PROYECTO 2:
JERARQUÍA DE MEMORIA DE DATOS

Arquitectura y Organización de Computadores II

Mayo –Junio 2019

Sergio García Esteban 755844

Irene Fumanal Lacoma 758325

Contenido

1- Introducción

En este proyecto, trabajamos con el procesador modificado en el proyecto 1 (MIPS Segmentado, gestión de riesgos y predictor), introduciendo una memoria cache de datos conectada a la memoria principal de datos a través de un bus semi-síncrono basado en el bus PCI que hemos estudiado.

El diseño de nuestro proyecto lo hemos reflejado en este documento en dos partes. La primera parte incluye el diseño del controlador de la memoria cache, las modificaciones en el MIPS para que se detenga cuando la memoria cache no pueda realizar la operación solicitada en un ciclo de reloj, y los contadores que debemos añadir para evaluar el rendimiento. En la segunda parte, explicamos el diseño definitivo, que incluye un buffer para las escrituras en memoria y así permitir al procesador ejecutar instrucciones y a la cache gestionar aciertos hasta que necesite el bus, que entonces detendremos el procesador.

Incluimos también una serie de pruebas completas para demostrar que nuestro trabajo funciona, además de datos sobre el rendimiento y una sección para explicar la distribución del proyecto y la metodología de trabajo.

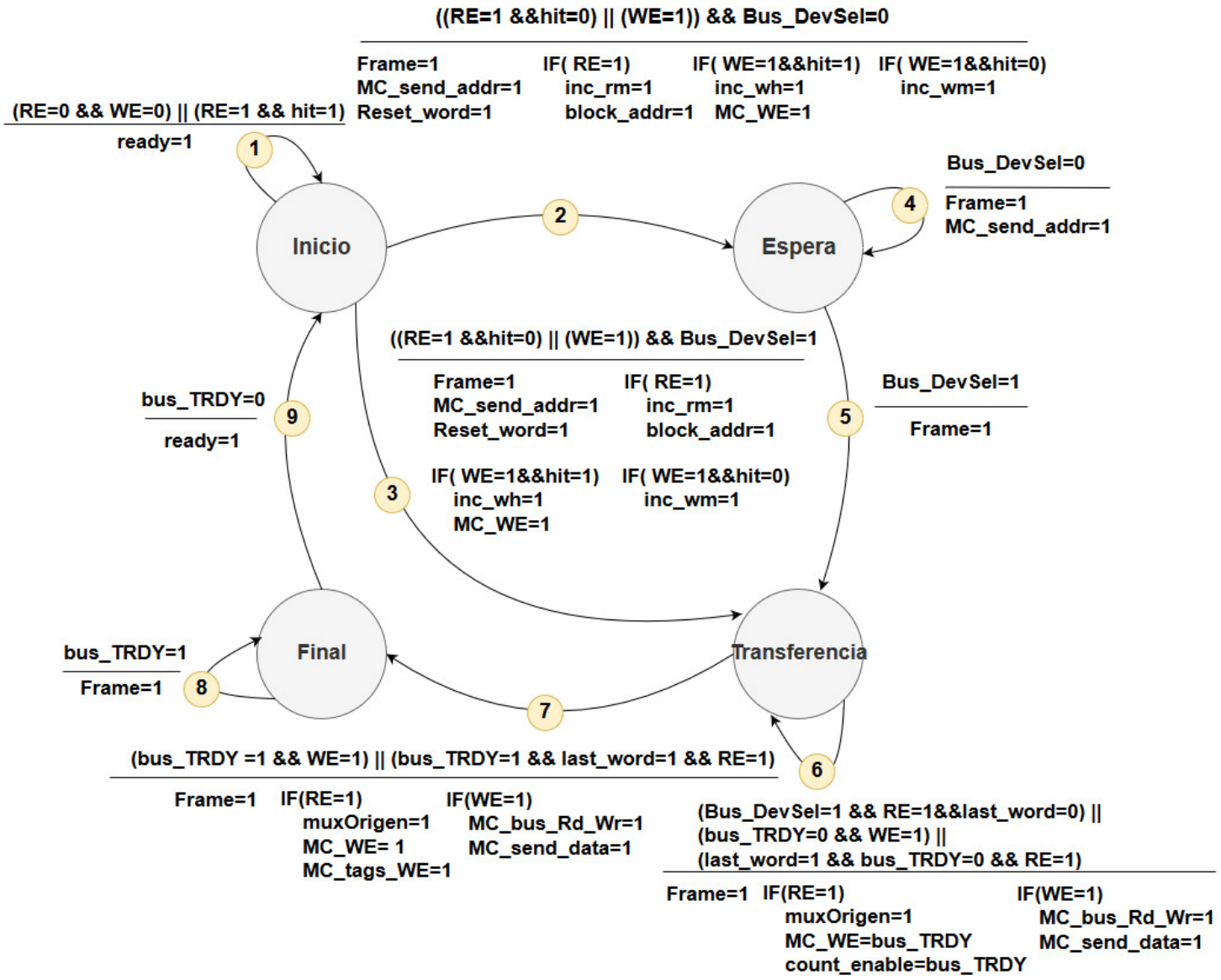
2- Parte 1: diseño base

El primer paso, antes de empezar a diseñar el controlador de la cache, fue estudiar el nuevo código VHDL proporcionado y comprender su funcionamiento. Identificamos todos los componentes y entendimos el autómata incluido del Slave (memoria de datos).

2.1- Diagrama de estados

Decidimos que el autómata del Master sería Meally, ya que, por un lado, el del Slave también lo era y sería más sencillo desarrollar la sincronización, y por otro, nos ahorraríamos estados.

El siguiente dibujo representa el diagrama de estados. Hemos numerado las transiciones para poder explicarlas posteriormente.



A continuación, explicamos los estados del diagrama y sus transiciones correspondientes.

Inicio: El controlador de la memoria cache (Master) permanecerá en este estado cuando la instrucción situada en la etapa de memoria de nuestro procesador no acceda a ésta, es decir, cuando no sean lw o sw. Además, si se trata de un acierto de lectura en cache, se proporcionará el dato directamente. En estos casos no hay que detener nuestro MIPS y por eso el valor de *ready* será 1. **(transición 1).**

Cuando la instrucción sea una escritura (acierto o fallo) o un fallo de lectura, saltaremos al estado *Espera* o directamente al estado *Transferencia*, dependiendo del valor de la entrada *Bus_DevSel* como observamos en el diagrama (**transiciones 2 y 3**). Independientemente del tipo de operación que sea (lectura o escritura), ponemos *Frame* a 1 para indicarle al Slave que la operación no ha terminado, enviamos la dirección de memoria por el bus y reiniciamos el contador de palabras (*Reset_word*, es una señal que hemos añadido). En caso de fallo de lectura activamos *block_addr* para indicar que la dirección enviada es la de bloque, y en caso de acierto de escritura activaremos la señal *MC_WE* para escribir en cache.

Espera: Se permanecerá en este estado a la espera de que el Slave reconozca la dirección enviada (hasta que *Bus_DevSel*sea 1). Mientras no sea reconocida, el Master mantendrá la dirección en el bus. (**transición 4**).

Cuando la dirección se confirme, pasa al estado *Transferencia* (**transición 5**).

Transferencia: Para comprender bien este estado, lo explicaremos en dos partes, dependiendo si es lectura o escritura.

A. Lectura: Mientras *last_word* valga 0, permaneceremos en este estado, y muxOrigen valdrá 1 para escribir en memoria cache lo que envía el Slave por el bus, y el permiso de escritura y el del registro que cuenta las palabras (*MC_WE* y *count_enable*) dependerán del valor de la entrada *bus_TRDY*. En el momento que *last_word* valga 1 (la última palabra habrá sido enviada), esperamos a que *bus_TRDY* valga 1 para avanzar al estado *Final.*
B. Escritura: Si es escritura, esperamos a que bus_TRDY valga 1 para pasar a *Final.*

Ambos casos se reflejan en la **transición 6**.

Cuando *bus_TRDY* sea 1 y se trate de una lectura, escribimos la última palabra en cache y actualizamos los tags. Si *bus_TRDY* es 1 y se trata de una escritura, ordenamos que se envíe el dato a memoria principal y que la operación va a escribir en ella (activamos *MC_send_data* y *MC_bus_Rd_Wr*). (**transición 7**)

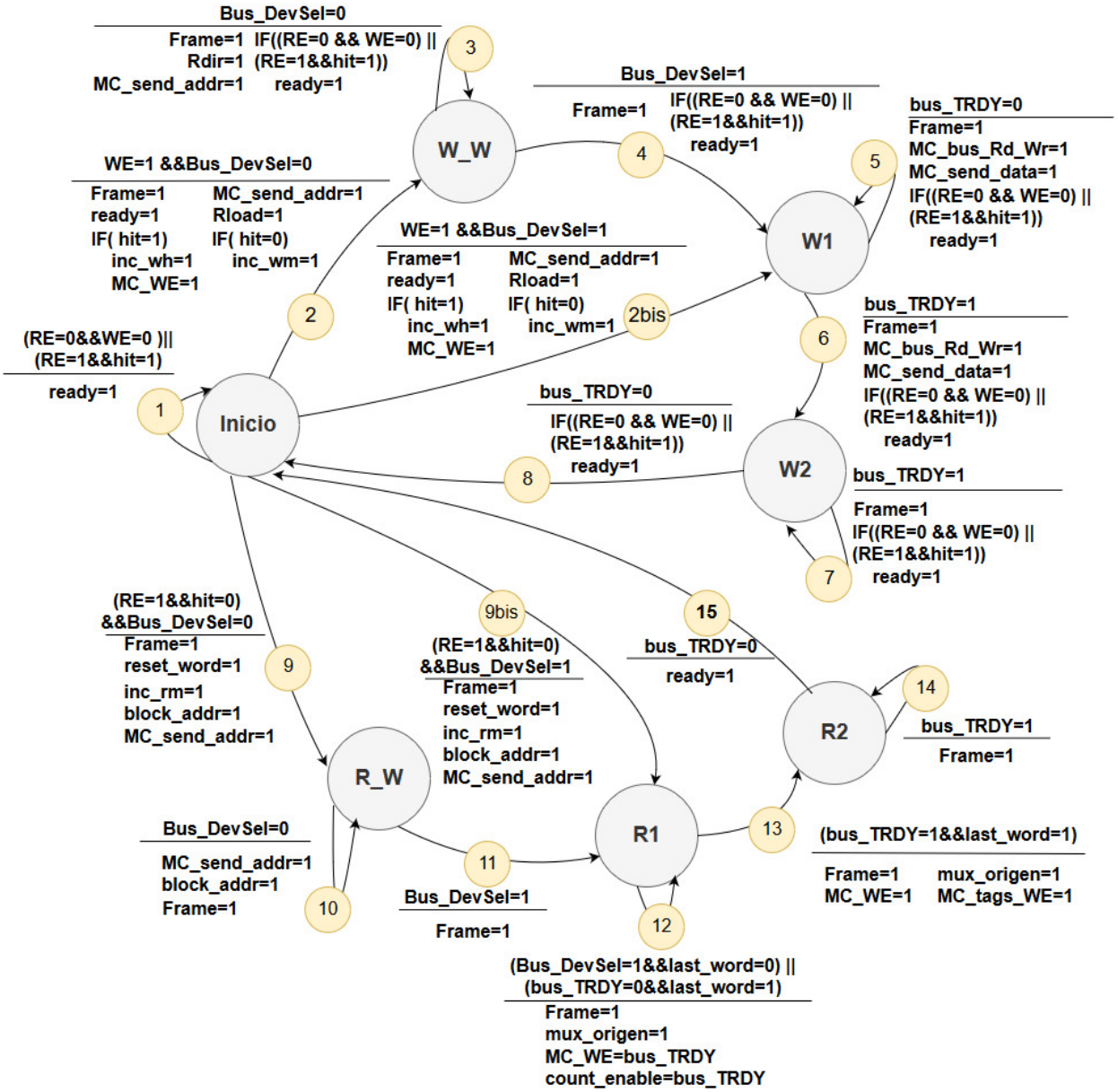
Final: permanecemos en este estado esperando a que el Slave nos indique que ha terminado la operación (**transición 8**). Cuando *bus_TRDY* valga 0, pondremos *ready* a 1 para terminar la detención del MIPS y volveremos al estado inicial a esperar la llegada de una nueva instrucción (**transición 9**).

3- Parte 2: diseño definitivo con buffer para escrituras

En este apartado mostramos el diagrama de estados definitivo después de añadir un buffer para las escrituras en memoria y explicamos los cambios en su funcionamiento. Cuando el controlador detecta una escritura, almacena el dato y no detiene el MIPS. Seguirá gestionando aciertos hasta que llegue una instrucción que solicite la utilización del bus, y en ese caso se detendrá el procesador.

3.1- Diagrama de estados

El autómata final tiene un camino para las escrituras y otro para los fallos de lectura. Tratamos de hacerlo similar al del apartado anterior, es decir, compartir estados en lectura y escritura guardando RE y WE en registros de 1 bit, pero nos dimos cuenta de que, de esa forma, era inevitable tener una condición de carrera al controlar el uso de la dirección de entrada o la guardada en el registro. De esta forma, tenemos más estados, pero es más sencillo.



Parece un diagrama totalmente distinto al que presentamos en la parte anterior, pero realmente no lo es. Hemos nombrado las transiciones para explicar de una forma sencilla el funcionamiento del autómata.

1. Lectura: El acierto de lectura y la interpretación de las instrucciones que no acceden a memoria se interpreta exactamente igual que en el autómata inicial (**transición 1).** El fallo de lectura es también muy similar. Detectamos que la instrucción en etapa de memoria del MIPS es una lectura y que además hay fallo en cache, y pasamos al estado R_W. En esa transición mandamos la dirección, reiniciamos el contador de palabra y mandamos que la dirección es de bloque (**transición 9**).

> **R_W**: similar al estado Espera. Seguimos enviando la dirección hasta que el Slave la confirma (**transición 10**). Cuando esto ocurre, pasa al estado **R1 (transición 11).**
>
> **R1**: igual al estado Transferencia del diseño base. Permanecemos en él con la **transición 12,** esperando a que el bus mande las palabras del bloque. Cuando el Slave envía la última palabra, pasa al estado **R2 (transición 13).**
>
> **R2:** permanece en este estado hasta que el valor de bus_TRDY valga otra vez 0 y seguir así el protocolo de sincronización descrito en el enunciado (**transiciones 14 y 15**).

2. Escritura: Cuando la instrucción que entra en cache es una escritura, saltamos al estado **W_W**. Durante esta transición cargamos los registros añadidos y activamos ready para decirle al MIPS que puede seguir mandando instrucciones (**transición 2**).

> **W_W**: similar al estado de Espera. Seguimos enviando la dirección igual que en lectura, e indicamos con Rdir que la dirección a enviar es la cargada en nuestro registro. En principio este último caso no debería darse en nuestro proyecto (**transición 3).** Cuando el Slave confirma la dirección enviada pasamos al siguiente estado **W1** con la **transición 4**.
>
> **W1**: similar al estado Transferencia. Indicamos el envío de los datos al Slave y le decimos que es una escritura (**transición 5).** En el momento que bus_TRDY vale 1 saltamos al estado **W2 (transición 6).**
>
> **W2:** similar a lectura. Esperamos a que bus_TRDY valga 0 para que cumpla el protocolo (**transiciones 7 y 8)**.
>
> Es importante remarcar, que una vez que mandamos cargar los registros, todas las transiciones posteriores harán la misma comprobación. Mirarán si la instrucción que llega requiere usar el bus para completarse y en caso de que lo haga, ready valdrá 0 y se detendrá el MIPS.

2.2- Lógica añadida para el buffer de escritura

Además del diagrama de estados, para poder implementar el buffer en las escrituras en memoria, hemos añadido y modificado partes de código VHDL.

Añadimos dos registros de 32 bits para guardar la dirección y el dato, que están situados en el archivo “MC_datos” y cuyos nombres son R_addr y R_data. Ambos registros se cargarán cuando la señal Rload esté activa (controlada desde el Master).

Las siguientes líneas de código han sido modificadas en el mismo archivo para seleccionar la dirección que queremos enviar al Slave.

```
MC_Bus_ADDR <=  ADDR(31 downto 2)&"00" when block_addr ='0' and Rdir='0' else
                ADDR(31 downto 4)&"0000" when block_addr='1' and Rdir='0' else
                RADDR(31 downto 2)&"00" when block_addr ='0' and Rdir='1' else
                RADDR(31 downto 4)&"0000" when block_addr='1' and Rdir='1' ;
```

Rdir es una nueva señal que hemos incorporado y que es una salida más de la unidad de control de la memoria cache. Activaremos la señal para avisar que tenemos que enviar la dirección del registro y no la de la entrada (la dirección la enviamos antes de cargar los registros).

También conectaremos la salida del registro que guarda el dato en escritura, con la entrada de datos del bus. Ya no necesitaremos escribir en el bus la entrada de datos de la MC sin pasar por el registro.

4- Modificaciones en el MIPS para la gestión de las detenciones.

Para gestionar las nuevas detenciones que deberán producirse en el MIPS, hemos utilizado la señal *Mem_ready* incorporada tal y como se nos indica en el material proporcionado para la elaboración del proyecto. A continuación, explicamos las modificaciones en el código en *MIPs_predictor_imcompleto* para permitir esto.

En primer lugar, no incrementaremos PC cuando Mem_ready valga 0.

```
load_PC <= avanzar_ID and Mem_ready; -- Si paramos en ID, hay que parar también en IF
```

Además, asignamos la señal de carga de los bancos de etapa a la señal Mem_ready. Aquí vemos un ejemplo:

```
Banco_ID_EX: Banco_EX PORT MAP ( clk => clk, reset => reset, load => Mem_ready, b
```

En el banco IF/ID debemos parar por datos y por memoria. En este caso, la señal load del banco se controlará así.

```
avanza_and_mem<=avanzar_ID and Mem_ready;
```

5- Descripción algorítmica de la MC y ciclos efectivos

Nuestra MC gestiona tanto aciertos y fallos en lectura como aciertos y fallos en escritura. Está compuesta por 4 conjuntos de un solo bloque, por lo que no necesita una política de reemplazamiento, tiene emplazamiento directo, política write-through en acierto de escritura, esto supone que escribe el dato en MC y en la memoria principal, y política write-around en fallo de escritura, únicamente se escribirá el dato en la memoria principal. Este es el algoritmo que describe su funcionamiento:

Especificación	Tiempos
Look-up(@x);	1
If (proc_r and miss(@x)) { Mp(rB,@x); waitforMp; Mc+X; }	CrB + 1
If (proc_w and hit(@x)) { Mc+x'; Mp(wW,X'); waitforMp; }	CwW
If (proc_w and miss(@x)) { Mp(wW,X'); waitforMp; }	CwW

En caso de fallo de lectura la penalización será CrB(MP)+1, donde CrB(MP) = 6 ciclos (para la primera palabra) +3* 2 ciclos (para las palabras restantes).

En caso de escritura, ambos casos tienen la misma penalización ya que la escritura en MC se realiza a la vez que la escritura de la palabra en memoria principal, la penalización será CwW(MP) = 7 ciclos.

Si la dirección que recibe el bus es la misma dirección que la enviada en la anterior transferencia CrB(MP) tardará únicamente 9 ciclos, ya que la primera palabra tardará 3 ciclos. En el caso de escritura ocurrirá lo mismo, CwW(MP) será 4 ciclos.

La fórmula de los ciclos efectivos de nuestra memoria es la siguiente:

$$Ceff = 1 + \frac{rm * (CrB + 1)}{R} + \frac{(wh + wm) * CwW}{R}$$

Tras la implementación de un buffer que registra la dirección y el dato a escribir en memoria, conseguimos que, en los eventos de acierto en escritura y fallo en escritura, el procesador pueda seguir ejecutando instrucciones mientras se realiza la transferencia, y el controlador de la MC gestionando aciertos en lectura. Si una instrucción necesita hacer una transferencia en el bus y aún no ha terminado la anterior transferencia, se parará la ejecución del procesador hasta el fin de la transferencia. Esta implementación supondrá en el mejor caso una mejora de los 7 ciclos correspondientes a CwW(MP) por escritura.

6- Pruebas realizadas

Para comprobar el correcto funcionamiento del proyecto, comenzamos realizando pruebas muy simples con el diseño base (sin el buffer para escrituras).

Prueba 1: bucle con todos los casos

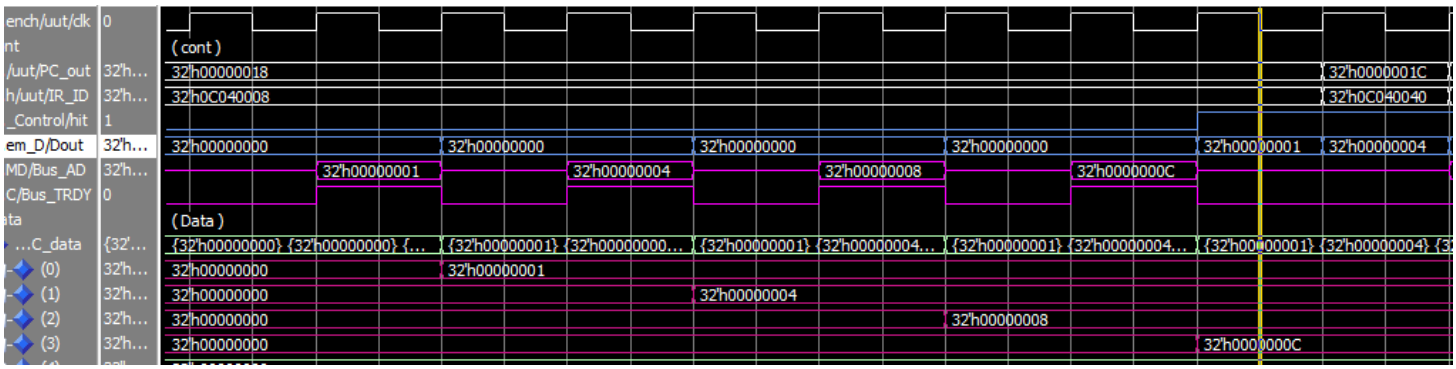
Una de las pruebas más completas que hemos diseñado consiste en un bucle, en el que cada iteración trabaja con un bloque distinto de la cache y hace: fallo de lectura con palabra0, acierto de lectura con palabra1, acierto de escritura con palabra2, fallo de escritura en el mismo bloque (cambiamos el tag).

El código de la prueba es el siguiente:

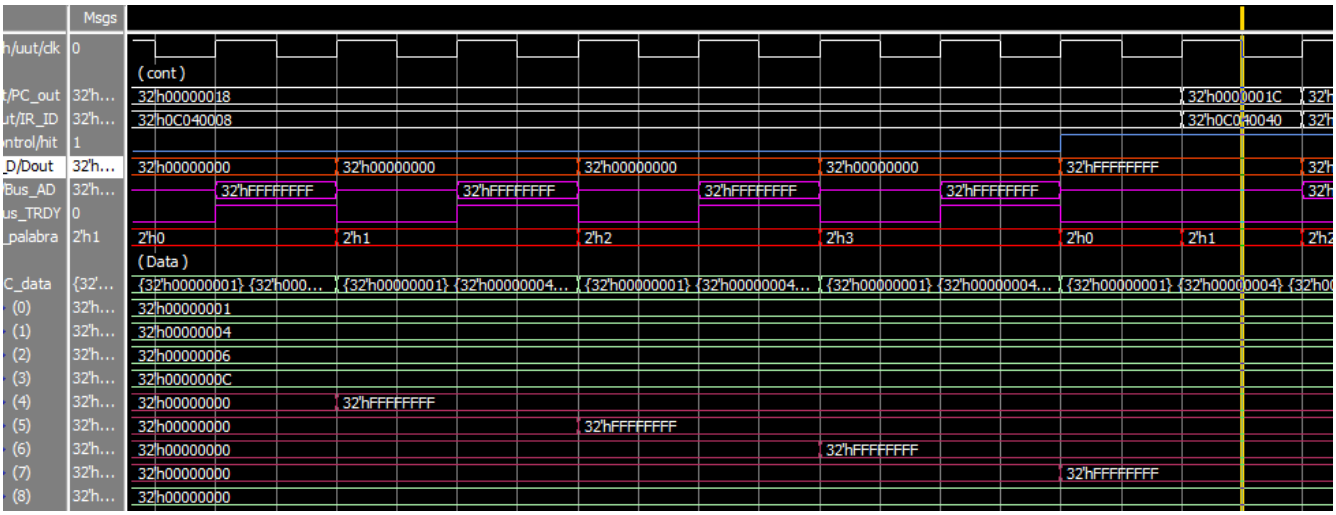
```
        LA R4, 6(R0)
        LA R1, 1(R0)
        LA R2, 0(R0)
bucle:  LW R5, 0(R0)            ;rm palabra 0
        LW R6, 4(R0)            ;rh palabra 1
        SW R4, 8(R0)            ;wh palabra 2
        SW R4, 64(R0)           ;wm palabra 0 (cambiamos el TAG)
        ADD R2, R2, R1
        LA R0, 16(R0)           ;cambio de conjunto
        BNE R2,R4,bucle
        SW R4,12(R0)            ;wm palabra 3
        LW R4,4(R0)             ;rm palabra 1
```

Aprovechamos esta prueba para mostrar el correcto funcionamiento de todos los casos con alguno de los bloques.

1. Fallo de lectura en la palabra 0 (conjunto 0): iteración 1. En la captura podemos ver como el MIPS está detenido. El Slave transfiere el dato, bus_TRDY está activo y al siguiente ciclo se desactiva y la palabra está escrita en cache y lo repite con las otras tres palabras del bloque. Después, hit se pone a 1 y la cache saca el dato de la dirección solicitada (primera palabra) por Dout. Finalmente observamos que el MIPS vuelve a cargar instrucciones.

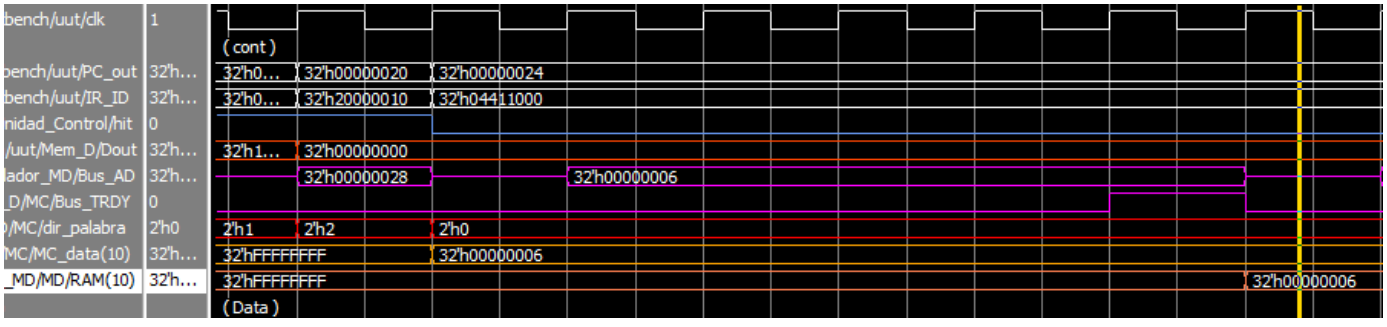


2. Acierto lectura en la palabra 1 (conjunto 1): iteración 2. En la captura observamos como la instrucción anterior ha escrito el conjunto 1 de la cache. Podemos ver (donde está el cursor amarillo) que la palabra solicitada es la 1, y que hit está a 1 porque es acierto, por lo tanto, la salida de la cache tiene el valor correcto.

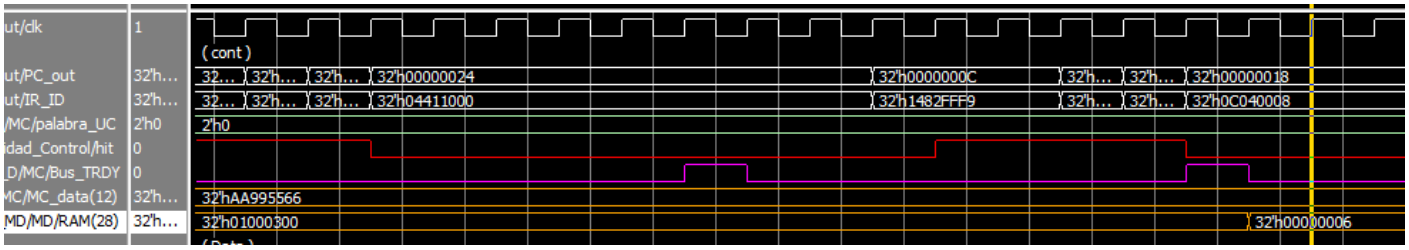


3. Acierto escritura palabra2 (conjunto2): iteración 3. En la siguiente captura vamos a destacar varias cosas. La primera es que la siguiente instrucción es también un SW, y observamos que efectivamente hay detención.

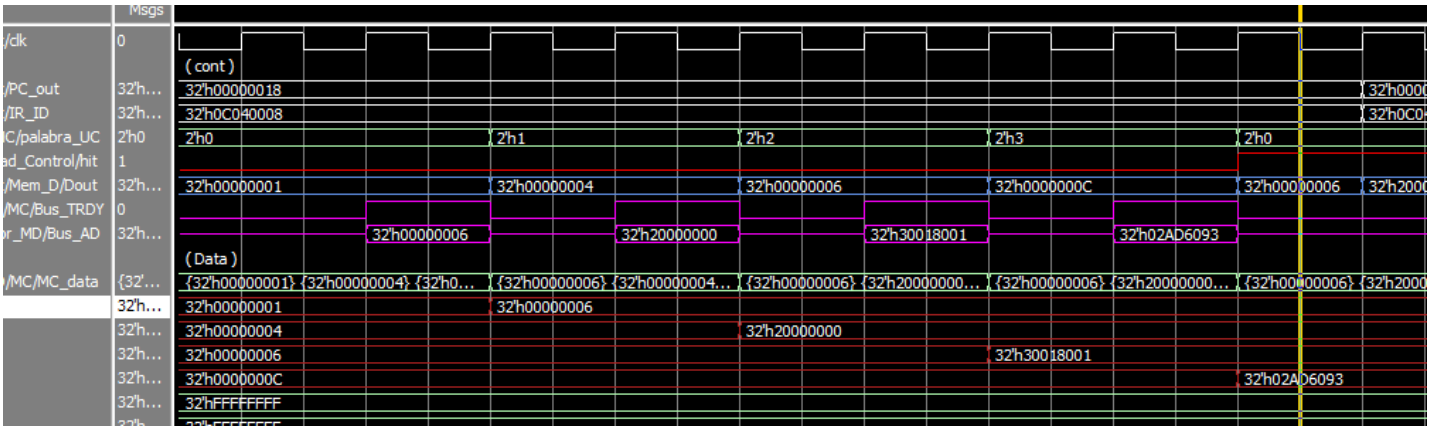
   Por otro lado, al tratarse de acierto de escritura debemos actualizar tanto memoria cache como principal (las líneas naranjas de abajo representan las direcciones de cache y principal). Se detecta el acierto con hit=1 y en el ciclo siguiente se actualiza en cache. Donde apunta el cursor amarillo vemos el dato escrito en memoria principal, un ciclo antes, Bus_TRDY estaba activo.



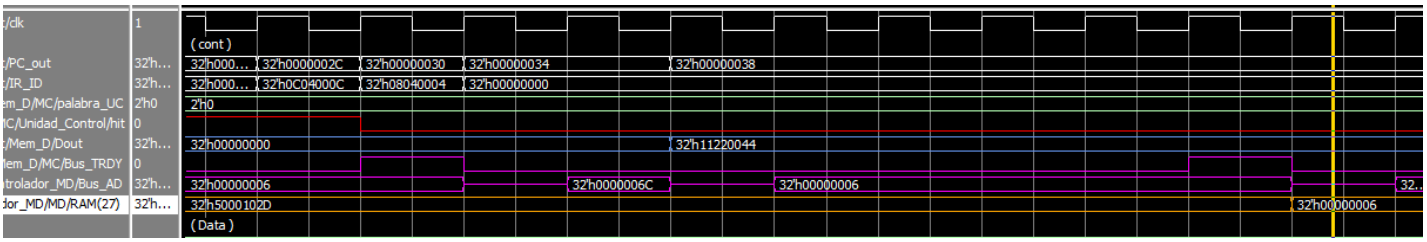
4. Fallo de escritura palabra 0 (conjunto 3): iteración 4. En la siguiente imagen observamos como al tratarse de fallo de escritura no se escribe en cache, solo se escribe memoria principal. En naranja están la dirección de la cache a la que pertenecería la palabra y la dirección de la memoria principal donde se escribe. (La instrucción anterior era una escritura también).



5. Fallo lectura conjunto 0 palabra 0 (el tag ha cambiado): iteración 5. En la siguiente imagen vemos que ocurre un reemplazo en el conjunto 0 provocado por el cambio de tag. Una vez acaba de escribirse en cache, hit se pone a un y sale el dato. En la sexta iteración habrá un reemplazo pero en conjunto 1.



6. Fallo escritura conjunto 2 palabra 3. (fuera del bucle): Vemos como envía la dirección y después el dato, y finalmente lo escribe en la dirección correspondiente de memoria principal (@27).



Prueba 2: Contadores y detenciones

La segunda prueba que presentamos tiene el objetivo de demostrar el correcto funcionamiento de los contadores añadidos y las distintas paradas que pueden producirse.

El código diseñado es el siguiente:

```
LW R1, 0(R0)            ;entre esta instrucción y la siguiente:
SW R1, 0(R0)            ;13 paradas memoria + 1 parada datos
LA R2, 2(R0)
nop
SW R1, 0(R0)            ;7-2-3 paradas de memoria
LA R3, 1(R3)
BNE R3, R2, #-1         ;(2 paradas datos +
ADD R4, R4, R5          ;1 parada de control) x2
ADD R4, R4, R5          ;no se detiene (UA)
```

En esta prueba cubrimos los principales casos de paradas, la segunda instrucción tendrá 1 ciclo de parada de datos (*load-uso*) en la etapa ID y en el siguiente ciclo (el *lw* entra en etapa MEM) se pararán todas las etapas durante 13 ciclos. Cuando la quinta instrucción (*sw*) entra en etapa MEM, a la transferencia del anterior *sw* aún le quedan 2 ciclos de transferencia (escritura tarda 7, como es la misma dirección que la anterior transferencia tarda 3 ciclos menos y han pasado 2 ciclos mientras se ejecutaban las instrucciones *la* y *nop*). Cuando la instrucción *bne* entra en etapa ID, necesita los datos cargados en el BR (parada de datos de 2 ciclos), justo después, el predictor predice NT y falla (parada de control de 1 ciclo), salta a la instrucción anterior y se repiten los 2 ciclos de parada de datos y el ciclo de parada de control. Las 2 ultimas instrucciones no provocan parada de datos gracias a la UA.

El valor de que tienen los contadores cuando acaba la ejecución es el siguiente y coincide con los resultados esperados:

7- Aportaciones, cuantificación de horas y autoevaluación.

Estimación del tiempo dedicado:

Dedicamos 4 horas para analizar y comprender los fuentes proporcionados, incluyendo el autómata del Slave. En el diseño inicial de la Unidad de Control nos llevó 3 horas y media. En la elaboración de la parte opcional con el buffer para escrituras invertimos 5 horas. La depuración de los dos diseños (con y sin buffer) nos llevó 25 horas. Finalmente, la memoria 4 horas.

Reparto del trabajo

Comenzamos trabajando conjuntamente para comprender el proyecto y los fuentes, y elaboramos el autómata del diseño base entre los dos. También depuramos este diseño juntos.

Planteamos el buffer de escrituras y fue Sergio el que realizó el diagrama de estados. Una vez hecho, Irene realizó pruebas para ver que todo iba correctamente y los fallos encontrados fueron corregidos por ambos. Finalmente, diseñamos las pruebas finales y la memoria fue realizada entre los dos.

Autoevaluación: Sergio

He aprendido mucho con esta asignatura, creo que he cumplido sus objetivos y me hubiera gustado dedicarle más tiempo. Si todo va bien, el año que viene me matricularé en la especialidad de Hardware para aprender y dedicar a ello unos años de mi vida. Si me puntuara a mi mismo en esta asignatura mi nota sería un 9.

Autoevaluación: Irene

Considero que he logrado cumplir los objetivos de la asignatura y he conseguido llevar su estudio al día gracias a la realización de las prácticas y los problemas. Los dos proyectos me han servido para entender todo y enlazar los conocimientos estudiados. Creo que mi nota en esta asignatura podría ser un notable.