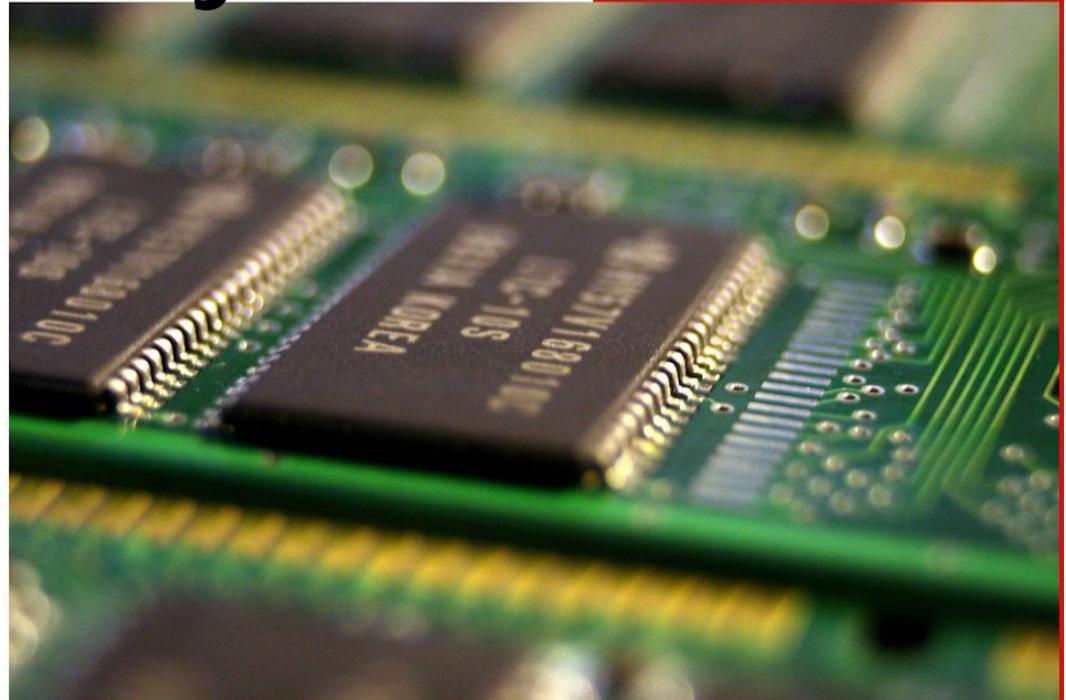


Proyecto 2 AOC II



**Nestor Mozón Gonzalez 735418
José Manuel Sánchez Aquilué 759267**

Introducción.	3
Diagrama de estados.	3
Explicación de nuestro diseño y hardware añadido.	5
Descripción algorítmica.	9
Impacto en rendimiento.	9
Pruebas realizadas - 1.	10
Mejoras optativas	15
Mejora optativa 2 - Adelantar envío	15
Mejora optativa 1 - Buffer para escrituras	19
Primera prueba (código subsanación del proyecto 1)	22
Pruebas realizadas - 2.	25
Prueba definitiva (RAM_I_tb2.vhd):	25
Resumen de nuestras aportaciones.	34
José Manuel.	34
Néstor:	35
Autoevaluación.	36
José Manuel.	36
Néstor.	36
Anexo: Diagrama de estados UC completa	38

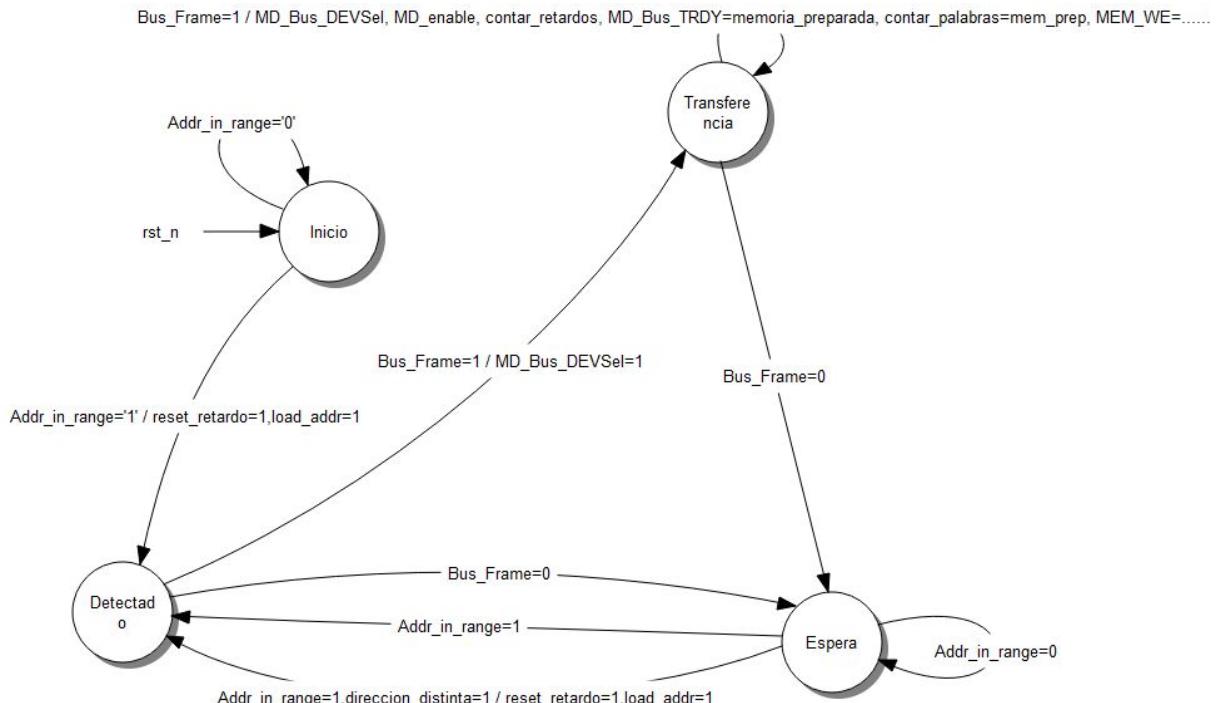
Introducción.

En este documento hemos dejado plasmado el proceso de elaboración del proyecto 2. Comenzamos desarrollando la unidad de control que pedía el ejercicio y haciendo los cambios de hardware necesarios. Una vez comprobado el funcionamiento de lo anterior, decidimos añadir unas mejoras con el fin de aumentar el rendimiento del procesador. Por un lado adelantamos el envío de palabras en operaciones de carga y por otro añadimos buffers de escritura.

Diagrama de estados.

Antes de proceder a teclear el código vhdl, leímos el enunciado con suma atención. Una vez que comprendimos lo pedido, examinaremos los códigos fuente nuevos y comenzamos a familiarizarnos con la memoria caché. El esquema de la memoria fue fundamental a la hora de realizar el ejercicio, ya que nos facilitó la comprensión de las señales de la memoria y los buses.

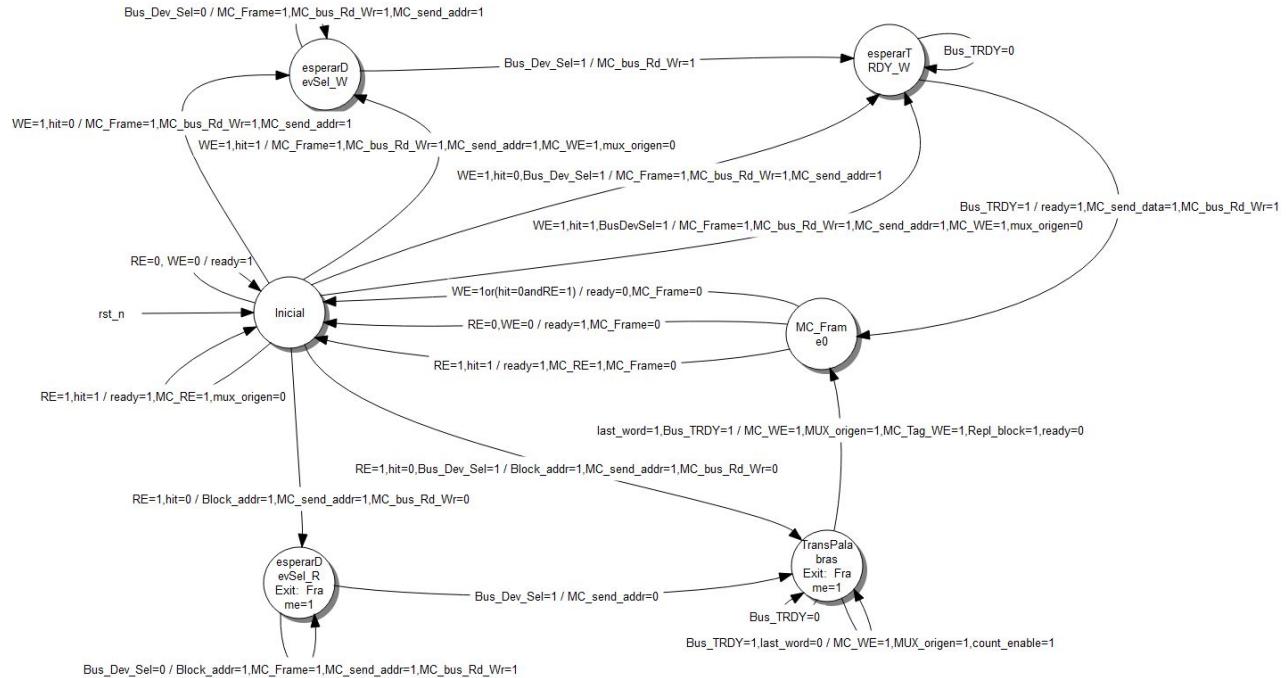
Con el propósito de entender mejor el controlador de la memoria de datos, dibujamos el autómata en qfsm (obviando parte de las señales de la transición de la transferencia):



Autómata del controlador de la MP

El siguiente paso fue la elaboración del diagrama de estados. Hizo falta una serie de esbozos en papel hasta dar con el autómata final. De hecho, nuestra versión definitiva de qfsm no la obtuvimos hasta realizar las pruebas en modelsim y reparar una serie de errores.

Nuestro diseño consta de seis estados, debido a que separamos los estados de lectura y de escritura. Aunque se podrían reducir, esta distinción no afecta al rendimiento. Sin embargo, consideramos que aporta claridad y legibilidad al diagrama de estados y al código. No hemos incluido las señales de los contadores, pero se pueden ver en el código más adelante.



Autómata de la UC de la MC

Como se puede observar, inicialmente tenemos cinco casos posibles: que no haya solicitudes (el caso trivial), y los cuatro correspondientes a aciertos y fallos de escritura (arriba) y lectura (abajo).

En escritura, si ha habido acierto, seguimos la política *Write Through*: escribimos primero en la MC y luego en la MP. En caso de fallo, siguiendo la política *Write Around*, solo necesitamos escribir en MP. Así, seguimos un camino idéntico al de acierto, omitiendo al principio la escritura en MC.

Al hacer la escritura en memoria caché en la primera transición en acierto de escritura (y no ser necesaria en fallo, evidentemente), en el último ciclo (la transición entre esperarTRDY_W y Frame0, que se encarga de enviar el dato a memoria principal) podemos poner el ready a 1, ya que la cache ya puede atender otra petición de CPU a partir del mismo.

Por otra parte, en lectura, el acierto es inmediato, nos devuelve directamente el dato y vuelve al estado inicial. Con el fallo, en cambio, debemos primero traer el bloque completo a MC, escribiendo en cuatro ciclos las palabras.

Sea lectura o fallo de escritura, al utilizar el bus, siempre tenemos que pasar por el estado MC_Frame0 para poner el Frame a 0 antes de volver al estado inicial. En caso contrario, se darían errores en la comunicación con la MP, que no tendría forma de distinguir una nueva transmisión de la antigua.

Finalmente, las transiciones que unen el estado inicial con esperarTRDY (en lectura y escritura), se han añadido para tener en cuenta el caso en el que la máquina Mealy del controlador de la MD devuelva el BusDevSel a uno en el mismo ciclo de la petición.

Explicación de nuestro diseño y hardware añadido.

En primer lugar, tomamos la decisión de implementar el controlador de la memoria cache. Nos limitamos a pasar nuestro diseño del autómata, sin las optimizaciones optativas, a instrucciones vhdl.

```
-- poner en el siguiente type el nombre de vuestros estados
type state_type is (Inicio,esperarDEVSel_R, esperarDEVSel_W,transPalabras,frame0,esperarTRDY_W);
----- -----
-- Poned aquí el código de vuestra máquina de estados
OUTPUT_DECODE: process (state, hit, last_word, bus_TRDY, RE, WE, Bus_DevSel, palabra_buscada)
begin
    -- Se comienza dando los valores por defecto, si no se asigna otro valor en un estado val.
    -- Así no hay que asignar valor a todas las señales en cada caso
    MC_WE <= '0';
    MC_bus_Rd_Wr <= '0';
    MC_tags_WE <= '0';
    MC_RE <= RE;
    ready <= '0';
    mux_origen <= '0';
    MC_send_addr <= '0';
    MC_send_data <= '0';
    next_state <= state;
    count_enable <= '0';
    Frame <= '0';
    Replace_block <= '0';
    block_addr <= '0';
    inc_rm <= '0';
    inc_wm <= '0';
    inc_wh <= '0';

    -- Estado INICIO:
    if (state = Inicio and RE= '0' and WE= '0') then -- si no piden nada no hacemos nada
        next_state <= Inicio;
        ready <= '1';
    end if;
```

```

    elsif (state = Inicio and RE='1' and hit='1') then
        next_state <= Inicio;
        ready<='1';
        MC_RE<='1';
        mux_origen<='0'; |
    elsif (state = Inicio and RE='1' and hit='0') then
        if (Bus_DevSel='0') then
            next_state <= esperarDEVSel_R;
        else
            next_state <= transPalabras;
        end if;
        Block_addr<='1';
        Frame<='1';
        MC_send_addr<='1';
        MC_bus_Rd_Wr<='0';
        inc_rm <='1';
    elsif (state = Inicio and WE='1' and hit='1') then
        if (Bus_DevSel='0') then
            next_state <= esperarDEVSel_W;
        else
            next_state <= esperarTRDY_W;
        end if;
        Frame<='1';
        MC_bus_Rd_Wr<='1';
        MC_send_addr<='1';
        MC_WE<='1';
        mux_origen<='0';
        inc_wh <='1';
    . . .
    elsif (state = Inicio and WE='1' and hit='0') then
        if (Bus_DevSel='0') then
            next_state <= esperarDEVSel_W;
        else
            next_state <= esperarTRDY_W;
        end if;
        Frame<='1';
        MC_bus_Rd_Wr<='1';
        MC_send_addr<='1';
        inc_wm<= '1';
-- LECTURA:
    elsif (state = esperarDEVSel_R and Bus_DevSel='0') then
        next_state <= esperarDEVSel_R;
        Block_addr<='1';
        Frame<='1';
        MC_send_addr<='1';
        MC_bus_Rd_Wr<='0';
    elsif (state = esperarDEVSel_R and Bus_DevSel='1') then
        next_state <= transPalabras;
        Frame<='1';
        MC_bus_Rd_Wr<='0';
    elsif (state = transPalabras and Bus_TRDY='0') then
        next_state <= transPalabras;
        Frame<='1';

```

```

    elsif (state = transPalabras and Bus_TRDY='1' and last_word='0') then
        next_state <= transPalabras;
        Frame<='1';
        MC_bus_Rd_Wr<='0';
        MC_WE<='1';
        mux_origen<='1';
        count_enable<='1';
    elsif (state = transPalabras and Bus_TRDY='1' and last_word='1') then
        next_state <= frame0;
        Frame<='1';
        MC_bus_Rd_Wr<='0';
        MC_WE<='1';
        mux_origen<='1';
        count_enable<='1';
        MC_tags_WE<='1';
        Replace_block<='1';
-- FRAME A 0:
    elsif (state = frame0 and RE='1' and hit='1') then
        next_state <= Inicio;
        ready<='1';
        MC_RE<='1';
        mux_origen<='0';
    elsif (state = frame0 and RE= '0' and WE= '0') then -- si no piden nada no hacemos nada
        next_state <= Inicio;
        ready <= '1';
    elsif (state = frame0 and (WE='1' or hit='0')) then -- si no piden nada no hacemos nada
        next_state <= Inicio;
        ready <= '0';

-- ESCRITURA:
    elsif (state = esperarDEVSel_W and Bus_DevSel='0') then
        next_state <= esperarDEVSel_W;
        Frame<='1';
        MC_bus_Rd_Wr<='1';
        MC_send_addr<='1';
    elsif (state = esperarDEVSel_W and Bus_DevSel='1') then
        next_state <= esperarTRDY_W;
        Frame<='1';
        MC_bus_Rd_Wr<='1';
    elsif (state = esperarTRDY_W and Bus_TRDY='0') then
        next_state <= esperarTRDY_W;
        Frame<='1';
        MC_bus_Rd_Wr<='1';
    elsif (state = esperarTRDY_W and Bus_TRDY='1') then
        next_state <= frame0;
        Frame<='1';
        ready<='1';
        MC_send_data <= '1';
        MC_bus_Rd_Wr<='1';
-- Poner aqui las condiciones de vuestra máquina de estado
-- elsif() then
-- else

    end if;

```

Después de llevar a cabo las pruebas de memoria caché, añadimos los contadores al mips. Hizo falta tener en cuenta que si el procesador está parado porque la memoria no está preparada, no hay que incrementar el resto de contadores.

```

inc_paradas_control <= '1' when(Mem_ready='1' and predictor_error='1') else '0';
inc_paradas_datos <= '1' when(Mem_ready='1' and (riesgo_lw_uso = '1' or riesgo_beq = '1')) else '0';
inc_paradas_memoria <= '1' when(Mem_ready = '0') else '0';
inc_mem_reads <= '1' when(Mem_ready='1' and IR_ID(31 downto 26)= "000010") else '0';
inc_mem_writes <= '1' when(Mem_ready='1' and IR_ID(31 downto 26)= "000011") else '0';

```

Por último, para efectuar la detención de memoria es necesario cambiar la señal de load de los bancos de registro. En los dos primeros, esa señal depende de la señal avanzar_id, por tanto ha sido necesario cambiarla. En el resto, hemos cambiado load para que cargue únicamente cuando mem_ready es igual a 1.

```

-- en funci n de los riesgos se para o se permite continuar a la instrucci n en ID
avanzar_ID <= '0' when (riesgo_lw_uso = '1' or riesgo_beq = '1' or Mem_ready='0') else '1';

[Banco_ID_EX: Banco_EX PORT MAP ( clk => clk, reset => reset, load => Mem_ready, busA => busA, busB => busB, busA_EX => busA_EX, busB_EX => busB_EX,
RegDst_ID => RegDst_ID, ALUSrc_ID => ALUSrc_ID, MemWrite_ID => MemWrite_ID, MemRead_ID => MemRead_ID,
MemtoReg_ID => MemtoReg_ID, RegWrite_ID => RegWrite_ID, RegDst_EX => RegDst_EX, ALUSrc_EX => ALUSrc_EX,
MemWrite_EX => MemWrite_EX, MemRead_EX => MemRead_EX, MemtoReg_EX => MemtoReg_EX, RegWrite_EX => RegWrite_EX,
ALUctrl1_ID => ALUctrl1_ID, ALUctrl_EX => ALUctrl_EX, inm_ext => inm_ext, inm_ext_EX=> inm_ext_EX,
inm_ext_EX=> inm_ext_EX);

[Banco_EX_MEM: Banco_MEM PORT MAP ( ALU_out_EX => ALU_out_EX, ALU_out_MEM => ALU_out_MEM, clk => clk, reset => reset, load => Mem_ready, MemWrite_EX => MemWrite_EX,
MemRead_EX => MemRead_EX, MemtoReg_EX => MemtoReg_EX, RegWrite_EX => RegWrite_EX, MemWrite_MEM => MemWrite_MEM, MemRead_MEM => MemRead_MEM,
MemtoReg_MEM => MemtoReg_MEM, RegWrite_MEM => RegWrite_MEM, BusB_EX => Mux_B_out, BusB_MEM => BusB_MEM, RW_EX => RW_EX, RW_MEM => RW_MEM);

[Banco_MEM_WB: Banco_WB PORT MAP ( ALU_out_MEM => ALU_out_MEM, ALU_out_WB => ALU_out_WB, Mem_out => Mem_out, MDR => MDR, clk => clk, reset => reset, load => Mem_ready,
MemtoReg_MEM => MemtoReg_MEM, RegWrite_MEM => RegWrite_MEM,
MemtoReg_WB => MemtoReg_WB, RegWrite_WB => RegWrite_WB, RW_MEM => RW_MEM, RW_WB => RW_WB );

```

Descripción algorítmica.

```
look up(@)                                1
if ( fallo )
    switch( tipo )
        case read: Mp(rB,@x); wait Mp; Mc+X; ret;      CrB+1
        case write: Mp(wW, X'); ret;                      CwW
else
    switch (tipo )
        case read: ret x';                            0
        case write: Mc+x'; Mp(wW, X'); ret;            CwW
```

Impacto en rendimiento.

$$C_{eff} = 1 + \frac{\sum wh \times CwW}{\sum refs} + \frac{\sum wm \times CwW}{\sum refs} + \frac{\sum rm \times (CrB + 1)}{\sum refs}$$

CwW= 8 ciclos

CrB= 11 ciclos

Hay que tener en cuenta que, debido a las optimizaciones opcionales, el número de ciclos varía en función de las instrucciones que vengan después y la posición en el bloque palabra que solicites. En el mejor de los casos un fallo de lectura puede suponer sólo una parada de 6 ciclos. Además mientras se escribe en memoria de datos se podría seguir utilizando el procesador sin necesidad de parar, siempre y cuando no llegara una instrucción que necesitase la memoria.

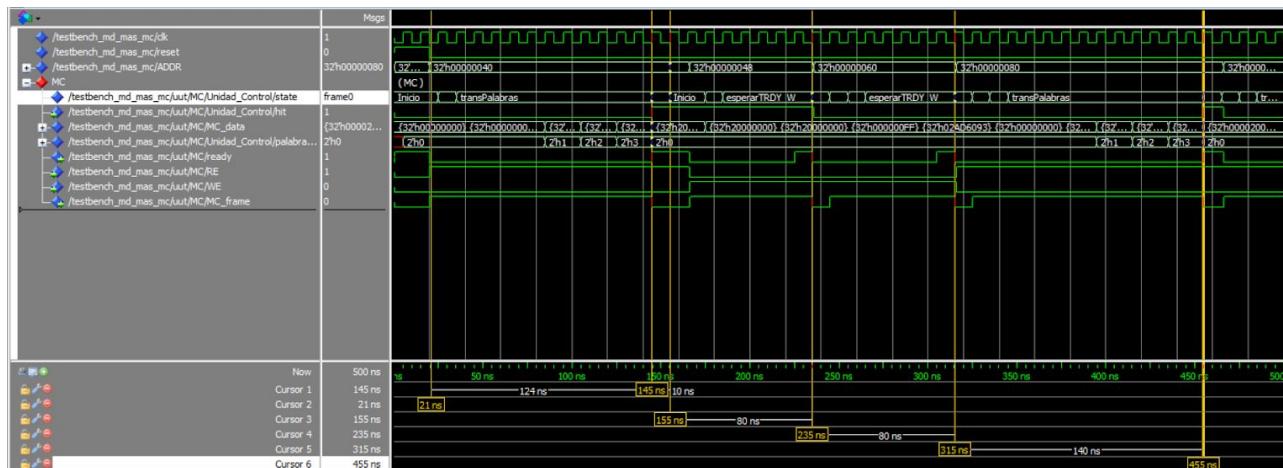
Pruebas realizadas - 1.

Como finalmente nos decidimos a realizar las dos partes optativas, dividimos las pruebas principalmente en dos partes. La primera, con los bancos de pruebas proporcionados por los profesores, comprueba el funcionamiento correcto del autómata de la parte obligatoria. La segunda, una vez implementadas ambas mejoras, es mucho más exhaustiva, y la expondremos después de la descripción de las mejoras.

Así, mostramos a continuación las primeras pruebas.

Banco de prueba de los profesores (*testBench_MD_mas_MC_alumnos.vhd*):

Esta prueba nos sirvió principalmente para encontrar errores o bien en nuestro autómata, o bien en nuestra traducción del mismo a vhdl. Tras solucionarlos (varios bits al revés, algunas señales que faltaban), vimos que seguía correctamente el comportamiento esperado:



Ondas de la primera prueba

Cada cursor (líneas amarillas) marca el procesamiento de una solicitud en la memoria por la UC (read o write). Entre la primera (fin del reset) y la segunda, Read Miss; entre la segunda y la tercera Read Hit; a continuación, dos Write Hit y, finalmente, dos Read Miss.

Observamos también que el MC_Frame baja como se esperaba tras cada utilización del bus.

En todo momento nos aseguramos de que el contenido en memoria, tanto caché como ram

Banco de prueba de los profesores (RAM_I_tb1.vhd):

En primer lugar, realizamos una prueba del mips con las instrucciones del proyecto anterior. Con el objeto de comprobar que, a pesar de nuestras modificaciones en memoria y las paradas adicionales, el resultado era el mismo (mutatis mutandi) que en el primer proyecto. Esta prueba tuvo lugar antes de las modificaciones opcionales.

Estas son las instrucciones que cargamos en memoria.

LW R0, 0(R0)

nop

LA R2, 4(R1); R2=4, R1=0

LW R3, 0(R2); R3= MD(x04)=4 #corto a distancia 1 con el LA

ADD R4, R3, R3; R4=8 # parada + cortos a distancia 2

SW R4, 4(R4); MD(x14)=8 #cortos a distancia 1

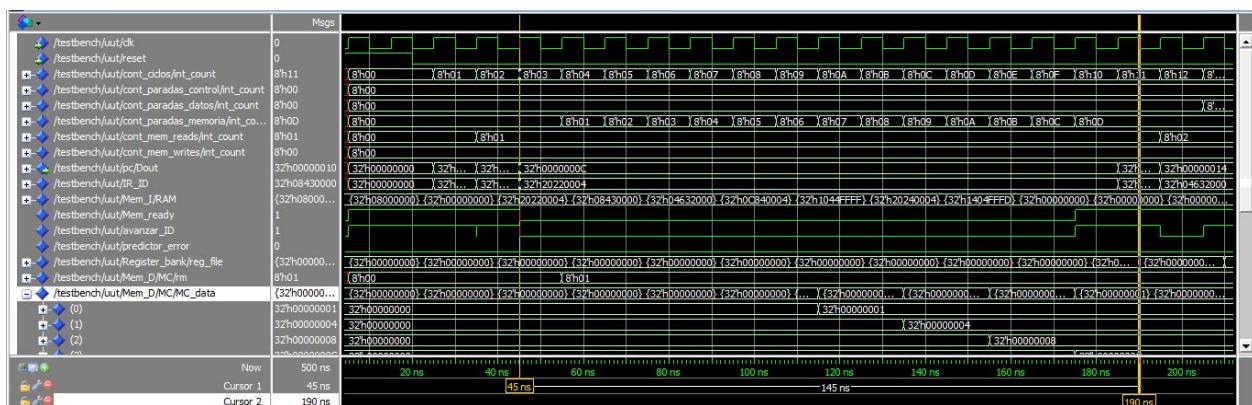
loop BEQ R2, R4, loop; No se salta la primera vez: R2=4, R4=8.

; Sí se salta la segunda (predictor falla): R2=4, R3=4 , y en las siguientes el predictor acierta

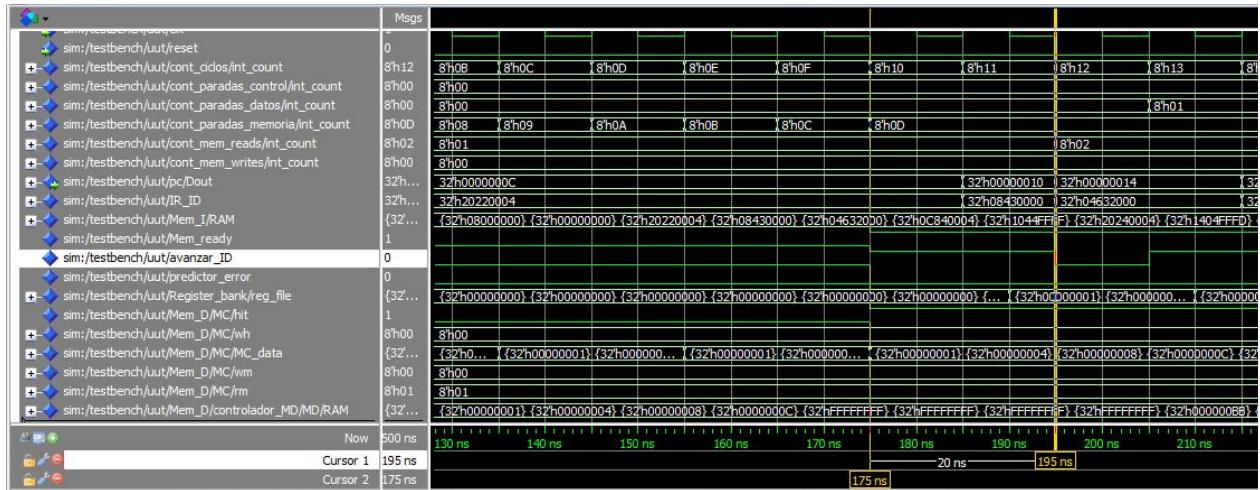
LA R4, 4(R1); R4=4, R1=0

BNE R0, R4, loop; Sí se salta: R4=4, R0=0; Se vuelve al BEQ anterior

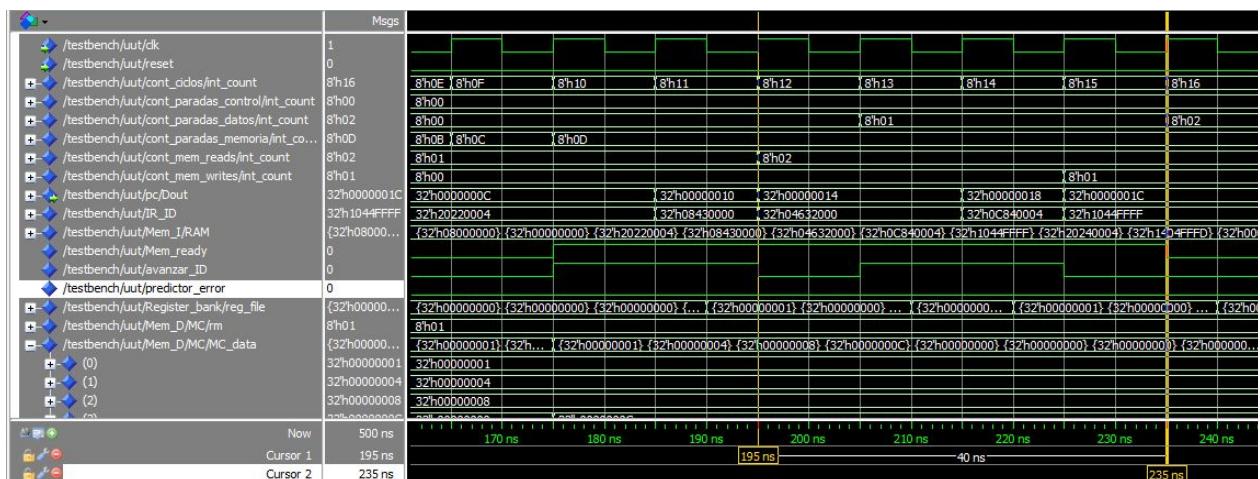
La primera detención se produce cuando el primer load accede a memoria y carga el primer bloque en caché. Como es un read miss, la detención dura 12 ciclos.



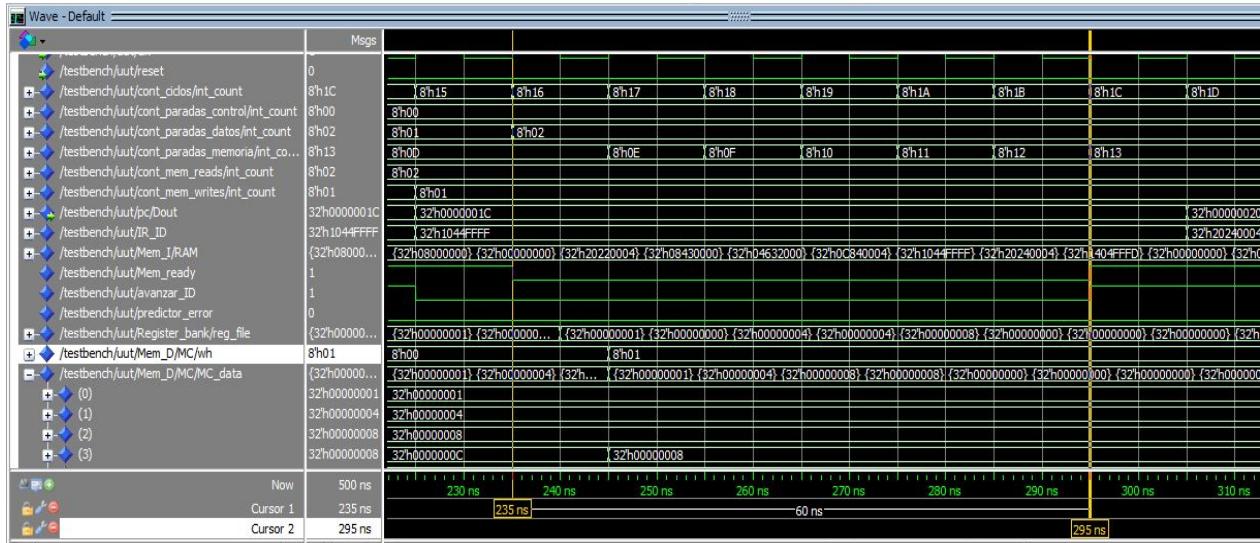
Por otra parte, cuando se procesa la segunda instrucción de load, no se produce ninguna parada porque es un read hit



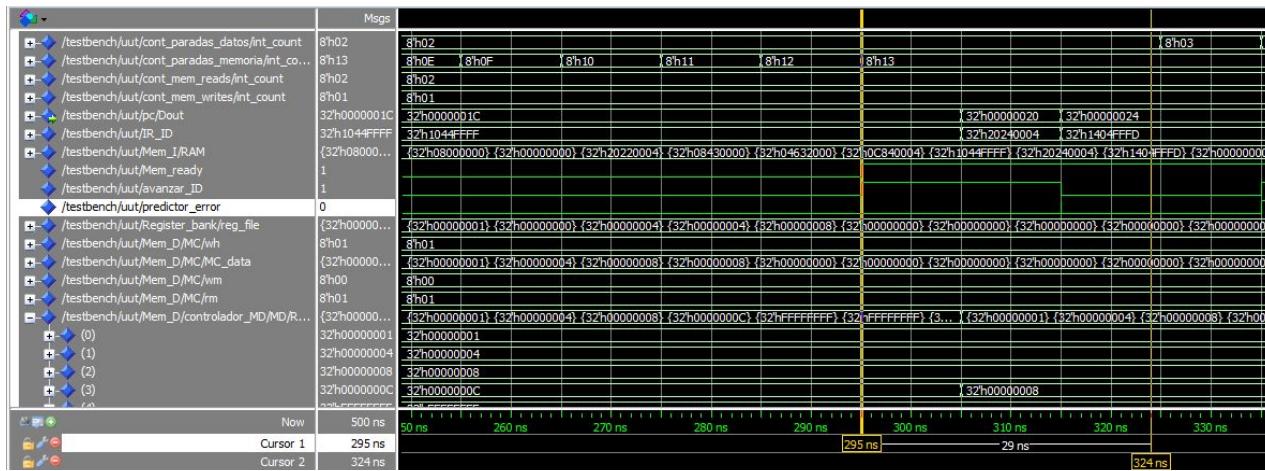
A continuación vienen la parada de datos que se daban en la prueba del proyecto uno. Seguidas de una parada de memoria por la instrucción de store.



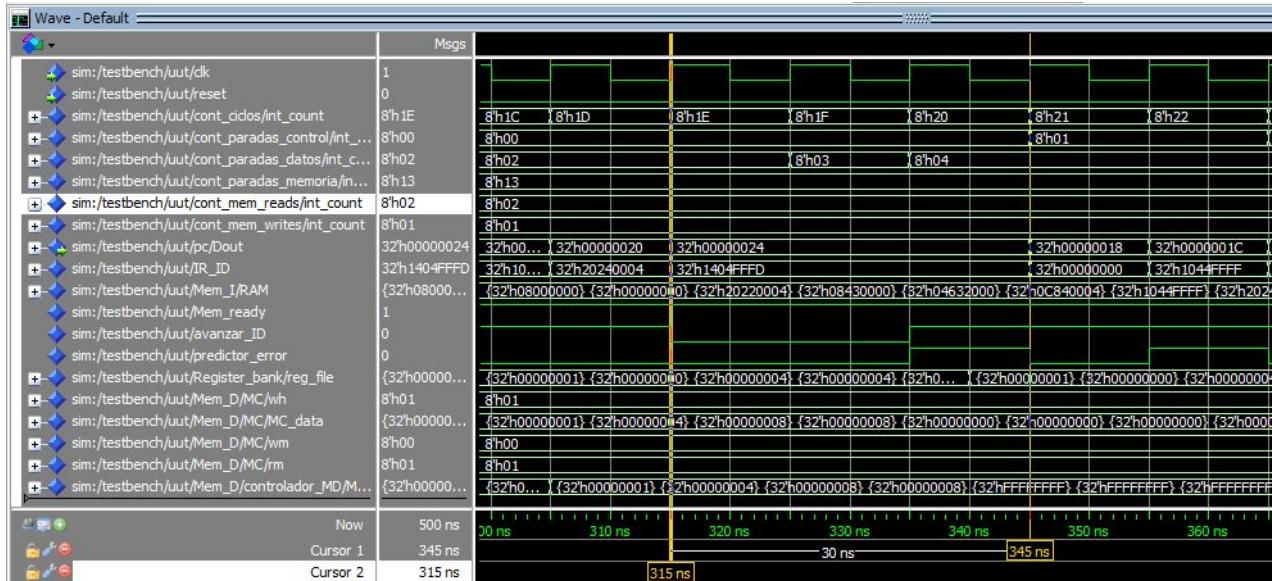
En este caso como es un write hit, ya que estamos escribiendo en el mismo bloque que hemos cargado anteriormente, la parada dura 6 ciclos



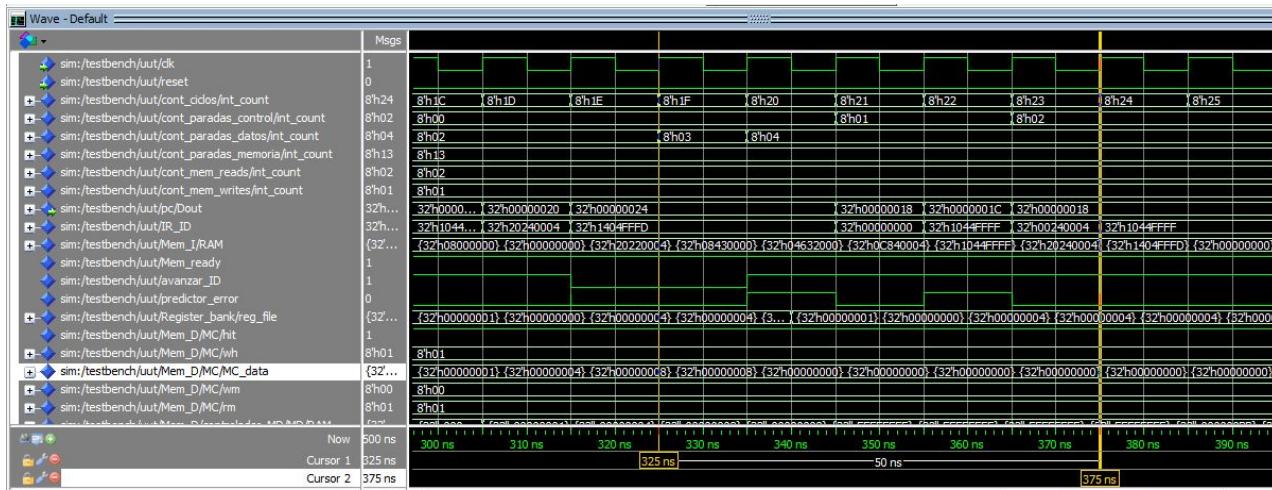
.Como era de esperar, también se actualiza la RAM.



Después, comprobamos que se produce la parada en beq como en el proyecto anterior.



Finalmente, observamos que el predictor de saltos se comporta de la manera prevista. Como podemos apreciar, este código se ejecutó en 35 ciclos, a continuación veremos cuánto tarda con las mejoras de rendimiento implementadas.



Mejoras optativas

Como hemos comentado, a continuación describiremos en detalle las mejoras, antes de exponer las pruebas que las engloban.

Inicialmente, decidimos comenzar por la optativa 2, sin tener claro si terminaríamos implementando también la primera. Finalmente, nos decidimos a realizar ambas, con lo que las presentaremos en este orden.

Mejora optativa 2 - Adelantar envío

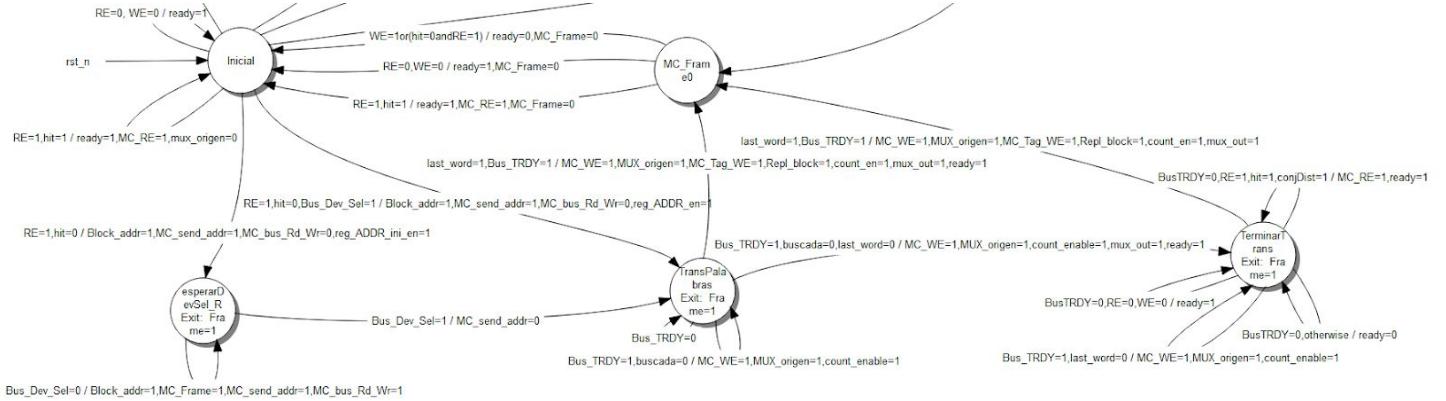
Para disminuir las paradas en memoria en caso de fallo de lectura, una de las opciones es enviar la palabra buscada al procesador en el mismo ciclo en que se recibe de la memoria principal, en lugar de esperar a reemplazar el bloque entero como hacía la primera versión de la caché. Para esto, en cuanto al autómata de la UC solo ha sido necesario añadir un estado adicional, *terminarTrans*, al que llegamos una vez se ha recibido la palabra buscada, y nos mantenemos hasta sustituir el resto del bloque. En este proceso, no obstante, debemos colocar el ready a 1 siempre que sea posible, atendiendo las nuevas peticiones del procesador.

Aunque podríamos haber optado por bloquear la CPU siempre que llegara cualquier petición de memoria, decidimos gestionar también aciertos de lectura en los ciclos en los que la memoria principal no está enviando ningún dato (BusTRDY=0).

Para el correcto funcionamiento del nuevo autómata hemos añadido estas cuatro nuevas señales, que detallaremos más adelante:

1. **reg_ADDR_ini_en**: se trata del enable de un nuevo registro añadido a la MC (reg_ADDR_ini, en el fichero MC_datos del proyecto) que contendrá la dirección pedida por la CPU al principio de un read miss (es decir, antes de devolver el ready a 1, es la dirección que está transmitiendo la MC desde la MD).
2. **buscada**: es la salida de un comparador entre palabra_UC (la salida del contador de la UC usada para la transferencia de las 4 palabras) y los bits de palabra de la dirección entrante del procesador (ADDR(3:2)). Así, es 1 si y solo si la palabra palabra_UC es la que busca la CPU.
3. **mux_out**: señal de controlador del nuevo multiplexor cuya salida es Dout, el dato enviado a CPU, y cuyas entradas son MC_Dout (como era en la versión original), para los aciertos de lectura, y Bus_Din (la palabra que llega desde MP), para transmitir directamente dicha palabra a CPU en caso de fallo.
4. **cjtoDist**: indica que el conjunto en el registro reg_ADDR_ini es distinto que el que llega por la entrada ADDR. Sirve para no dar falsos hits al aceptar lecturas en el estado *terminarTrans*.

A continuación, presentamos el nuevo esquema del autómata (solo mostramos la parte correspondiente a lectura, ya que la de escritura se mantiene idéntica, aunque en el anexo mostramos el autómata definitivo tras las dos mejoras):



Autómata de la UC de la MC con mejora optativa 2

Los primeros cambios serían la adición de `reg_ADDR_ini_en` en las dos transiciones salientes de *Inicio* en lectura, ya que necesitamos guardar la dirección (específicamente los bits de etiqueta y conjunto) antes de pasar a *TerminarTrans* para sustituir en el conjunto apropiado de la RAM de Tags la etiqueta inicial y, al mismo tiempo, poder gestionar nuevos aciertos de lectura en *terminarTrans*.

Así, en el autómata, el estado *TransPalabras* se mantiene bastante similar al original, solo que ahora presenta una nueva condición de salida: que buscada sea 1 (y no sea la última palabra, `last_word=0`), cuando la palabra que llega de MP es precisamente la pedida. En ese caso, además de escribirla en la caché, la enviamos directamente a CPU por Dout (gracias a `mux_out=1`). En ese mismo instante, desbloqueamos el procesador (`ready=1`), y pasamos al nuevo estado *TerminarTrans*.

Este se encarga de terminar la transmisión de las palabras restantes, mientras atiende a nuevas peticiones del procesador. Se mantiene en este estado mientras queden palabras por transmitir (`last_word` todavía no ha sido 1). Cuando TRDY es 1, es decir, MP está transmitiendo un dato, solo mantenemos desbloqueado el procesador si no llega ninguna petición a memoria (`RE` y `WE` son 0). En caso contrario, como MC está ocupada escribiendo la palabra que llega de MP, se debe bloquear hasta el ciclo siguiente.

Por su parte, cuando TRDY es 0, la memoria caché está preparada para gestionar aciertos de lectura. No obstante, como hemos comentado, debemos comprobar antes que el conjunto apuntado por la dirección entrante no sea el mismo que estamos reemplazando, ya que, por ejemplo, si se ha sustituido la primera palabra, el tag en la RAM aún no se ha actualizado (se hace con `last_word`) y se daría un hit, pudiendo enviar la palabra recién traída, que no se corresponde con la dirección pedida. En caso contrario, en el resto de conjuntos, se pueden dar aciertos de lectura con normalidad, manteniendo `mem_ready` a 1.

Ocurre lo mismo si no llegan peticiones (RE y WE son 0), mem_ready sigue siendo 1. En cualquier otro caso, si llega un fallo de lectura o cualquier petición de escritura, detenemos el procesador, ya que la MC debe antes terminar de sustituir el bloque.

Finalmente, cuando llega last_word=1, se da la transición a frame0, manteniendo ready a 1 solamente si no llegan peticiones (ya que estamos usando el bus, se trata básicamente de un caso especial de la transición ya descrita con TRDY a 1).

A continuación, presentamos el código del autómata correspondiente a esta parte:

```
-- Estado INICIO:
if (state = Inicio and RE= '0' and WE= '0') then -- si no piden nada no hacemos nada
    next_state <= Inicio;
    ready <= '1';
elsif (state = Inicio and RE='1' and hit='1') then
    next_state <= Inicio;
    ready<='1';
    MC_RE<='1';
    mux_origen<='0'; --estaba a 1 y creo que tiene que ser 0
elsif (state = Inicio and RE='1' and hit='0') then
    if (Bus_DevSel='0') then
        | next_state <= esperarDEVSel_R;
    else
        | next_state <= transPalabras;
    end if;
    Block_addr<='1';
    Frame<='1';
    MC_send_addr<='1';
    MC_bus_Rd_Wr<='0';
    inc_rm <='1';
    reg_ADDR_ini_en<='1'; -- guardamos la dirección (opcional 2)
    . . .

-- LECTURA:
elsif (state = esperarDEVSel_R and Bus_DevSel='0') then
    next_state <= esperarDEVSel_R;
    Block_addr<='1';
    Frame<='1';
    MC_send_addr<='1';
elsif (state = esperarDEVSel_R and Bus_DevSel='1') then
    next_state <= transPalabras;
    Frame<='1';
elsif (state = transPalabras and Bus_TRDY='0') then
    next_state <= transPalabras;
    Frame<='1';
elsif (state = transPalabras and Bus_TRDY='1' and palabra_buscada='0') then
    next_state <= transPalabras;
    Frame<='1';
    MC_WE<='1';
    mux_origen<='1';
    count_enable<='1';
elsif (state = transPalabras and Bus_TRDY='1' and last_word='1') then
    next_state <= frame0;
    Frame<='1';
    MC_WE<='1';
    mux_origen<='1';
    count_enable<='1';
    MC_tags_WE<='1';
    Replace_block<='1';
    mux_out<='1';
elsif (state = transPalabras and Bus_TRDY='1' and palabra_buscada='1') then
    next_state <= terminarTrans;
    Frame<='1';
    MC_WE<='1';
    mux_origen<='1';
    mux_out<='1';
    count_enable<='1';
    ready<='1';
. . .
```

```

-- TERMINAR TRANSMISION (PALABRA YA ENVIADA)
elsif (state = terminarTrans and Bus_TRDY='0') then
    next_state <= terminarTrans;
    Frame<='1';
    if (RE='0' and WE='0') then
        ready<='1';
    elsif (RE='1' and hit='1' and cjtoDist='1') then
        MC_RE<='1';
        ready<='1';
        -- en cualquier otro caso, ready=0
    end if;
elsif (state = terminarTrans and Bus_TRDY='1') then
    MC_WE <= '1';
    mux_origen <= '1';
    count_enable <= '1';
    Frame<='1';
    if (RE='0' and WE='0') then
        ready<='1';
    end if;
    if (last_word='0') then
        next_state <= terminarTrans;
    else
        next_state <= frame0;
        MC_tags_WE<='1';
        Replace_block <= '1';
    end if;
end if;

```

Hardware añadido:

Para esta parte, se han añadido principalmente cuatro componentes, que se corresponden a las cuatro señales descritas al principio de este apartado:

1. reg_ADDR_ini:

Registro de 32 bits situado en la memoria caché (archivo MC_Datos) que almacena la dirección de un fallo de lectura hasta que termine su procesamiento. Es un componente imprescindible para permitir el funcionamiento de la caché tras el envío de la palabra buscada al procesador (si se quieren gestionar aciertos de escritura en otros bloques). Aunque solo sería necesario guardar los bits de tag y de conjunto para este caso, hacerlo de 32 b nos permite utilizarlo también para la parte optativa 1.

Su enable es reg_ADDR_ini_en, señal ya descrita al principio, activada por la UC (en los casos de fallo de lectura) al principio de la gestión de uno.

Su entrada es la dirección que llega del procesador, ADDR, y su salida, separada en bits de conjunto y etiqueta, van a la RAM de Tags como dirección de escritura y dato de entrada, respectivamente.

Sus bits de conjunto también llegan a un nuevo multiplexor que, en función de mux_origen, elige entre estos (cuando es 1) y la dirección del conjunto de ADDR (cuando es 0). Esto permite utilizar el conjunto apropiado según sea escritura en caché (fallo de lectura), o lectura (acuerdo).

2. Comparador para buscada:

Esta vez colocada únicamente en la Unidad de Control de la Caché, para esta señal solo ha sido necesario el comparador ya descrito y una nueva entrada en MC con la palabra de la dirección de CPU. Otra opción habría sido generar esta señal en la Caché en lugar de en su UC.

3. mux_out:

Este multiplexor se sitúa en la MC_Datos, como se ha comentado, antes de la salida Dout de la memoria en conjunto. Su señal de control, evidentemente, viene de la UC.

4. **cjtoDist:**

Es un comparador, también situado en MC_Datos que compara, como su nombre indica, los conjuntos de la dirección del procesador y del registro reg_ADDR_ini. Su salida es pues una nueva entrada de la UC_MC.

Mejora de rendimiento:

Este cambio supone una mejora muy variable, no solo en función de la secuencia de instrucciones tras un fallo de lectura, sino incluso en la dirección solicitada.

Por cómo está diseñado el controlador de la memoria principal, siempre envía las palabras de cada bloque en orden: primero la 0, luego la 1, hasta la 3. Por tanto, si el procesador solicita la primera palabra, la memoria se la enviará directamente, mientras que si solicita la cuarta, deberá esperar a la transmisión de las otras tres. Para beneficiarse al máximo de los cambios realizados, se podría implementar en dicho controlador que se enviara primero la palabra solicitada en cualquier caso, y después el resto.

Finalmente, como hemos dicho, la diferencia de ciclos también depende de las instrucciones siguientes. El procesador solo se mantendrá desbloqueado durante la transmisión del resto de palabras a MC en dos casos: si no llegan peticiones a memoria, y si llegan aciertos de lectura. Se debe tener en cuenta, no obstante, que en este proceso los aciertos de lectura están limitados a los tres conjuntos que no se están actualizando en caché.

Mejora optativa 1 - Buffer para escrituras

En este caso, el objetivo es reducir los ciclos de parada por escrituras añadiendo registros para hacer de buffer. Nuestro diseño permite no detener el procesador en absoluto tras una escritura, siempre que no vuelva a haber ninguna petición de escritura adicional, o algún fallo de lectura hasta que se actualice la palabra en memoria principal. También se pueden dar aciertos de escritura, ya que la caché ya se ha actualizado con la palabra dada y el resto de bloques y palabras están intactos.

Para esta parte, el hardware añadido ha sido el siguiente:

1. **reg_ADDR_ini:**

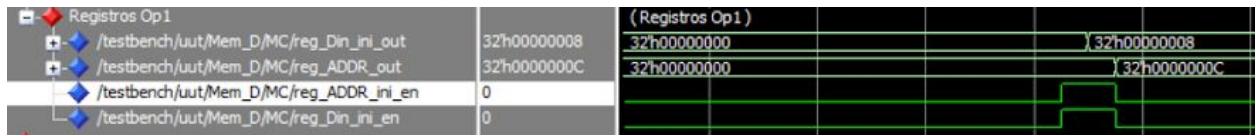
Aunque ya estaba presente en la anterior mejora, en esta, además de conectar nuevas salidas, ha tenido la siguiente modificación (también aplicada al registro reg_Din_ini, que comentaremos a continuación).

Para disponer de los datos en los nuevos registros en el mismo ciclo en que se cargan, hemos optado por actualizarlos a ciclo cambiado, con la negación del reloj, de forma que se actualicen en los flancos de bajada. En vhdl el registro ha quedado así:

```
reg_ADDR_ini: reg32 port map(  Din => ADDR, clk => clk_inv, reset => reset, load => reg_ADDR_ini_en, Dout => reg_ADDR_out);
```

Donde *clk_inv* es la negación de *clk*.

A continuación, una comparación de dos registros, el primero con el ciclo cambiado y el segundo no:



Así, vemos que el comportamiento es el esperado, el primero se carga medio ciclo antes, permitiendo su lectura instantáneamente. (En la versión final ambos registros funcionan como el primero).

En cuanto a las salidas de *reg_ADDR_ini*, además de las comentadas en el apartado anterior, ahora ha sustituido a *ADDR* en las entradas al multiplexor controlado por *block_addr*, cuya salida es *MC_bus_ADDR* (la dirección que va al bus desde la *MC_Datos*):

```
--Si es escritura se manda la dirección de la palabra y si es un fallo de lectura la dirección del bloque que causó el fallo
MC_Bus_ADDR <= reg_ADDR_out(31 downto 2)&"00" when block_addr = '0' else
reg_ADDR_out(31 downto 4) &"0000";
```

2. *reg_Din_ini*:

El nuevo registro, también de 32b, similar a *reg_ADDR_in*, pero que almacena el dato de entrada, en lugar de la dirección, para las escrituras. Como se ha comentado, también funciona a ciclo cambiado. También se encuentra en la caché de datos, *MC_Datos*.

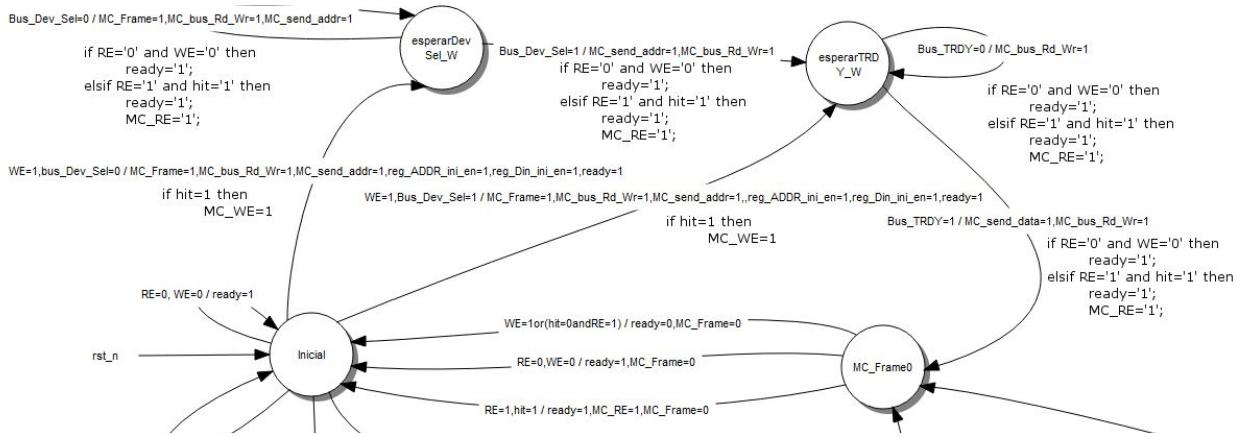
Su entrada es *Din*, el dato proveniente del procesador, y su enable es *reg_Din_ini_en*, que es una nueva salida de la *UC_MC*.

```
reg_Din_ini: reg32 port map( Din => Din, clk => clk_inv, reset => reset, load => reg_Din_ini_en, Dout => reg_Din_ini_out);
```

Su salida sustituye directamente a *Din*:

```
MC_Bus_data_out <= reg_Din_ini_out; -- se usa para mandar el dato a escribir
```

A continuación, la parte del autómata de la *UC* correspondiente a esta mejora (de nuevo, en el anexo se encuentra el autómata definitivo completo). Hemos considerado que es mucho más legible mostrando condiciones en las transiciones en lugar de dividir cada una en varias, ya que buena parte de las salidas son comunes.



De esta forma, las transiciones que salen del estado inicial se mantienen prácticamente iguales a las del autómata original. Las únicas diferencias son que ahora activamos los enables de los nuevos registros (reg_ADDR_ini_en y reg_Din_ini_en) para guardar la dirección y datos en los mismos, y que desbloqueamos el procesador inmediatamente (ready=1), ya que, en cualquier caso, la MC ya está lista para atender nuevas peticiones.

Tras esto, se debe esperar a DevSel y a TRDY, manteniendo el ready a 1 mientras o bien lleguen aciertos de lectura (como se ha comentado, todas las palabras en caché tienen su valor esperado), o bien no lleguen otro tipo de peticiones a memoria.

Se mantiene así hasta que hemos recibido ambas señales, cuando finalmente mandamos el dato a memoria (de forma idéntica desde el punto de vista de la UC, lo único que cambia es la ruta de datos, ya que ahora se envía el contenido del registro) y volvemos a frame0 para bajar Frame a 0.

En vhdl, este es el código correspondiente a la parte del autómata mostrada:

```

elsif (state = Inicio and WE='1') then
    if (Bus_DevSel='0') then
        next_state <= esperarDEVSel_W;
    else
        next_state <= esperarTRDY_W;
    end if;
    Frame<='1';
    MC_bus_Rd_Wr<='1';
    MC_send_addr<='1';
    reg_ADDR_ini_en<='1';
    reg_Din_ini_en<='1';
    ready<='1';
    if (hit='1') then
        MC_WE<='1';
        inc_wh <='1';
    else
        inc_wm<= '1';
    end if;

```

```

elsif (state = esperarDEVSel_W) then
    Frame<='1';
    MC_send_addr<='1';
    MC_bus_Rd_Wr<='1';
    if (Bus_DevSel='0') then
        next_state <= esperarDEVSel_W;
    else
        next_state <= esperarTRDY_W;
    end if;
    if (RE='0' and WE='0') then
        ready<='1';
    elsif (RE='1' and hit='1') then
        MC_RE<='1';
        ready<='1';
    -- en cualquier otro caso, ready es 0
    end if;

elsif (state = esperarTRDY_W) then
    Frame<='1';
    MC_bus_Rd_Wr<='1';
    if (Bus_TRDY='0') then
        next_state <= esperarTRDY_W;
    else
        next_state <= frame0;
        MC_send_data <= '1';
    end if;
    if (RE='0' and WE='0') then
        ready<='1';
    elsif (RE='1' and hit='1') then
        MC_RE<='1';
        ready<='1';
    -- en cualquier otro caso, ready es 0
    end if;

```

Primera prueba (código subsanación del proyecto 1)

Suponemos que en la posición 4 de memoria está cargado el valor 4.

LW R0, 0(R0)

nop

LA R2, 4(R1); R2=4, R1=0

LW R3, 0(R2); R3= MD(x04)=4 #corto a distancia 1 con el LA

ADD R4, R3, R3; R4=8 # parada + cortos a distancia 2

SW R4, 4(R4); MD(x14)=8 #cortos a distancia 1

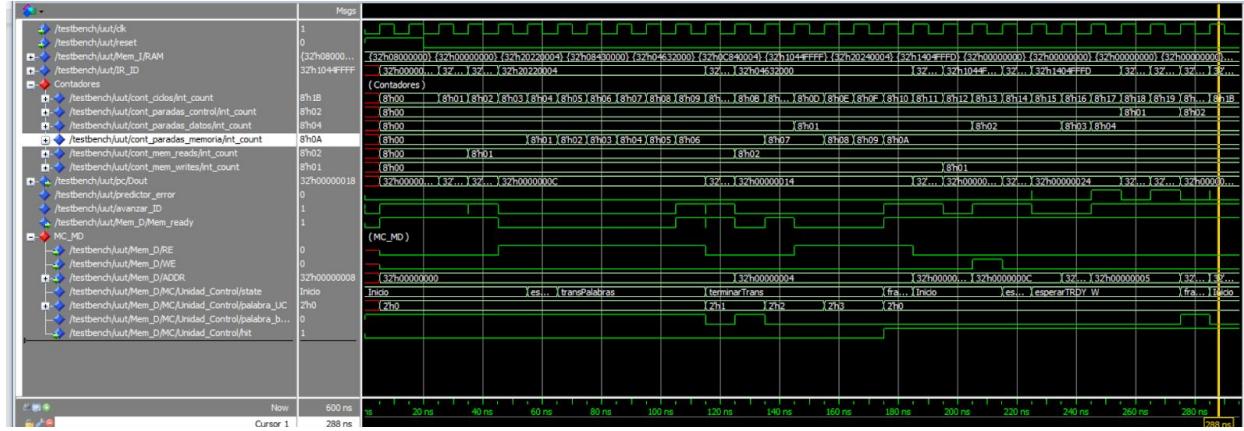
loop BEQ R2, R4, loop; No se salta la primera vez: R2=4, R4=8.

; Sí se salta la segunda (predictor falla): R2=4, R3=4 , y en las siguientes el predictor acierta

LA R4, 4(R1); R4=4, R1=0

BNE R0, R4, loop; Sí se salta: R4=4, R0=0; Se vuelve al BEQ anterior

Observamos una evidente mejora de rendimiento al disminuir las paradas de memoria a 10 (desde 19 en la primera versión y 14 solamente con la optativa 2), lo que representa una disminución de casi el 50%. Además, el autómata pasa correctamente por los estados esperados.



Revisitando parte de la ejecución de este mismo ejemplo (correspondiente al código a continuación), vemos el origen de buena parte de la mejora.

SW R4, 4(R4); MD(x14)=8#cortos a distancia 1

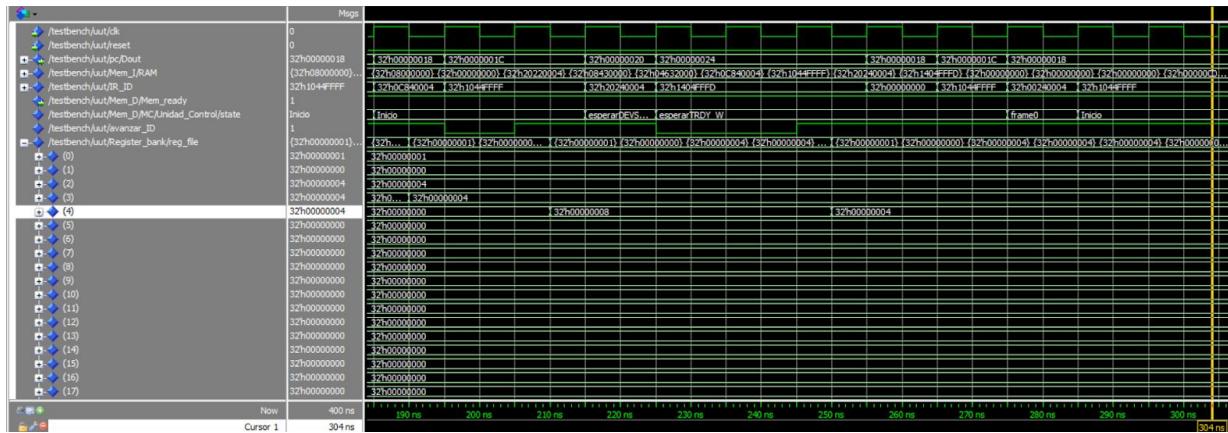
BEQ R2, R4, loop; No se salta la primera vez: R2=4, R4=8.

; Sí se salta la segunda (predictor falla): R2=4, R3=4 , y en las siguientes el predictor acierta

LA R4, 4(R1); R4=4, R1=0

BNE R0, R4, loop; Sí se salta: R4=4, R0=0; Se vuelve al BEQ anterior

Cuando antes debía terminar la ejecución de la petición del store para continuar, ahora puede gestionarse mientras el procesador continúa con el resto de instrucciones. Al tratarse de saltos, no realizan peticiones a memoria (RE y WE son 0), con lo que mem_ready se mantiene a 1, como se puede observar a continuación:



Ocurriría lo mismo si se dieran aciertos de lectura o cualquier otra instrucción que no suponga accesos a memoria.

Pruebas realizadas - 2.

Finalmente, una vez instaladas las optimizaciones, realizamos las pruebas definitivas.

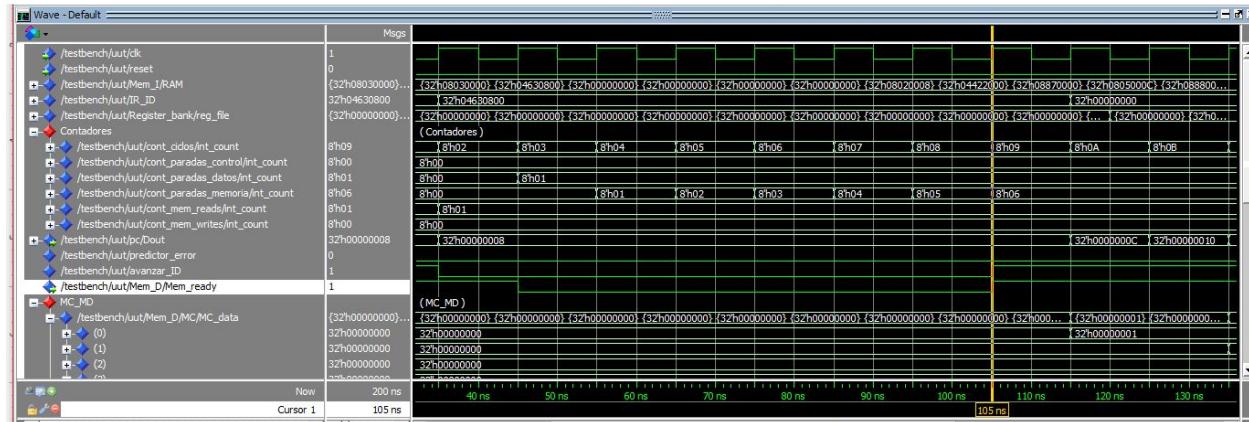
Prueba definitiva (*RAM_I_tb2.vhd*):

Supongo que la palabra de la posición 0 es 1 y la de la posición 8 es 8.

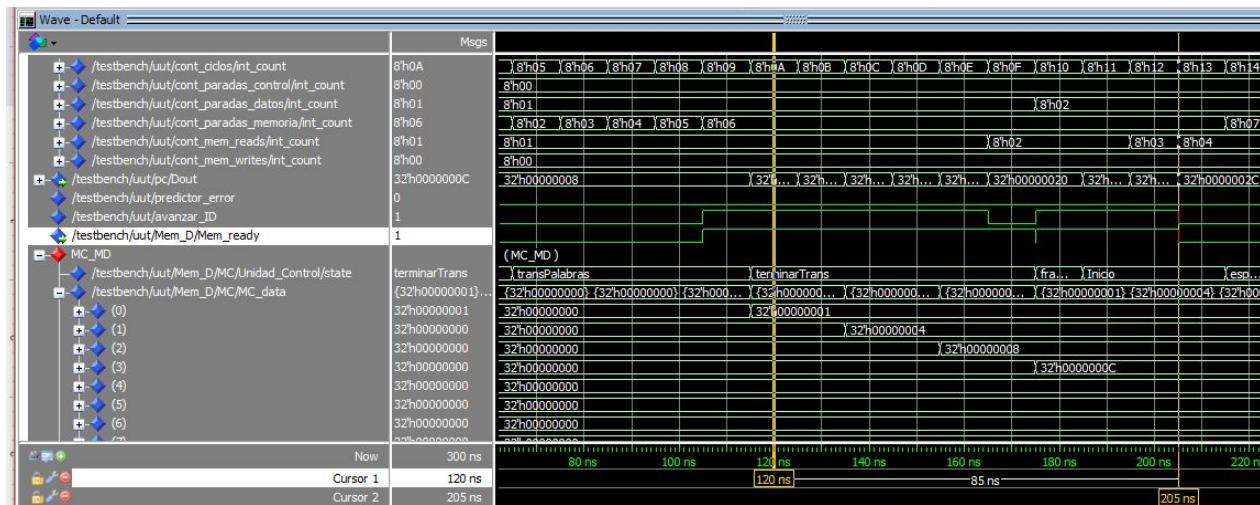
Para comprobar que nuestra versión definitiva funciona correctamente, hemos evaluado todos los casos posibles (fallos y aciertos de lectura y escritura en los diferentes conjuntos y operaciones del procesador y aciertos de lectura mientras la memoria de datos está ocupada).

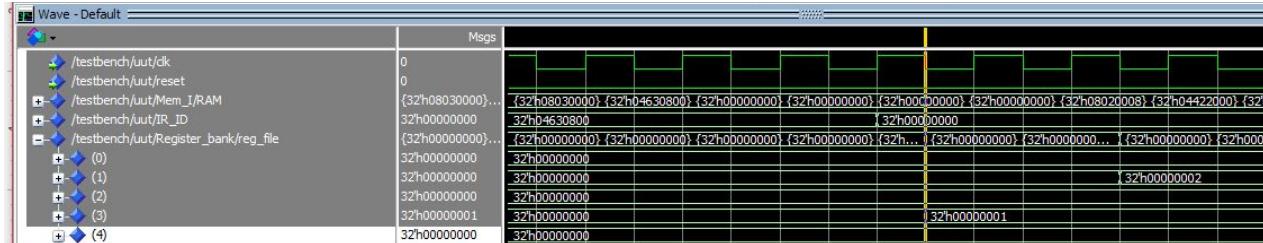
0	08030000	LW R3,0(R0)
4	04630800	add R3, R3, R1; r1=2;
8	00000000	
C	00000000	
10	00000000	
14	00000000	; tanto la suma como estas nops se ejecutan mientras la memoria de datos está ocupada.
18	08020008	LW R2, 8(R0); cargo la tercera palabra load hit. r2=8
1C	04422000	add R2,r2,r4; r4=16=0x10
20	08870000	LW r7,0(r4); load miss, carga otro conjunto.
24	0805000C	LW R5, C(R0); load hit cargo palabra del primer conjunto.
28	0888000C	LW r8,C(r4); load miss, cargo la última palabra del conjunto que se está cargando.
2C	04841000	add R4,r4,r2; r2=32=0x20
30	0C480004	SW r8, C(r2); store miss conjunto 3
34	04443000	add R2,r4,r6; r6=48=0x30
38	08290004	LW R9, 4(R4); load hit palabra del conjunto 1.
3C	0C85000C	SW R5, C(R4); store hit conjunto 1
40	08290000	LW R9, 0(R4); load hit palabra del conjunto 2.
44	0CC70004	SW r7, 4(R6) ; store miss conjunto 4
48	04C41000	add R6,r4,r2;
4C	08480004	LW r8, 4(r2); remplazo del conjunto 0

La primera detención se produce al cargar el primer bloque en caché. Gracias a nuestra anticipación, como hemos solicitado la primera palabra, nos permite utilizar el procesador mientras se están cargando las palabras en caché. Anteriormente se producía una parada de doce ciclos, ahora es de seis.

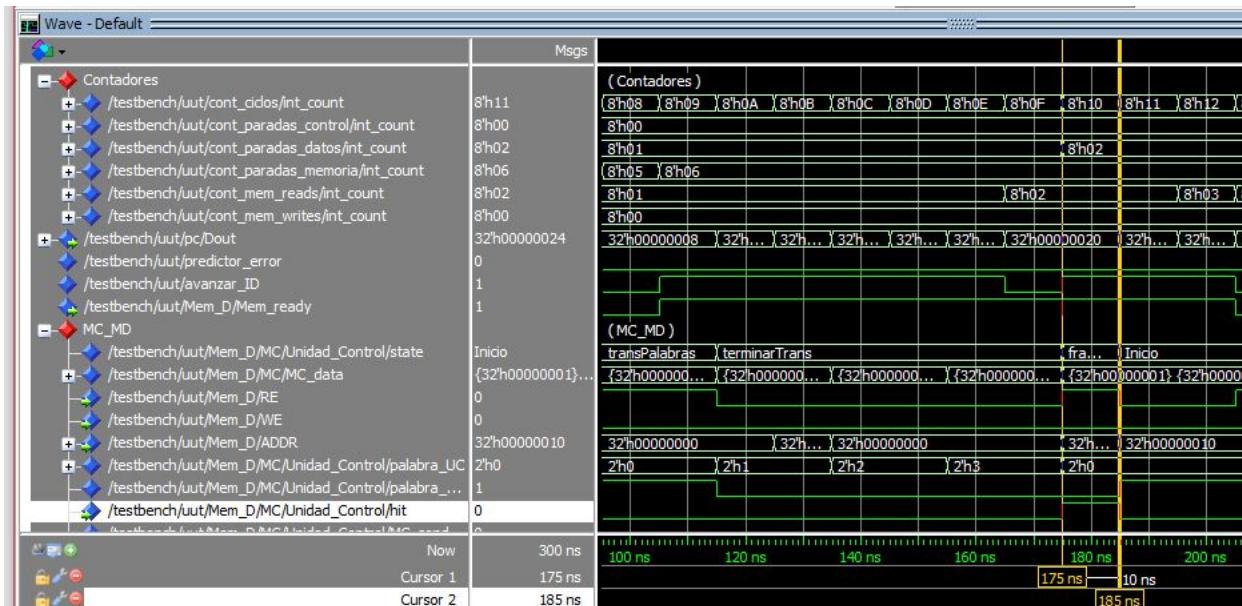


Como podemos observar, la caché se carga con el contenido esperado mientras el procesador ejecuta las sumas y las nops (ya que men_ready es igual 1). Además en el banco de registros figuran los datos esperados, lo cargado en memoria y el resultado de la suma.

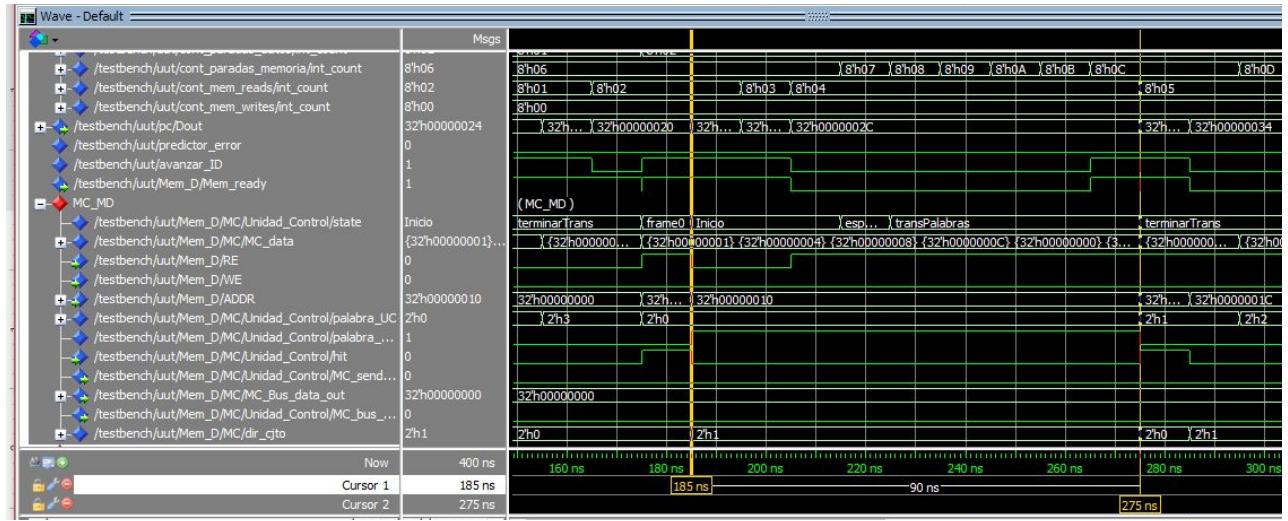




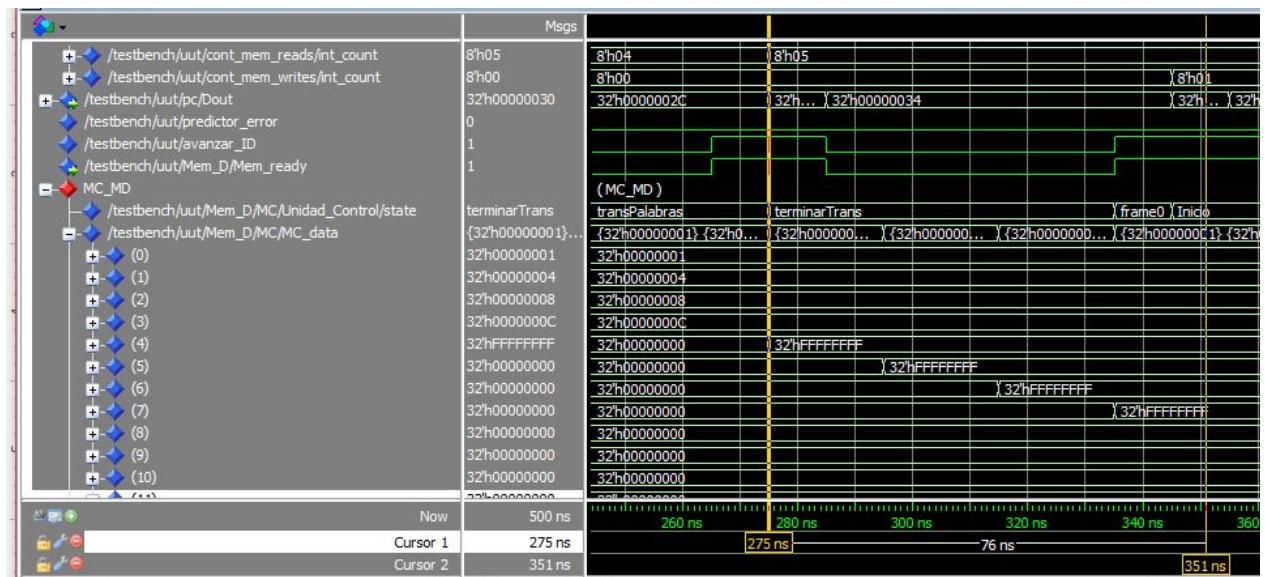
Después se produce un acierto de lectura, al cargar la tercera palabra del bloque anterior. Como es un acierto de lectura no hay ningún tipo de detención y en un solo ciclo nos da la palabra pedida. En cualquier caso, se produce una parada de datos por la suma siguiente.

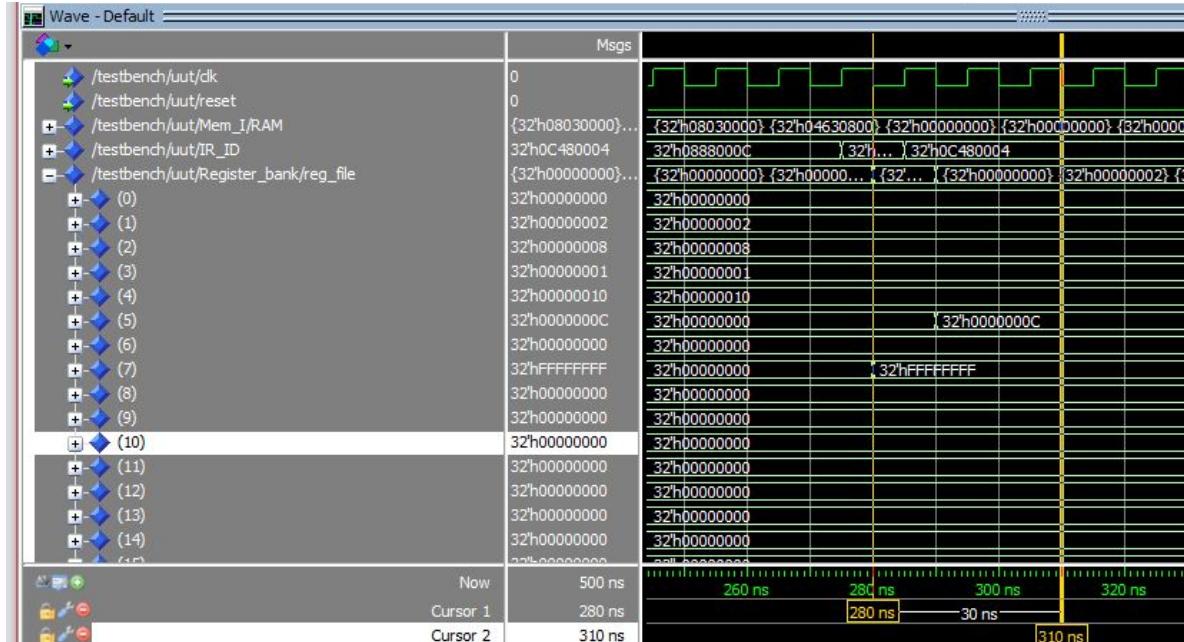


Ahora, vamos a cargar otro conjunto en caché. En esta ocasión, vamos a volver a pedir la primera palabra. Como se puede apreciar, nuevamente se efectúa una parada de seis ciclos y la dirección de conjunto, como es natural, es distinta. Cuando el control de memoria caché avisa al procesador de que la memoria está preparada, se realiza un instrucción de carga de una dirección del otro conjunto. Lo más destacable es que, a pesar de que todavía se están cargando las palabras en memoria, como se ha producido un acierto de lectura, la memoria nos da la palabra que hemos solicitado sin necesidad de esperar ni de detener el procesador.

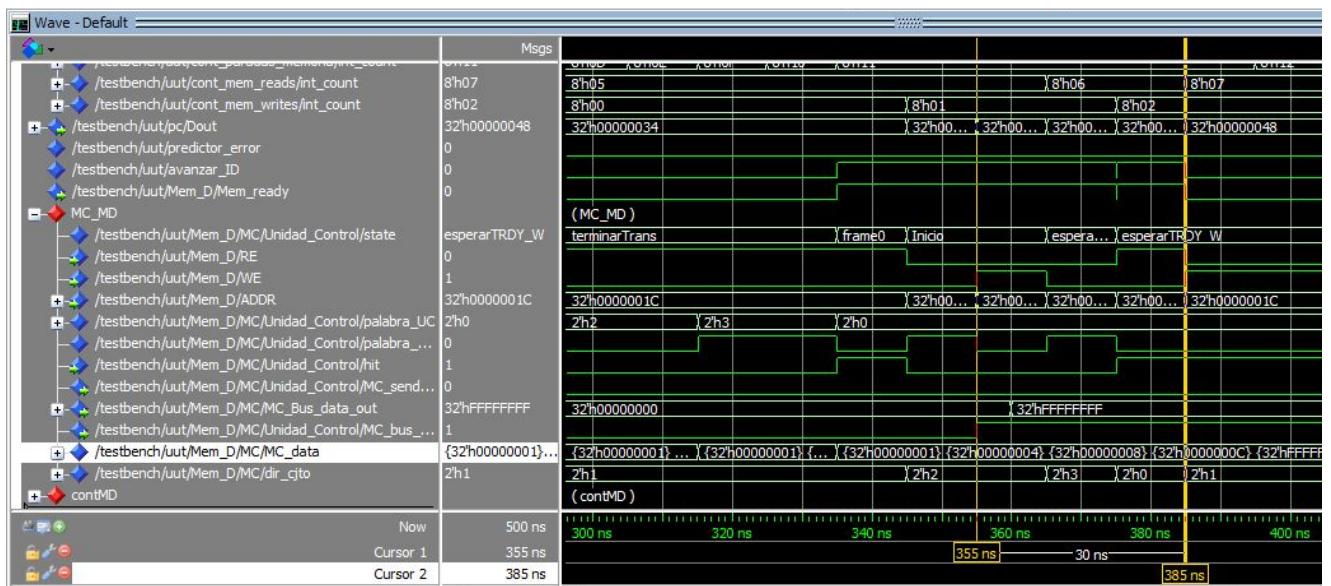


Podemos comprobar que nos hemos obtenido las dos palabras solicitadas, que se encuentran cargadas en el banco de registros, y que la memoria caché contiene el bloque esperado en su lugar.

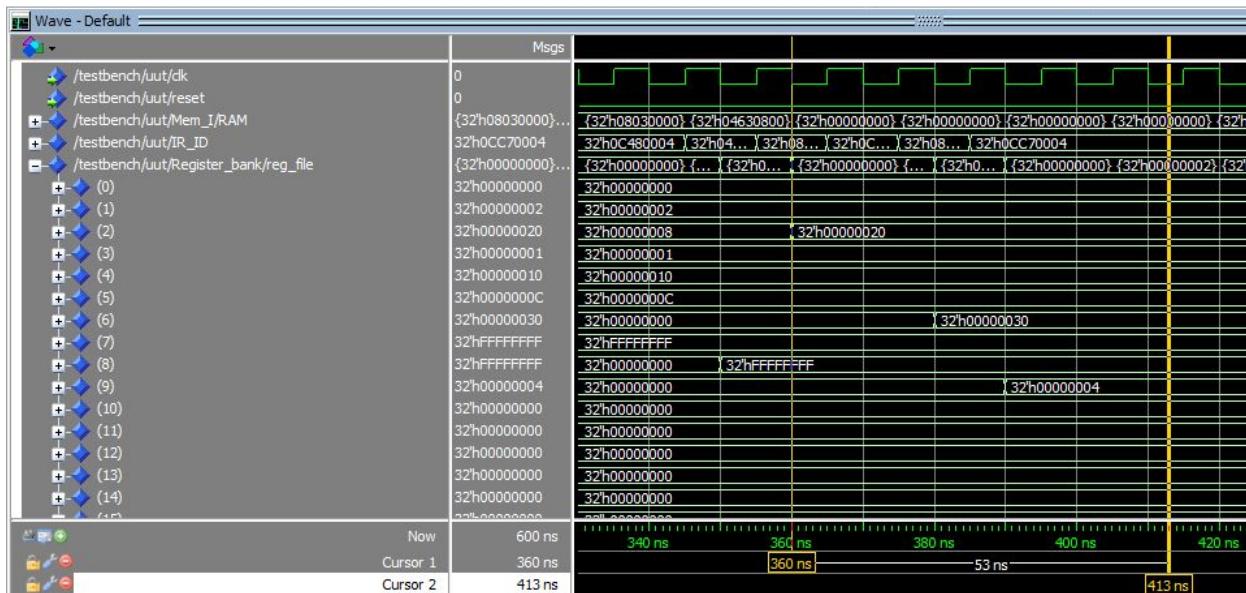
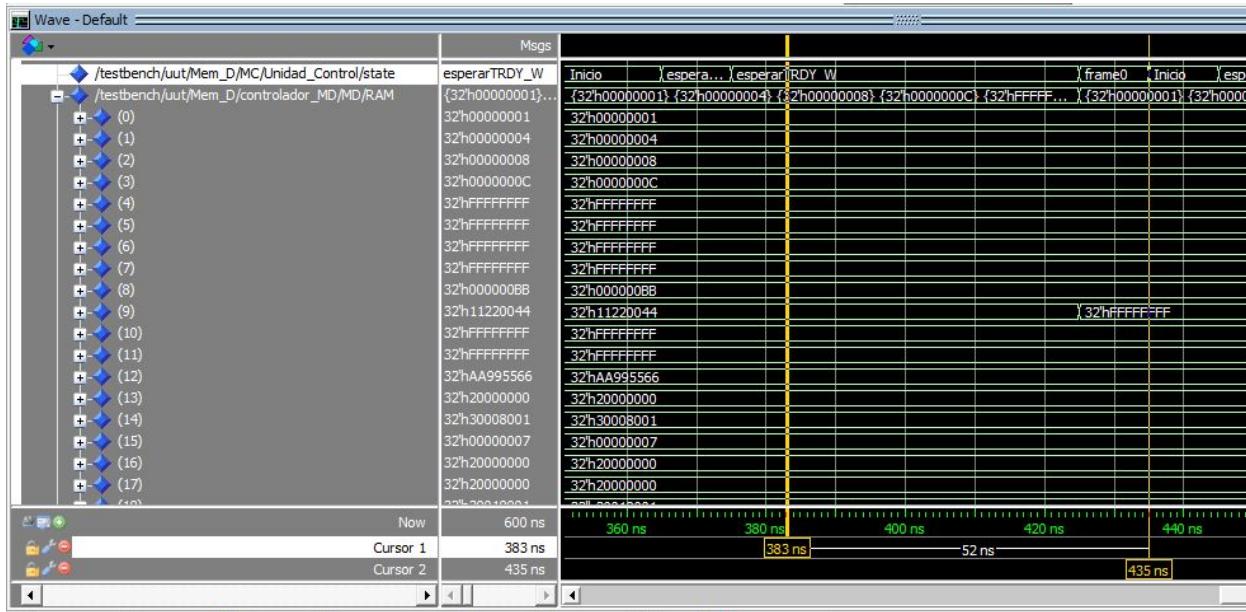




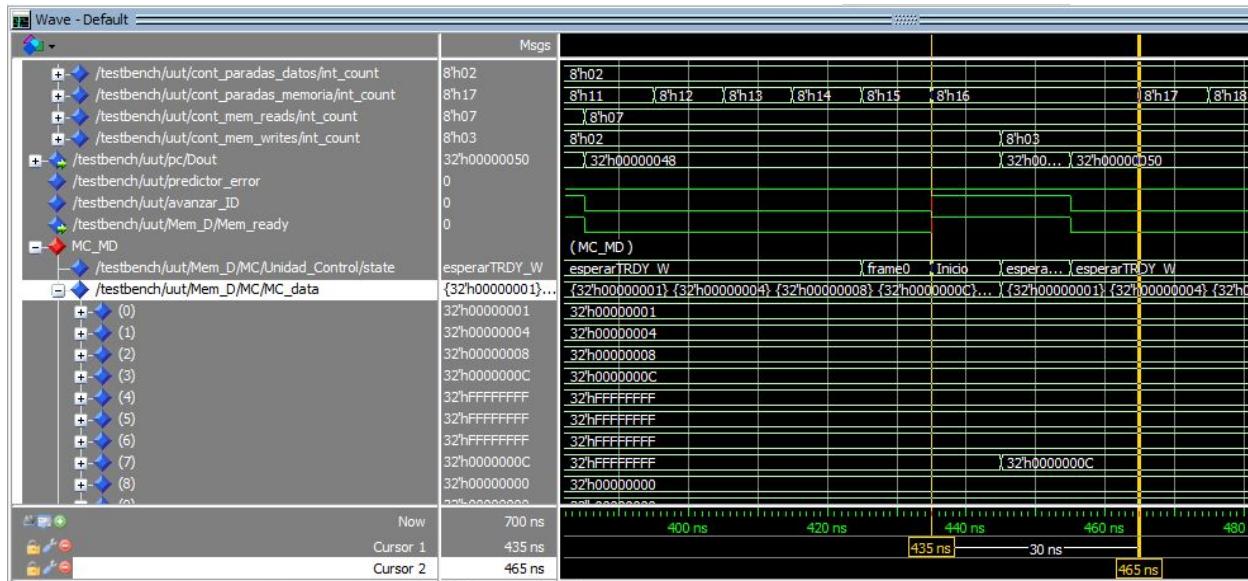
La siguiente instrucción pide una palabra del conjunto que se está cargando, por tanto, es un fallo de lectura y se ha de detener el procesador hasta que se termine de cargar el conjunto. A continuación vamos a almacenar un dato en memoria cuyo bloque no está cargado en caché. Así que se producirá una escritura directa en memoria de datos. Gracias a nuestra optimización podremos seguir utilizando el procesador a la vez que la memoria de datos trabaja. En la siguiente figura se puede observar como, a pesar de producirse un write miss, mem_ready sigue a uno.



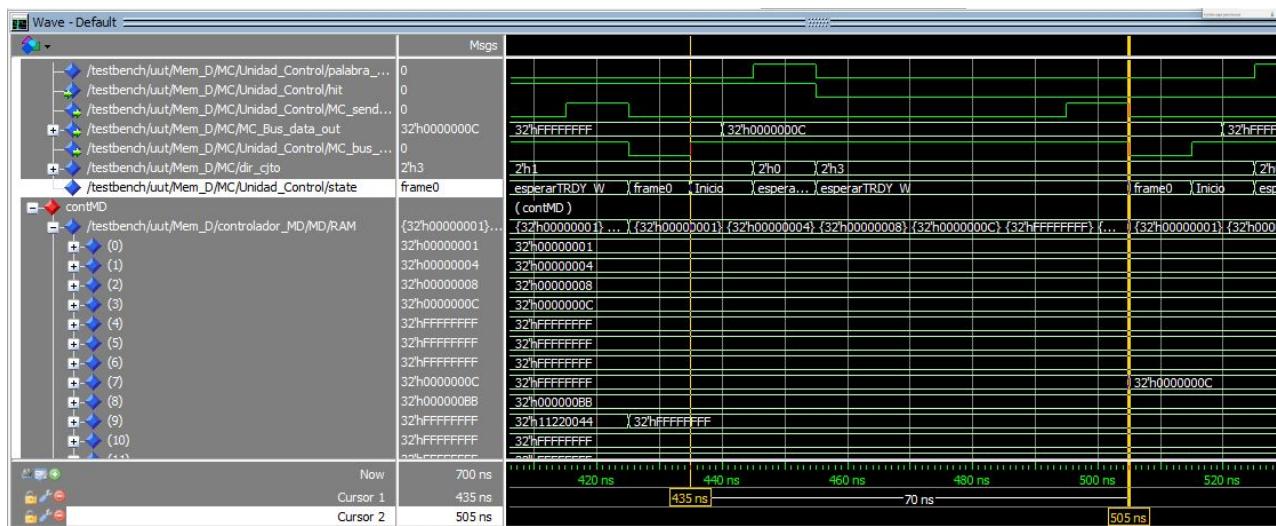
Cabe destacar que mientras la memoria de datos está trabajando, hemos pedido un dato de caché del conjunto cero. Al ser un acierto de lectura nos han concedido la palabra en ese mismo ciclo. Tanto en memoria de datos como en el banco de registros, hemos cargado la información esperada. El banco de registros contiene el resultado de la suma y el dato leído de caché.



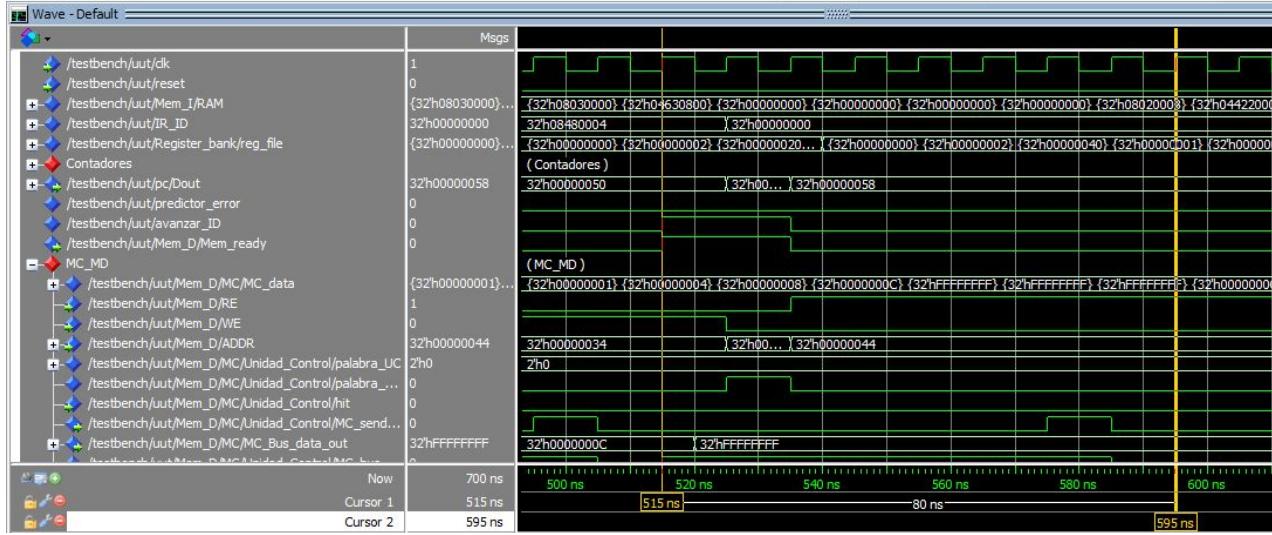
Como la siguiente instrucción a la de carga es una instrucción de almacenamiento y todavía no ha terminado el almacenamiento anterior, se produce una ligera detención. En este caso es un acierto de escritura. Debido a nuestra mejora, el procesador no llega a detenerse (hasta que le llegué otra instrucción que requiera el uso de memoria). Nada más recibir la instrucción de store, se produce una escritura en caché a ciclo cambiado. A continuación, se ejecuta una instrucción de acierto de escritura, mientra la memoria de datos está trabajando. Dado que la siguiente instrucción es de almacenamiento, se lleva a cabo una detención del procesador.



Unos cuantos ciclos después se produce la escritura en la memoria de datos.



En la siguiente instrucción realizamos un almacenamiento a una dirección, de un bloque que no está cargado y que correspondería al conjunto 4, seguida de una suma. Como no hemos intercalado suficientes instrucciones útiles el procesador permanece parado un buen rato, situación que se ha dado en casos anteriores.



Al igual que en el read miss anterior, se escribe en memoria el dato correcto en la dirección correcta. Por último, cargamos en memoria caché un bloque del conjunto cero distinto al ya cargado. Es decir, se produce un reemplazo. Este se produce correctamente. En esta ocasión, como hemos solicitado la segunda palabra en vez de la primera, el procesador ha estado parado durante 8 ciclos.



Finalmente, el programa ha tardado en ejecutarse 66 ciclos y se han producido 42 paradas de memoria. Con nuestro mips le hubiera costado 85 ciclos. Por otro lado, si hubiésemos intercalado los loads y stores con instrucciones útiles, solo se hubiesen producido

20 paradas de memoria (las primeras de los load miss). Es decir, tendríamos 20 paradas frente a las 61 del mips sin optimizar.

Resumen de nuestras aportaciones.

José Manuel.

Fecha.	Tiempo dedicado.	Tipo de tarea.
17/5/19	2h 30 min	Leer y entender el enunciado y los fuentes y empezar el autómata en papel
23/5/19	15m	Tutoria para preguntar dudas del autómata.
23/5/19	2h	Pasar el automata a vhdl.
23/5/19	2h	Depurar
27/5/19	2h	Seguir depurando
28/5/19	1h	Meter código de parada del mips.
30/5/19	4h	Hacer pruebas y plasmarlas en memoria. Contribuir al optional 2.
31/5/19	4h	Debugear optativa 2
31/5/19	2h	Debugear optativa 1
1/6/19	3h	Plasmar pruebas en memoria y escribir la notación algorítmica.
2/6/19	3h	Retoques a la memoria
3/6/19	1h	Puesta a punto y entrega

SUMA	26h 45 min	
------	------------	--

Néstor:

Fecha.	Tiempo dedicado.	Tipo de tarea.
17/5/19	2h 30 min	Leer y entender el enunciado y los fuentes y empezar el autómata en papel
23/5/19	15m	Tutoria para preguntar dudas del autómata.
23/5/19	2h	Pasar el automata a vhdl.
25/5/19	2h	Depuración del autómata en papel y vhdl. Primeras simulaciones
29/5/19	2h	Planteamiento en papel de la mejora optativa 2, añadidas algunas señales a MC y UC_MC.
29/5/19	2h	Memoria, primeras pruebas en limpio.
29/5/19	1h30	Más avances/retoques en el autómata y ruta de datos del optativo 2
30/5/19	4h	Implementación y depuración del optativo 2
31/5/19	1h30	Más depuración del optativo 2 y algunos detalles de la parte obligatoria
31/5/19	2h30	Planteamiento y comienzo de implementación del optativo 1

31/5/19	4h	Ajustes de implementación y depuración del optativo 1
2/6/19	3h	Memoria de mejoras optativas
2/6/19	3h	Correcciones generales de código y autómatas
3/6/19	3h	Retoques a la memoria (partes optativas)
3/6/19	1h	Últimos retoques y entrega
TOTAL:	34h	

Más o menos nos ha llevado el tiempo previsto. Sin embargo, no hemos dedicado la cantidad de tiempo estimada en cada apartado. La depuración ha sido más rápida de lo previsto y el resto de fases han sido un poco más costosas. Además vemos razonable que nos cueste un poco más de tiempo de lo previsto debido a que hemos implementado mejoras de rendimiento.

Autoevaluación.

José Manuel.

En las líneas generales considero que he llevado esta asignatura al día, en gran medida gracias a los problemas semanales y las prácticas. Además creo he afianzado unos conocimientos básicos de hardware y computadores.

Sería difícil darme una puntuación a mí mismo porque esta depende de varios factores. Siendo crítico, nuestro primer proyecto podía haber sido mejorable en algunos puntos. En este proyecto mi esfuerzo dedicado ha sido mayor y he mostrado más preocupación por los detalles. Siempre y cuando no se produzcan errores en sus pruebas (que hemos tratado de evitar insistentemente) y no se le otorgue mucha importancia al primer proyecto a la hora de evaluar; opino que merecería una calificación notable, o incluso sobresaliente.

Néstor.

Creo que he alcanzado los objetivos precisamente por la constante carga de trabajo de la asignatura, entre los problemas semanales, las prácticas y los dos trabajos. Sin duda habría sido imposible alcanzarlos sin esta por el carácter práctico del temario.

En cuanto a la autoevaluación, es algo que encuentro difícil. Aunque en general he llevado el estudio al día (no puede ser de otra forma si se hacen los problemas y prácticas), cuando empezamos los proyectos me costó un tiempo adaptarme tanto al nuevo lenguaje vhdl como al entorno ModelSim (no sabíamos ni siquiera guardar macros para reutilizar señales), con lo que me supuso más esfuerzo y tiempo de lo previsto todo lo correspondiente al primer proyecto. Este desajuste supuso al final la entrega de un trabajo que no se correspondía con lo que considero que habíamos aprendido. En este último proyecto, por otra parte, todo el proceso ha sido más llevadero y eficiente, y nos ha permitido (creo que a los dos) aportar más. Así, supongo que mi nota dependería principalmente del peso que se le dé a cada proyecto

Anexo: Diagrama de estados UC completa

