal *bufferizado*, de manera que la dirección se envía de forma correcta, al igual que al mandar la palabra. Como la siguiente instrucción sigue siendo un WE como vemos en la señal no se vuelve a activar *ready* hasta llegar a *Inicio*

12.3. Casos de utilidad

Dado que tenemos 2 formas en las que se puede producir un write. Tenemos varias posibilidades a estudiar, aunque lo que se puede decir desde un primer momento es que esta modificación puede beneficiar bastante en cuanto a acortar los ciclos de según que programas.

Hay algo de casuística involucrada en las instrucciones posteriores a estos Writes. Por ejemplo, de producirse un *Write Miss*, si las siguientes instrucciones produjesen un Read Hit sobre la vía que va a ser reemplazada en cache, los Read Hits seguirían llegando hasta que no se escribiesen los TAGs en cache. El problema, es que por como está implementada la UC, el contenido de las 3 primeras palabras recién traídas de MD del bloque se cambian sin actualizar el TAG del bloque en cache. Es por ello, que solo podemos garantizar que los Read

Hit sean válidos si llegan en los ciclos de *write_md* o *write_md_send_block_addr* (por la implementación que se ha realizado). Es por ello, que en el caso de estar en *write_md_bring_block* y nos llegue un Read Hit, si coinciden la vía del hit con el valor de *via_2_rpl*, ignoraremos el hit y esperaremos hasta que se produzca el hit. No necesitamos esta logica adicional en el resto de estados, ya que al salir del estado *write_md_bring_block*, ya se habrán actualizado los TAGs y lo que antes activaba la señal Hit, ya no lo hará.

Si que se plantea la opción de, con lógica adicional en el estado de *bring_block*, aprovechar los ciclos en los que no se ha traído ninguna palabra del bus, cosa que haríamos comparando las señales *palabra_UC* (es decir, que sea "00") y *bus_TRDY* (también sea 0) en caso de llegar ese Read Hit. Se ha decidido no intentar implementar y testear completamente esta opción debido a la falta de tiempo para debidamente realizar esta implementación.

Por ello, se puede aprovechar el caso de Read Hits sobre vías reemplazadas siempre y cuando estén a una distancia de 1-4 * de la instrucción de Write Miss que las precede.

* Está distancia es el rango más amplio observado en simulación en el que una instrucción sw con la implementación de la UC de la MC en el que la instrucción va desde Inicio a write_md_bring_block en caso de miss.

Este caso es el más problemático que nos podemos encontrar, puesto que si el Read Hit no se produce en el mismo set, o, de ser en el mismo set, se produce en la vía que no va a ser reemplazada, no hay conflicto de ningún tipo y se pueden aprovechar los 5 o 19 ciclos que tarda un Write Hit/Miss respectivamente.

13. ANEXO 3. APARTADO OPCIONAL : ETHICAL HACKING

13.1. Preludio: Acerca del planteamiento para el apartado en la sección de incidencias

En la sección de incidencias de la asignatura, se actualizó el apartado para el *Hacking Ético*, planteando un programa que degradase el rendimiento de la cache a base de invalidaciones producidas por instrucciones *lw_inc*. Ahora bien, el equipo no se cayó en cuenta de la existencia de este comentario, por lo que el problema se planteo desde el punto de vista de una cantidad ingente de reemplazamientos, sin permitir el acceso rápido que proporciona la cache y cargando palabras de más.

Por una parte, sí que se debe mencionar que las medidas planteadas **pueden paliar el problema de las invalidaciones**, puesto que indirectamente estas producirán indirectamente posteriores misses, los cuales se tienen en cuenta en el cálculo del *umbral de eficiencia*.

Por otra parte es cierto que el planteamiento del problema podría haber llevado a correderos diferentes:

- Se habría realizado un programa en el que se cargasen bloques de la *MD*, sobre los que se realizarán posteriormente operaciones (tanto de escritura como lectura) que produjesen un *hit*, pero sobre bloques invalidados debido a *lw_inc*'s. El resultado del programa comportaría unas métricas tan negativas como el presentado posteriormente.
- En cuanto a las soluciones, podría haberse planteado la desactivación de la MC como en este apartado, aunque parece algo drástico. Alternativamente, podría haberse implementado un mecanismo que contabilizase *hits invalidados*, y que ante un número elevado activase otra lógica en la MC para mantener la coherencia entre la MC y la MD (un sumador que al cargar la palabra de la MD almacene el valor incrementado también en la vía correspondiente).

Ahora bien, dado a que el apartado ya había sido terminado para el momento de la lectura del comentario, y que apenas quedaba 1 día para el *deadline*; no se ha profundizado más en esta idea. De todos modos, se considera que la investigación realizada en el presente anexo también puede ajustarse dentro de la gestión de degradaciones debido a un mal uso de la cache.

13.2. Descripción

La finalidad de este apartado es la diametralmente opuesta al test de integración elaborado: **explorar el comportamiento más ineficiente del sistema *MC***. Los esfuerzos invertidos en esta sección se centran en dos tareas:

1. Elaborar un programa que 'hackee' la Memoria Cache provocando un número ingente y constante de *misses*.

2. Estudiar posibles implementaciones adicionales para paliar las situaciones que puedan comprometer la eficiencia del sistema de memoria Cache.

13.3. Programa de *Ethical Hacking*

El programa propuesto para saturar a la cache de *misses* tiene, a alto nivel, el objetivo de calcular $16 * 4 * i \ \forall i = 1, 2, 3, 4$. Ahora bien, esta operación se realizará de una manera extremadamente peculiar, para alcanzar el objetivo real del mismo programa.

De este modo, se han escogido, para cada uno de los cuatro sets de la MC, 4 direcciones de bloque (por los primeros 4 valores del campo *tag*):

1. set 1: *0x0000, 0x0040, 0x0080, 0x00C0.*

2. set 2: *0x0010, 0x0050, 0x0090, 0x00D0.*

3. set 3: *0x0020, 0x0060, 0x00A0, 0x00E0.*

4. set 4: *0x0030, 0x0070, 0x00B0, 0x00F0.*

A lo largo de todo el programa, una vez se realice un acceso a una palabra de un determinado bloque, no se podrá acceder al mismo hasta haber realizado un acceso al resto de bloques seleccionados. Esto llevará a que dicho bloque haya sido expulsado de la cache para el momento en el que se vuelva a acceder a él, provando con toda seguridad un fallo; y un nuevo reemplazamiento (nótese que esto mismo se podía haber conseguido con 3 bloques por conjunto, pero con 4 la vía en la que se encontrará un bloque será la misma siempre, mejorando la trazabilidad del programa).

Así pues, el programa iterará 16 veces, y en cada iteración k irá almacenando en la primera palabra de todos los bloques seleccionados de la via i el valor de (i + 1) * k. Finalmente, para cada vía, se leerá y sumará el contenido de la primera palabra de todos los bloques (16*(i+1)*4), almacenándose en la primera palabra de cada bloque escogido para el conjunto 1, sucesivamente. Por las políticas que se han descrito anteriormente, **todos los accesos a la MC provocarán un *miss*** (excepto 3 para cargar datos iniciales), lo que generará un total de $misses = 16 * 4 * 4 \ misses_{bucle} + 5 * 4 \ misses_{final} = 276 misses$.

A continuación, se adjunta el programa ***ethicalHacking.asm***, cuyos contenidos iniciales para la *MD* y *MI* se pueden encontrar comentados en *RAM_128_32_P2_2025_bucle_lectura.vhd* y *RAM_I_test_exceptions.vhd*, respectivamente:

```
;; TEST PARA REALIZAR UN "HACKING  TICO " EN EL SISTEMA DE MEMORIA CACHE
;; Para ello, se van a saturar todos los conjuntos de ambas v as de la memoria cahce
;; Se escogeran 4 bloques por conjunto, los cuales se ir n cargando sucesivamente. De este
;; modo, los pares de bloques se ir n expulsando entre s  de la cache sin dar opci n a
;; que se de siquiera un hit.

;; Contenido de la MD inicial:
;; 1, 2, 3, 4

;; NOTA : Bloques por v a seleccionados arbitrariamente:
;; SET 0: 0x0000, 0x0040, 0x0080, 0x00C0
;; SET 1: 0x0010, 0x0050, 0x0090, 0x00D0
;; SET 2: 0x0020, 0x0060, 0x00A0, 0x00E0
;; SET 3: 0x0030, 0x0070, 0x00B0, 0x00F0

;; En cada v a i, el programa calcula 4* 16 * i+1 de una manera curiosa
;; Carga de atributos de la MD:
lw r0, 0(r0); @0x0000 (cargar el valor 1)
            ; READ MISS OBLIGATORIO -> VIA 0 SET 0
lw r1, 4(r9); @0x0004 (cargar el valor 2)
             ; READ HIT -> VIA 0 SET 0
lw r2, 8(r9); @0x0008 (cargar el valor 3)
             ; READ HIT -> VIA 0 SET 0
lw r3, 12(r9); @0x000C (cargar el valor 4)
                ; READ HIT -> VIA 0 SET 0
add r4, r3, r3; @0x0010
add r4, r4, r4; @0x0014 (r4 = 4*4 = 16)

add r5, r0, r5 ; @0x0018
add r6, r1, r6 ; @0x001C
add r7, r2, r7 ; @0x0020
add r8, r3, r8 ; @0x0024
;; BUCLE PRINCIPAL

;; ------- SET 0 -------
INI: sw r0, 0(r9); @0x0028 (almacenar el valor 1 * k en la direcci n 0 x00000000)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 0  (excepto en la primera iteraci n, que es hit)
sw r0, 64(r9); @0x002C (almacenar el valor 1 * k en la direcci n 0 x00000040)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 0 (excepto en la primera iteraci n, que es obligatorio)

sw r0, 128(r9); @0x0030 (almacenar el valor 1 * k en la direcci n 0 x00000080)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 0

sw r0, 192(r9); @0x0034 (almacenar el valor 1 * k en la direcci n 0 x000000C0)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 0
```

Listing 10: ethicalHacking.asm

```
;; ------- SET 1 -------
sw r1, 16(r9); @0x0038 (almacenar el valor 2 * k en la direcci n 0
    x00000010)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 1 (excepto en la
                    primera iteraci n , que es obligatorio)
sw r1, 80(r9); @0x003C (almacenar el valor 2 * k en la direcci n 0
    x00000050)
                ;  WRITE MISS CONFLICTO -> VIA 1 SET 1  (excepto en la
                    primera iteraci n , que es obligatorio)
sw r1, 144(r9); @0x0040 (almacenar el valor 2 * k en la direcci n 0
    x00000090)
                ;  WRITE MISS CONFLICTO -> VIA 0 SET 1
sw r1, 208(r9); @0x0044 (almacenar el valor 2 * k en la direcci n 0
    x000000D0)
                ;  WRITE MISS CONFLICTO -> VIA 1 SET 1

;; ------- SET 2 -------
sw r2, 32(r9); @0x0048 (almacenar el valor 3 * k en la direcci n 0
    x00000020)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 2 (excepto en la
                    primera iteraci n , que es obligatorio)

sw r2, 96(r9); @0x004C (almacenar el valor 3 * k en la direcci n 0
    x00000060)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 2  (excepto en la
                    primera iteraci n , que es obligatorio)
sw r2, 160(r9); @0x0050 (almacenar el valor 3 * k en la direcci n 0
    x000000A0)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 2
sw r2, 224(r9); @0x0054 (almacenar el valor 3 * k en la direcci n 0
    x000000E0)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 2

;; ------- SET 3 -------

sw r3, 48(r9); @0x0058 (almacenar el valor 4 * k en la direcci n 0
    x00000030)
                ; WRITE MISS CONFLICTO -> VIA 0 SET 3 (excepto en la
                    primera iteraci n , que es obligatorio)
sw r3, 112(r9); @0x005C (almacenar el valor 4 * k en la direcci n 0
    x00000070)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 3 (excepto en la
                    primera iteraci n , que es obligatorio)

sw r3, 176(r9); @0x0060 (almacenar el valor 4 * k en la direcci n 0
    x000000B0)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 3
sw r3, 240(r9); @0x0064 (almacenar el valor 4 * k en la direcci n 0
    x000000F0)
                ; WRITE MISS CONFLICTO -> VIA 1 SET 3
```

Listing 11: ethicalHacking.asm

```
add r0, r0, r1; @0x00D4
add r0, r0, r2; @0x00D8
add r0, r0, r3; @0x00DC

sw r0, 128(r9); @0x00E0 (almacenar el valor 2 * 16 *4 en la direcci n 0
    x000000020)
                    ;; WRITE MISS CONFLICTO -> VIA 0 SET 0

lw r0, 48(r9); @0x00E4 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 0 SET 3
lw r1, 112(r9); @0x00E8 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 1 SET 3
lw r2, 176(r9); @0x00EC (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 0 SET 3
lw r3, 240(r9); @0x00F0 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 1 SET 3

add r0, r0, r1; @0x00F4
add r0, r0, r2; @0x00F8
add r0, r0, r3; @0x00FC

sw r0, 192(r9); @0x0100 (almacenar el valor 2 en la direcci n 0
    x00000010)
                    ;; WRITE MISS CONFLICTO -> VIA 1 SET 0

beq r0, r0, 65535; @0x0104 (si no , vuelvo a iterar)
;; FIN DEL PROGRAMA
```

Listing 12: ethicalHacking.asm

```
add r0, r0, r1; @0x00D4
add r0, r0, r2; @0x00D8
add r0, r0, r3; @0x00DC

sw r0, 128(r9); @0x00E0 (almacenar el valor 2 * 16 *4 en la direcci n 0
    x000000020)
                    ;; WRITE MISS CONFLICTO -> VIA 0 SET 0

lw r0, 48(r9); @0x00E4 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 0 SET 3
lw r1, 112(r9); @0x00E8 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 1 SET 3
lw r2, 176(r9); @0x00EC (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 0 SET 3
lw r3, 240(r9); @0x00F0 (cargar el valor 4 * 16)
                    ;; READ MISS CONFLICTO -> VIA 1 SET 3

add r0, r0, r1; @0x00F4
add r0, r0, r2; @0x00F8
add r0, r0, r3; @0x00FC

sw r0, 192(r9); @0x0100 (almacenar el valor 2 en la direcci n 0
    x00000010)
                    ;; WRITE MISS CONFLICTO -> VIA 1 SET 0

beq r0, r0, 65535; @0x0104 (si no , vuelvo a iterar)
;; FIN DEL PROGRAMA
```

Listing 13: ethicalHacking.asm

Los resultados de la simulación fueron absolutamente negativos, hasta tal punto que el contador de *misses* desbordó, como se muestra a continuación:

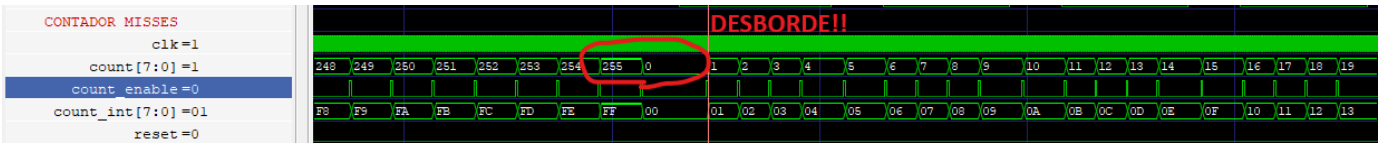

Figura 30: DESBORDE en el contador *count_m*

.

Y es que los bloques con tags 0-2 y 1-3 se encuentran en un incesante re-emmplazamiento, como se puede observar a continuación:

Figura 31: REEMPLAZAMIENTO CONTINUO

.

Esto lleva a un *performance* fatídico: se necesitan 5561 ciclos para la terminación del programa; y para realizar 276 escrituras/lecturas de palabras, se ha necesitado transferir de MD a MC 1380 palabras (proporción 4/1 por siempre leer bloque).

13.4. ¿Cómo paliar el problema?

El programa presentado anteriormente tenía un comportamiento particular: el recorrido de la MD no era secuencial; y el orden seguido daba lugar a una, aunque ligera, ininterrumpida competencia por los bloques de la cache. Y aunque el comportamiento de este programa sea *demasiado* particular, no se debe obviar que la heterogeneidad, versatilidad (y casi infinidad) de los programas posibles generan grandes 'áreas' particulares de comportamientos, muchos de los cuales pueden no encajar bien con el sistema *MC* diseñado. Por tanto, la primera (y muy importante) medida para minimizar la situaciones que produzcan un pobre rendimiento de la *MC* es **analizar profusamente el contexto de uso del sistema**. En caso de que los resultados de la investigación lleven a una alta estimación de programas *ineficientes*, se debería considerar la sustitución de la arquitectura de la MC, por muy radical que parezca la medida.

Ahora bien, aunque la previsión de programas puramente *ineficientes* sea baja, siempre podrán existir fragmentos de un programa que se comporten peor, y se considera una medida muy positiva que la *MC* sea capaz de detectar dinámicamente estas situaciones y responder en consecuencia. Respecto a la **detectabilidad de situaciones de saturación de la cache por el sistema original**, cabe mencionar que es bastante escaso. En este sentido, existen contadores para operaciones de escritura y palabra sobre la cache, pero no un contador de accesos a direcciones cacheables que poder comparar con el contador de *misses*.

LLegados a este punto, cabe preguntarse : **¿Cómo determinar cuándo el rendimiento de un programa es negativo? ¿Cúal es el umbral?**. El equipo trató de resolver este dilema a través de la comparativa entre el tiempo empleado para cargar un bloque y el tiempo en leer palabras:

- Cuando se produce un fallo, se añade un tiempo adicional de $CrB(MD) - L = 3$ *ciclos*, que es el de cargar las 3 palabras restantes del bloque.
- Realmente, tan solo se '*aprovecha*' el acceso rápido de la cache en las operaciones de lectura, dado que por la política *Write-through* toda escritura equivale a una transferencia de bus. Dadas las características de la MD, $CrW(MD) = CwW(MD) = 8$ *ciclos*, por lo que:

$$ciclosAhorradosPorReadHits = CrW(MD) - 1 = 7ciclos$$

- Dados los dos puntos desarrollados, se puede aducir que **para que una transferencia de bloque sea rentable, debe venir acompañada de al menos un read hit**. Deben descartarse además aquellos read *hits* que provienen de read *misses*, los cuales ya han sido tenidos en cuenta en los cálculos anteriores.

13.4.1. Contramedidas llevadas a cabo por el equipo

Ahora que se ha establecido un umbral de eficiencia para el uso de la *MC* por un programa, se debe decidir cual debe ser la respuesta apropiada. Ante ello, una posible opción puede venir por intentar cambiar en tiempo de ejecución la arquitectura de la MC. Pero esto se considera **bastante complejo**, puesto que no solo queda implicada las implementaciones adicionales a nivel *HW*, sino la responsabilidad lógica de decidir cuál debe ser la arquitectura destino.

La opción que se ha considerado más sencilla es la de **desactivar el sistema de la *MC***. De este modo, cuando se alcance un estado del programa en el que

$misses >= (readHits - readMisses) * 1{,}15$ (se descuentan los read hits provenientes de un read miss, y se añade un "margen"de 15 % de ineficiencia)

el sistema de memoria cache (las vías) se desactivará (***Cache Bypassing***), y las operaciones de escritura y lectura se tratarán a nivel de palabra siempre (como en el caso de la *MD_Scratch*). Además, se decidió esperar a que el número de

misses sea superior a 25 (a que los efectos negativos puedan ser notables).

Ahora, bien, **¿Qué ocurre si la ineficiencia no proviene a nivel de programa, sino a nivel de bloque de programa, y llega un punto en el que el sistema cache implica un mejor rendimiento?**. Ante ello, el equipo decidio introducir la posibilidad de reactivar el sistema cache en estos casos. Ahora bien, esto plantea un problema, y es el de tomar métricas de la MC siendo que esta se encuentra desactivada. Respecto a esto, el equipo se inspiró en las siguientes técncas:

- ***Ghost Caching + Tag-Only Monitoring***: No se desactivarán totalmente las vías de la *MC*, si no que se mantiene el sistema de tags. De este modo, se empleará una especie de *cache fantasma*, simulando el comportamiento de búsqueda y reemplazo de bloques (poder contabilizar hits y misses con el tag presente), pero sin consumir la energía ni el tiempo de escribir datos ni traer bloques enteros de la MD.
- ***Cache toggling***: El sistema de vías de la cache se irá adaptando a las condiciones del programa, siempre que este alcance una estabilidad. Es interesante considerar en este punto la posible introducción del *set duelling*. Por ejemplo, en vez de desactivar ambas vías, podría haberse cambiado lal política de reemplazo de la vía 1, y tomar métricas en paralelo.

Para detectar en tiempo de ejecución situaciones ineficientes, y con el fin de poder gestionar la correcta activación de la cache, se introdujeron los siguientes elementos en vhd:

- Contador ***cont_readMisses***: Contador para contar misses debidos a *lw*.

```
1
2 cont_readMisses: counter generic map (size => 8) -- Cuenta
      fallos en lectura
3   port map (clk => clk, reset => reset, count_enable =>
        inc_rm, count => rm_count);
```

Listing 14: MC_DATOS_2025_WT.vhd

- señal de salida MC - **inc_rm**: Para controlar el anterior contador.

```
1
2     when block_transfer_addr =>
3             ...
4       if (Bus_DevSel = '0') then -- Ning n dispositivo
            reconoce la direcci n -> ERROR
5             ...
6       else -- Se procede a una transferencia de bloque
7         inc_m <= '1'; -- Se incrementa el n mero de misses
8         if (RE = '1') then -- Incrementar n mero de read
            misses
```

```
9           inc_rm <= '1'; -- Se incrementa el n mero de read
                misses
10        end if;
11        next_state <= bring_block_data;
12      end if;
```

Listing 15: COMPLETAR_UC_MC_2025_WT_CICLO_ARB.vhd

- **Contadores espejo y señales *enable* para originales**: Para mantener métricas de la cache ajustadas a la realidad, se ha decidido duplicar los contadores de las métricas originales (*write, miss, invalidations y reads*). De este modo, estos contadores *espejo* contabilizan como si la *MC* estuviese siempre activadan (serán los empleados en medidas de eficiencia), mientras que los originales solo se ponen en funcionamiento cuando la cache está activa. Se muestra un ejemplo para el caso ed *cont_m*:

```
1
2  cont_m: counter generic map (size => 8)
3      port map (clk => clk, reset => reset, count_enable =>
           enable_cont_m, count => m_count);
4  enable_cont_m <= '1' when (MC_desactivada = "0" and inc_m =
       '1') else '0'; -- Se activan los contadores cuando la
       cache esta activada
5  ...
6  cont_m_mirror: counter generic map (size => 8)
7      port map (clk => clk, reset => reset, count_enable =>
           inc_m, count => m_count_mirror);
```

Listing 16: MC_DATOS_2025_WT.vhd

- **Contador del número de misses periódico**: Para evitar inmediatos cambios de estado en programas volátiles, se introduce un contador *cont_mPer*, que cuenta los misses desde que el inicio de una fase de *MC* activada:

```
1
2  -- Se al que se activa cuando  el n mero de misses es
       mayor que 25 desde que se accedi  a este estado es mayor
        que 25.
3
4  mayor_25 <= '1' when (mPer_count > x"19") else '0';
5
6  -- Contador de misses peri dico (se resetea cada vez que se
        vuelve a ciclo normal):
7  cont_mPer: counter generic map (size => 8)
8      port map (clk => clk, reset => reset_mPer, count_enable
           => inc_m, count => mPer_count);
9
```

```
reset_mPer <= '1' when (cambiar_Estado_cache = '1' and
    MC_desactivada="1") or reset = '1' else '0'; -- Se
    resetea el contador de misses cada vez que se vuelve a
    activar la MC
```

Listing 17: MC_DATOS_2025_WT.vhd

- **Contador de accesos a MD con cache desactivada**: Por la misma razón que en el punto anterior, se contabiliza el número de accesos a memoria a partir de una transición a una *MC* desactivada. De este modo, si se vuelve a un *comportamiento eficiente* de la cache han de esperarse 15 accesos para poder volver a activarla.

```
-- Contador de accesos a MD con MC desactivada
cont_accMDBruto: counter generic map (size => 8)
    port map (clk => clk, reset => reset_cont_accMD,
        count_enable => inc_accMd, count => accMD_count);
accMd_mayor_15 <= '1' when (accMD_count > x"0F") else '0';
    -- Se activa cuando el n mero de accesos a MD con MC
    invalidada es mayor a 15

reset_cont_accMD <= '1' when (cambiar_Estado_cache = '1' and
     MC_desactivada="0") or reset = '1' else '0'; -- Se
    resetea el contador de accesos a MD cada vez que se
    vuelve a activar la MC
```

Listing 18: MC_DATOS_2025_WT.vhd

- **Señal *mc_ineficiente***: Se activa cuando el número de *misses* es excesivo, acorde a los cálculos realizados anteriormente.

```
mc_ineficiente <= "1" when unsigned(m_count_mirror) >
    ( (unsigned(r_count_mirror) - unsigned(rm_count)) * 115)
        / 100 else "0";
```

Listing 19: MC_DATOS_2025_WT.vhd

- **Registro para estado de activación de la cache**: Se introduce un registro *Estado_Cache* que almacenará el estado de la cache (Dout="1"si y solo si está desactivada). Tiene como entrada *mc_ineficiente* (si la cache es ineficiente), se actualiza con una nueva señal *cambiar_Estado_Cache* (para retardo inicial de accesos).

```
-- Se al para gestionar un cambio de estado retardado
cambiar_Estado_cache <= '1' when ((MC_desactivada = "0" and
    mc_ineficiente = "1" and mayor_25 = '1') or (
    MC_desactivada = "1" and mc_ineficiente = "0" and
    accMd_mayor_15 = '1')) and (mem_ready = '1') else '0'; --
     Se espera a que el acceso a memoria se termine con
    mem_ready = '1' para evitar transiciones en medio de
    transferencias.
```

```
3
4  -- Registro con el estado de la cache
5  Estado_cache: reg generic map (size => 1)
6            port map (  Din => mc_ineficiente, clk => clk,
                reset => reset, load => cambiar_Estado_Cache,
                Dout => MC_desactivada);
7  MC_desactivada_logical <= MC_desactivada(0); -- Se usa para
       indicar que la cache esta desactivada
```

Listing 20: MC_DATOS_2025_WT.vhd

- **Nueva señal de entrada MC - MC_desactivada**: Señal de entrada que indica si la cache está desactivada. Importante incluirla en la *lista de sensibilidad* del proceso *OUTPUT_DECODE*, pues es parte del comportamiento *Mealy* del autómata.
- **Nueva lógica para la gestión de tags**: Surge de la inconsistencia producida al mantener el sistema de tags pero no de datos cuando la MC está desactivada. Se decidió que al volver a un estado total de la *MC*, **todos los bloques de la cache se invaliden**. Para hacer esto posible, se añadió a ***Via_2025_WT*** una nueva señal de entrada: *invalidar_all*.

```
1  -- SE A ADE UNA ENTRADA PARA ETHICAL HACKING: INVALIDAR
       TODOS LOS BLOQUES AL VOLVER A ESTADO NORMAL
2  mask_invalidate   <=  "0000" when invalidar_all='1' else
3                          "1110" when dir_cjto="00" else
4              "1101" when dir_cjto="01" else
5              "1011" when dir_cjto="10" else
6              "0111" when dir_cjto="11" else
7              "0000";
8
9  -- Valid bits are set to '1' when a new block has been
       stored in MC (Tags_WE ='1')
10 validate_bit <= '1' when (Tags_WE ='1') else '0';
11 -- Valid bits are set to '0' when it is a fetch_inc
       operation, and it is a hit. In the second case we have to
        invalidate the block, since the data is going to change
       in memory
12
13 update_valid_bits <= validate_bit or (invalidate_bit and
       internal_hit) or invalidar_all; --Si dan la orden de
       invalidar,  s lo  se invalida donde haya acierto
14                                                  -- NUEVO
                                                       ETHICAL
                                                       HACKING:
                                                       UPDATE
                                                       TAMBI N
                                                       AL
                                                       INVLAIDAR
                                                        TODOS
```

```
15 --  we select the proper mask to validate or invalidate

16 valid_bits_in <= (valid_bits_out and mask_invalidate)   when
       invalidar_all='1' else -- SE INVALIDAN TODOS LOS BLOQUES
17                  (valid_bits_out OR mask_validate)    when
                        validate_bit ='1' else
18           (valid_bits_out AND mask_invalidate) when
                   invalidate_bit ='1' else
19           valid_bits_out;
```

Listing 21: Via_2025_WT.vhd

- **Modificación de lógica de *MC_Bus_data_out***: El sistema original no permitía escribir directamente en *MD*, ya que saca el dato de la *MC*. Como ahora la MC puede estar desactivada, es necesario cambiar la lógica:

```
1 MC_Bus_data_out <=  Din when (addr_non_cacheable = '1' or
      MC_desactivada = "1") else -- NUEVO!! AHORA NO SE PUEDE
      ESCRIBIR SIEMPRE EN CACHE
2           MC_Dout; -- is used to send the data to be written
```

Listing 22: MC_DATOS_2025_WT.vhd

Con estos elementos, se **modificó el autómata del controlador de la MC** para ajustarse al nuevo estado en el que la cache está desactivada. Ello conllevó fundamentalmente añadir nuevas transiciones para gestionar granularmente casos en los que la cache está desactivada, y algún comportamiento *Mealy* asociado al *Ghost Cache Monitoring*. A continuación, se adjunta el nuevo grafo de estados, así como la especificación de transiciones añadidas/modificadas:

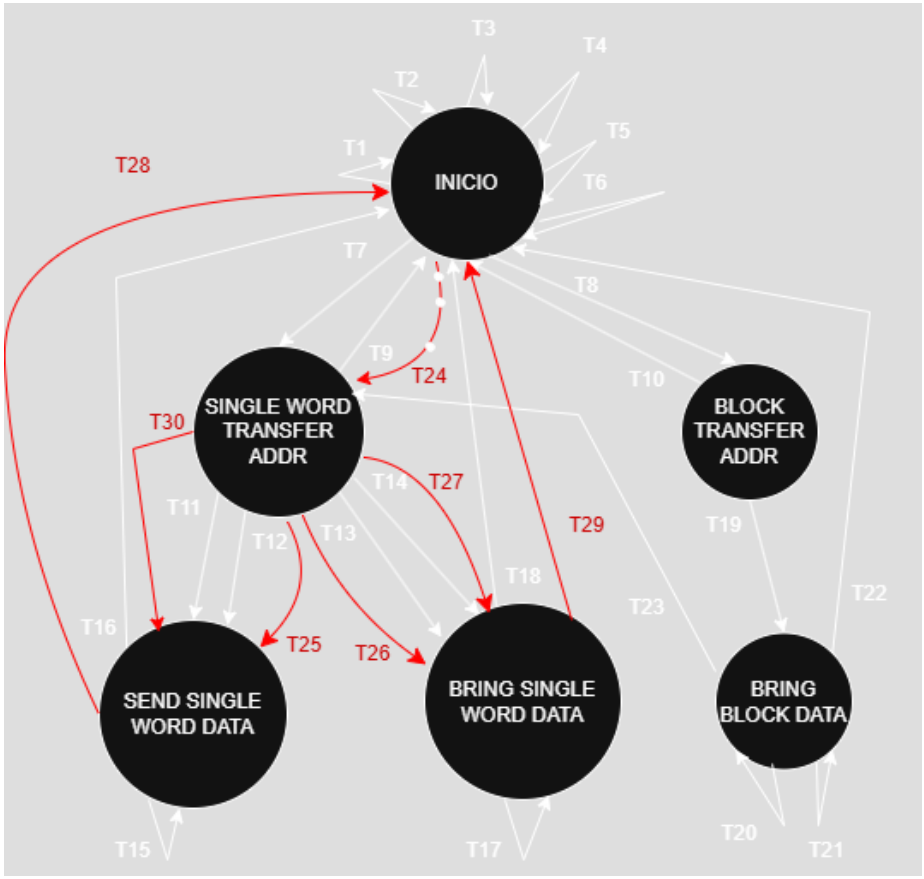

Figura 32: GRAFO DE ESTADOS PARA EL AUTÓMATA DE LA UC de la MC. ETHICAL HACKING.

ESTADO	SEÑALES ACTIVADAS
SINGLE WORD TRANSFER ADDR	Frame, MC_send_addr_ctrl; If (RE OR (Fetch_Inc and addr_non_cacheable) then MC_bus_Read else if(WB) then MC_bus_Write else MC_bus_Fetch_inc; If Bus_DevSel = '1' and addr_non_cacheable = '0' and MC_desactivada = '1' then inc_accMd

Cuadro 8: Tabla de señales de comportamiento de estado. ETHICAL HACKING

TRANSICIÓN	CONDICIONES	SEÑALES ACTIVADAS
T19	**Bus_Devsel = '1'**	**inc_m, inc_rm <= RE**
T24	**((RE = '1' or WE= '1') and hit = '0' and addr_non_cacheable = '0' and MC_desactivada = '1') or (RE = '1' and hit = '1' and MC_desactivada = '1')** and **bus_grant = '1'**	**bus_req**
T25	**Bus_DevSel = '1'** and **WE = '1'** and **hit = '1'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**inc_w**
T26	**Bus_DevSel = '1'** and **RE = '1'** and **hit = '1'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**inc_r**
T27	**Bus_DevSel = '1'** and **RE = '1'** and **hit = '0'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**inc_m, inc_rm, inc_r**
T28	**Bus_TRDY = '1'** and **WE = '1'** and **hit = '0'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**ready, MC_tags_WE**
T29	**Bus_TRDY = '1'** and **RE = '1'** and **hit = '0'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**ready, mux_output = "01", MC_tags_WE**
T30	**Bus_DevSel = '1'** and **WE = '1'** and **hit = '0'** and **addr_non_cacheable = '0'** and **MC_desactivada = '1'**	**inc_w, inc_m**

13.4.2. Resultados

Una vez llevadas al cabo las implementaciones, se volvió al origen del problema: el programa *ethicalHacking.asm*. Ahora bien, la nueva ejecución del programa tuvo un resultado distinto, como se puede ver a continuación:

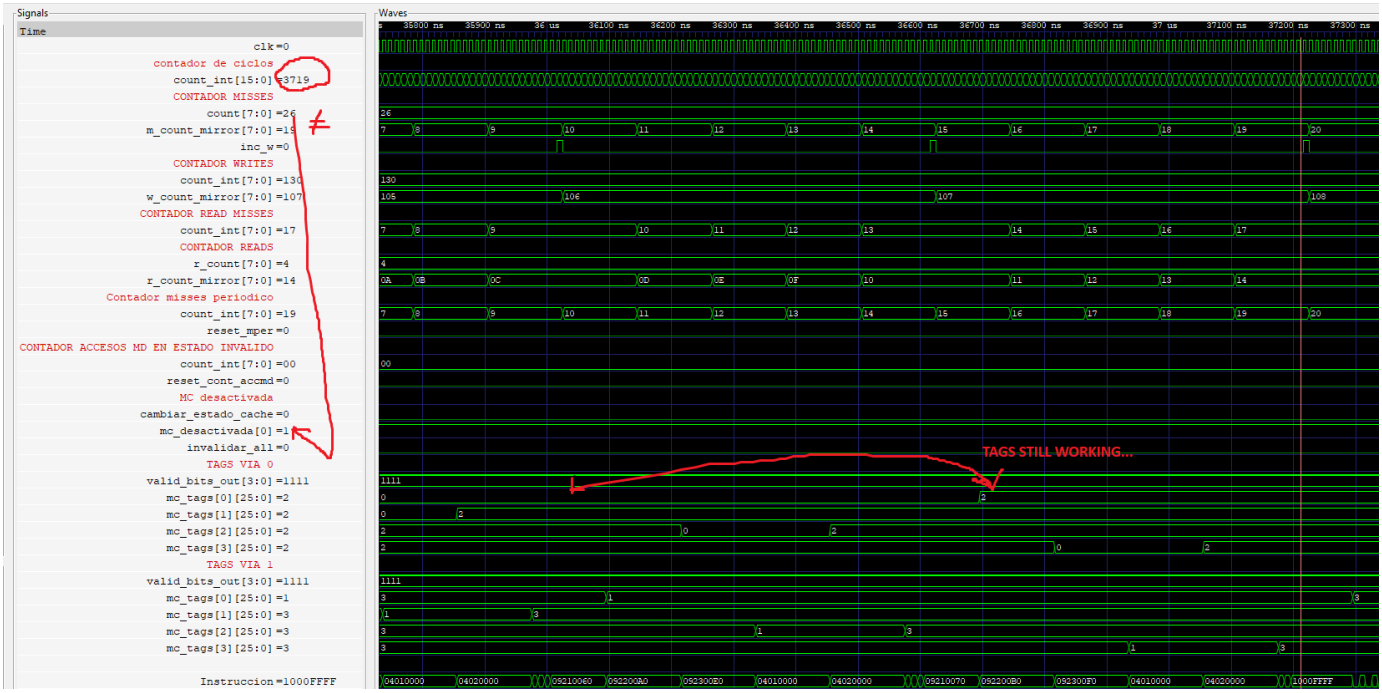

Figura 33: ETHICAL HACKING. MEJORA

(NOTA: El comportamiento no es del todo correcto el desborde de contadores)

Como se puede ver, los tags de la cache siguen alternándose como antes (pues el sistema de tags sigue activo), pero en los inicios del programa mismo el sistema de la *MC* se ha desactivado (mc_desactivada = 1). Nótese como el valor de los contadores originales y .^{es}pejo.^{es} diferente. En cuanto a los resultados, son muy favorecedores; y es que el programa tarda 3719 ciclos en llegar a la última instrucción, lo que resulta en un

$$Speedup = \frac{5561}{3719} = 1{,}49$$

Por último, también se adjunta una pequeña simulación para ver como funciona la reactivación de la MC cuando supera de nuevo el umbral de eficiencia. Para ello, se procede a explotar de *misses*, para posteriormente leer de la misma dirección (*hits fantasma*) hasta que se vuelva a estado 'normal':

```
sw r0, 0(r9)
sw r0, 64(r9)
sw r0, 128(r9)
sw r0, 192(r9)
sw r0, 0(r9)
sw r0, 64(r9)
sw r0, 128(r9)
sw r0, 192(r9)
sw r0, 0(r9)
sw r0, 64(r9)
sw r0, 128(r9)
... (HASTA DESACTIVAR LA CACHE Y M S ALL )....
lw r0, 0(r9)
lw r0, 0(r9)
lw r0, 0(r9)
lw r0, 0(r9)
lw r0, 0(r9)
lw r0, 0(r9)
.... (HASTA REACTIVAR LA CACHE) ....
```

Listing 23: ethicalHacking.asm

A continuación, se adjuntan dos trazas del programa:

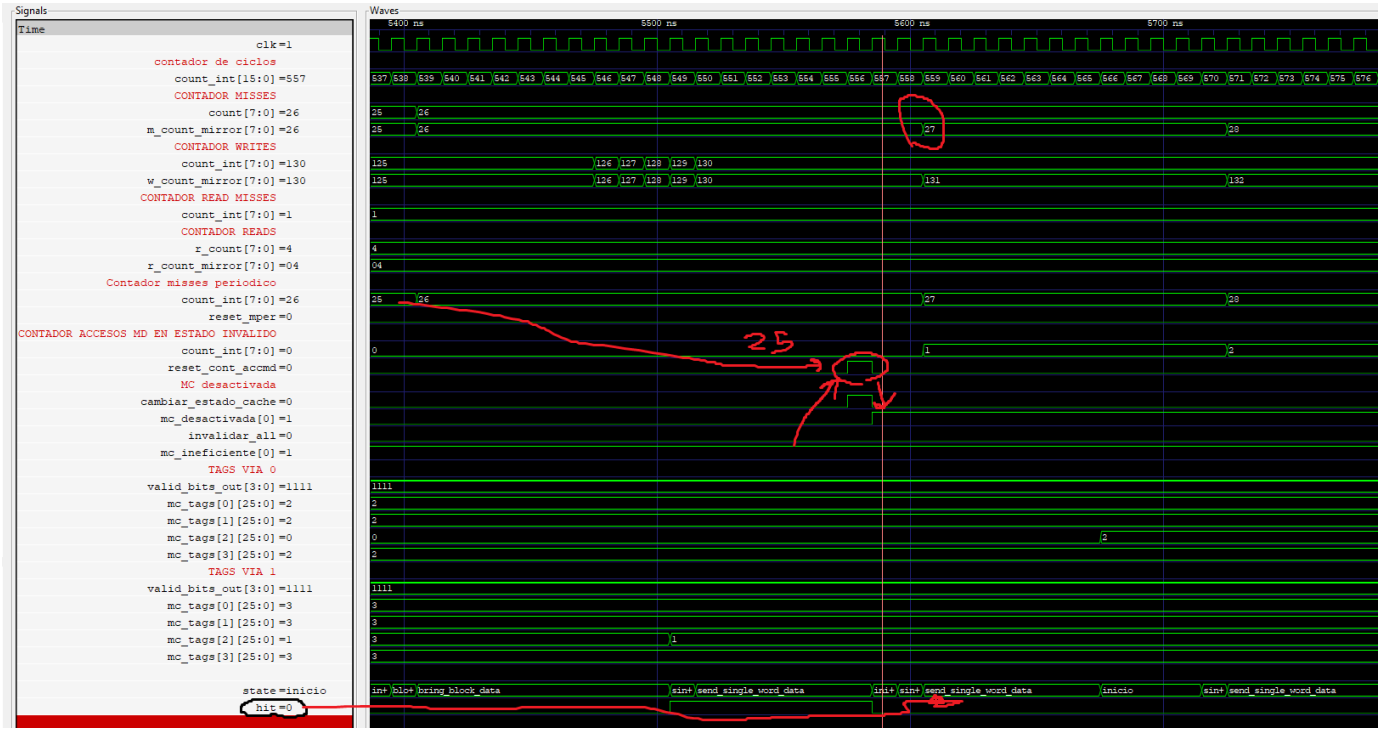

Figura 34: Desactivación de la MC

.

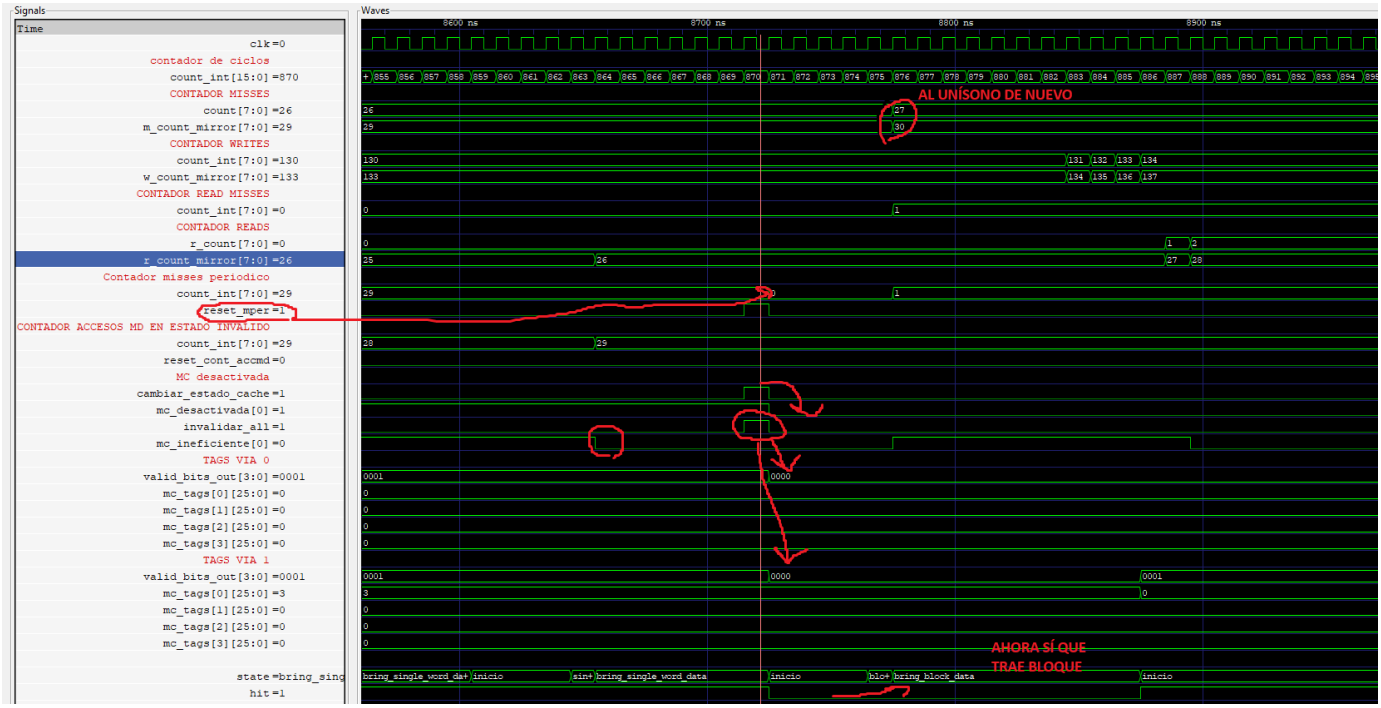

Figura 35: Reactivación de la MC

.

En la primera imagen, se puede observar como a pesar de que la MC se encontraba por debajo de su umbral de eficiencia, el estado no cambia hasta que que el número de *misses* supera el valor de 25. Así, aunque la siguiente operación ocasionase un *write-miss* en condiciones normales, ahora llevará a una sola transferencia (envío) de palabra con la MD.

En la segunda imagen, se puede observar como cuando la cache vuelve a *ser* eficiente, se invalidan los tags de las dos vías, y la siguiente operación es un *read-miss* obligatorio. Nótese, además, que el contador de *misses periódico* se ha reseteado para dejar un lapso hasta la siguiente desactivación.