# **Memoria Proyecto 2**.

Jerarquía de Memoria. MIPS Segmentado y Memoria Cache

Rael Clariana, 760167
Devid Dokash, 780131

27 de Mayo de 2022

## Índice

# 1. Introducción

Tras cumplir los objetivos de la primera parte del proyecto e implementar un procesador MIPS segmentado de 5 ciclos con ruta de datos de operaciones enteras y en coma flotante con unidad de anticipación en parte entera y unidad de detención de ambas partes, en este proyecto, se introducirá en el procesador una jerarquía de memoria ya especificada en VHDL.

Esta jerarquía incluye una Memoria cache, que peleará por el acceso a Memoria frente a un componente IO Master, una memoria de datos más lenta y una segunda memoria de datos, la memoria Scratch, una memoria de acceso rápido cuyos contenidos no deben guardarse en memoria cache, todos interconectados con un bus multiplexado semi síncrono, y un arbitro que gestionará la entrada al bus entre los distintos Masters. El objetivo del proyecto es la implementación de la unidad de control de la memoria cache para el correcto funcionamiento de la jerarquía en el procesador.

La dirección de la memoria se descompone de la siguiente manera:


Figura 1: Descomposición de la dirección.

# 2. Implementación de la Memoria Cache

1. **Estado FETCH**: primer estado de la unidad de control de la Memoria Cache, en este estado se realiza la decodificación de la dirección recibida y su estado en caché.
   - Si la instrucción no es de lectura ni escritura, la unidad de control ignorará la dirección.
   - Si la instrucción es de lectura y el tag tras la decodificación se encuentra en cache, pone el dato correspondiente en el bus de salida en el mismo ciclo.
   - Si la instrucción es de lectura pero el tag no esta en cache, realizará una petición del bus hasta que se lo otorguen y paralizará el procesador.
   - Si la instrucción es de escritura, independientemente de si el tag esta o no en cache, realizará la petición del bus hasta que se lo otorguen y paralizará el procesador.

   Una vez le otorguen el bus, la unidad de control pasará a enviar la dirección y las señales de control correspondientes.

2. **Estado SEND_ADDR**: segundo estado de la unidad de control, en este estado se envían la dirección recibida en estado de *Fetch* y las señales de control correspondientes:
   - Si la instrucción era de lectura, por una parte, la dirección enviada será la dirección del bloque correspondiente si la dirección es cacheable o directamente la dirección recibida ya que no es necesario traer ningún bloque si la dirección no es cacheable, y por otra, una señal indicando que el dato enviado es una dirección (MC_send_addr_ctrl) y una señal indicando que la operación en memoria será de lectura (MC_bus_Rd_Wr = 0).
   - Si la instrucción era de escritura, la dirección a enviar será la recibida en *Fetch* independientemente de si es cacheable o no, y las señales de control indicando que el dato enviado es una dirección y que la operación en memoria será de escritura (MC_bus_Rd_Wr = 1).

   Las señales serán puestas en sus correspondientes buses hasta que el componente perteneciente a la dirección recibida mande una confirmación de que ha reconocido el bus (Bus_DevSel).

3. **Estado DATA_TRNF**: tercer y ultimo estado de la unidad de control, en este estado se realizará la transferencia de datos correspondiente a las acciones realizadas en el estado de *Send_addr* una vez el componente ha reconocido su dirección:

   - Si la instrucción era de lectura y la dirección era cacheable, procederá a la escritura de los datos que vaya recibiendo por parte de la Memoria a través del bus multiplexado en la dirección de la Memoria Cache asociada a la dirección del bloque enviada, y gestionará un contador de palabras escritas para verificar cuando terminará la transferencia y confirmar a la Memoria que ha procesado la misma. Cuando sea la última palabra a escribir, activará la señal last_word y volverá al estado inicial.
   - Si la instrucción era de lectura y la dirección no era cacheable, esperará a que la Memoria envié el dato de la dirección enviada, lo pondrá en el bus de salida, pondrá en marcha el procesador y volverá al estado inicial.
   - Si la instrucción era de escritura, esperará a que la Memoria este disponible y le enviará el dato a través del bus multiplexado, si además, la dirección ha dado acierto en cache, se escribirá en la misma, si ha dado fallo o no es cacheable, no lo escribirá. Una vez la Memoria indique que puede procesar el dato, se pondrá en marcha el procesador y volverá al estado inicial.

   En todos los diferentes casos la unidad de control deberá esperar a que la Memoria de datos indique que esta disponible para continuar la transferencia (bus_TRDY), mientras, seguirá poniendo cada valor en su bus.

# 3. Modificaciones de la Memoria Cache

Se parte de la unidad de control base de la memoria cache y se le implementan una serie de optimizaciones para mejorar su eficiencia en número de ciclos y en tiempo de ejecución. Por una parte, se han añadido nuevos componentes o se han rediseñado otros para poder implementar las optimizaciones.

1. Registros de dirección y dato: para poder permitir que el procesador siga ejecutando instrucciones de manera paralela, la memoria cache necesita guardar la dirección y el dato de la instrucción que ha hecho acceso a memoria ya que si se deja avanzar al procesador, ambos se perderían.

```
-- (linea 182) Registro de dirección:
addr_buffer: reg32 port map (
        Din   => ADDR,
        clk   => clk,
        reset => reset,
        load  => buffer_enable,
        Dout  => saved_addr
);

-- (linea 202) Registro de dato:
data_buffer: reg32 port map (
        Din   => Din,
        clk   => clk,
        reset => reset,
        load  => buffer_enable,
        Dout  => saved_data
);
```

2. Registros de control: para poder verificar si la unidad de control sigue ocupada se debe guardar una señal de control que indique que sigue procesando una escritura en memoria y otra señal de control que indique que el dato ya ha sido anticipado con el bus ya que con el avance del procesador las señales de control se pierden por una parte y por otra, la dirección de la palabra pedida de nuevas instrucciones podría coincidir con el contador de palabra actual y darle un dato incorrecto.

```
-- (linea 150) Dato anticipado:
served_re_Reg : reg1 port map (
        Din   => set_served_re,
        clk   => clk,
        reset => reset,
        load  => re_enable,
        Dout  => served_re
);

-- (linea 158) Ocupado en escritura:
busy_wr_Reg : reg1 port map (
        Din   => set_busy_wr,
        clk   => clk,
        reset => reset,
        load  => wr_enable,
        Dout  => busy_wr
);
```

3. Multiplexor de direcciones: con este multiplexor se elige entre la dirección guardada o la dirección que hay en el bus.

```
-- (linea 196) Multiplexor de direcciones:
mux_addr <= ADDR when (buffer_addr = '0') else saved_addr;
```

4. Pequeñas modificaciones de la cache: ahora la decodificación de la dirección varía según se requiera directamente la dirección de bus o la dirección guardada (mux anterior), la dirección a enviar y la señal de si la dirección es cacheable o no vienen directamente de la dirección guardada, ya que al permitir que avance el procesador, la dirección de bus que era la que usaban ambas señales nombradas desaparece y los valores no son los correspondientes a la instrucción que esta tratando la unidad de control.

```
-- (linea 196) Definicion de addr_non_cacheable:
addr_non_cacheable <= '1' when mux_addr(31 downto 8) = x"100000" else '0';

-- (linea 198) Decodificacion a partir de la salida del mux:
tag <= mux_addr(31 downto 6);
dir_cjto <= mux_addr(5 downto 4); -- Emplazamiento asociativo.
dir_word <= mux_addr(3 downto 2) when (mux_origen = '0') else palabra_UC;

-- (linea 325) Uso de los valores guardados:
MC_Bus_ADDR <= mux_addr(31 downto 2)&"00" when (block_addr ='0')
          else mux_addr(31 downto 4)&"0000";

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

5. Rediseño del arbitro: para poder solapar el ultimo ciclo de transferencia de cualquier máster hay que rediseñar el arbitro de manera que se pueda garantizar el bus en este ultimo ciclo.

```
-- (linea 54) Señal overlap.
overlap <= '1' when (bus_frame= '1') and (last_word = '1') and (Bus_TRDY =
↪  '1')
          else '0';

-- (linea 73) Si req0 esta activado y prioridad vale 0 y se concede el bus
↪  al dispositivo 0.
grant0 <= ((not(bus_frame) or overlap) and ((Req0 and not(priority)) or
↪  (Req0 and not(Req1))));

-- (linea 76) Si req 1 esta activado y prioridad vale 1 se concede el bus
↪  al dispositivo 1.
grant1 <= ((not(bus_frame) or overlap) and ((Req1 and priority) or (Req1
↪  and not(Req0))));
```

Por otra, se han realizado modificaciones a la unidad de control:

1. **Estado FETCH**: adicionalmente en este estado se hace uso del multiplexor, como todavía no ha sido guardada la dirección y o una lectura o una escritura ambas con acierto requieren de la dirección en bus, se hace uso de este multiplexor.
   - *Si la instrucción es de lectura pero el tag no esta en cache*, guardará además la dirección y el dato en los nuevos registros y pondrá la señal de dato anticipado a 0. Una vez otorgado el bus, pasará a enviar la dirección.
   - *Si la instrucción es de escritura*, ahora, escribirá el dato en memoria cache si hay acierto o es una dirección cacheable. Independientemente de si hay acierto o no o si es cacheable o no, guardará la dirección y el dato en los nuevos registros y pondrá la señal de ocupado en escritura a 1 y pasará al nuevo estado *Sav_state* sin paralizar el procesador.
2. **Estado SAV_STATE**: nuevo estado de la unidad de control, en este estado se seguirán procesando todas aquellas instrucciones que sean de lectura y den acierto o aquellas que no hagan acceso a memoria, aquellas que no cumplan ambos requisitos conllevarán la paralización del procesador. Una vez se le garantice el bus, pasará a estado Send_Addr.
3. **Estado SEND_ADDR**: ahora se podrán anticipar instrucciones si se esta procesando una escritura en Memoria:
   - *Si la instrucción era de lectura*, mantiene el mismo comportamiento.
   - *Si la instrucción era de escritura*, ahora también se procesará las instrucciones que cumplan los mismos requisitos que en el estado Sav_state, si no, se paralizará el procesador.
4. **Estado DATA_TRNF**: ahora, se podrá anticipar el dato y además permitir al procesador continuar la ejecución de las instrucciones en paralelo:
   - *Si la instrucción era de lectura y además la dirección es cacheable*, ahora además también tendrá la posibilidad de anticipar el dato, si la palabra pedida coincide con el contador de palabras transferida y no es ultima palabra, la unidad de control anticipará el dato en bus además de escribirla en cache (mux_output = 1), pondrá la señal de que el dato ha sido anticipado a 1 y a partir de aquí, pondrá en marcha el procesador únicamente si las instrucciones no acceden a memoria, ya que si acceden, requieren del bus y ya esta ocupado para la escritura en cache.
   - *Si la instrucción era de lectura y la dirección no era cacheable*, el comportamiento no varia.
   - *Si la instrucción era de escritura*, realizará el mismo procesamiento de las nuevas instrucciones bajo los mismos requisitos. Una vez pase a estado inicial, pone la señal de control de ocupado en escritura a 0.

# 4. Descripción algorítmica de la jerarquía de memoria y expresión de ciclos efectivos

Sea L la latencia de la primera palabra y R la latencia del resto de palabras, entonces:

- El **numero de ciclos de lectura de bloque en la memoria de datos** será su latencia inicial mas el numero de palabras por bloque menos una por su latencia por palabra, que será tres ya que el bloque es de cuatro:

$$CrB(Md) = 5 + 3 * 2 = 11$$

- El **numero de ciclos de escritura de palabra en la memoria de datos** será su latencia inicial más un ciclo de escritura para la palabra recibida:

$$CwW(Md) = 5$$

- El **numero de ciclos de lectura y escritura de palabra de la memoria de datos scratch**:

$$CrwW(MdScratch) = CrW(MdScratch) = CwW(MdScratch) = 2$$

La descripción algorítmica de la jerarquía de memoria viene dada por el siguiente algoritmo junto a sus tiempos:

| Descripción algorítmica de la jerarquía de memoria | Tiempos |
|---|---|
| look-up(@x´) | 1 |
| if miss(@x´) then | - |
| waitfor Arb; | 2 (1 si ext) |
| if (proc_r) then | - |
| if !non_cacheable(@x´) then | - |
| u = FIFO(); Mc-u | 0 |
| Md(rB, @x); waitfor Md; Mc+x; | CrB(Md) + 3 |
| else | |
| Mds(rW, @x´); waitfor MdS; ret x´; | CrW(MdS) + 1 |
| end if; | - |
| elsif (proc_w) then | - |
| if !non_cacheable(@x´) then | - |
| Md(wW, X´); waitfor Md; ret; | CwW(Md) + 3 |
| else | - |
| MdS(wW, X´); waitfor MdS; ret; | CwW(MdS) + 1 |
| end if; | - |
| end if; | - |
| end if; | - |
| | - |
| switch (proc_r/w) | - |
| case proc_r: ret x´; | 0 |
| case proc_w: Mc+x´; Md(wW, X´); waitfor Mds; ret; | CwW(Md) + 3; |

Expresión de calculo de los ciclos efectivos donde *los aciertos y fallos en escritura de direcciones cacheables se incluyen juntos* ya que en ambos casos el *número de ciclos es el mismo* y *los fallos de lectura o escritura en direcciones no cacheables se incluyen juntos* ya que en ambos casos también tienen el *mismo numero de ciclos*:

$$C_{ef} = 1 + 1{,}5 + 0 + \frac{\sum w_c \times (CwW\,(Md) + 3)}{R} + \frac{\sum rm_c \times (CrB\,(Md) + 3)}{R} + \frac{\sum m_{nc} \times (CrwW\,(MdS) + 1)}{R}$$

# 5. Ejemplo de código

Para realizar el ejemplo vamos a usar el código de la prueba1_Mips.vhd

```
.code
    LW R0,0(R0)    ; Read-Miss  Via 0 Cjto 0
    LW R1,0(R0)    ; Read-Hit   Via 0 Cjto 0
    ADD R1,R1,R1   ;
    ADD R1,R1,R1   ;
    SW R0,0(R1)    ; Write-Hit  Via 0 Cjto 0
    ADD R1,R1,R1   ;
    LW R2,0(R0)    ; Read-Hit   Via 0 Cjto 0
    ADD R0,R2,R0   ;
    NOP            ;
    SW R0,0(R1)    ; Write-Hit  Via 0 Cjto 0
.end
```

$$SpeedUp = \frac{T_{ex}Antiguo}{T_{ex}nuevo} = \frac{(CPI * I * Tc)}{(CPI * I * Tc)} = \frac{(16 * Tc)}{(43 * Tc)} = 0,372$$

Como se puede observar los ciclos sin memoria cache son mucho menores, esto sucede porque la memoria cache tarda muchos más ciclos en dar el dato ya que ahora la memoria tiene un retardo de acceso y además tiene que acceder a un bus compartido a diferencia de la primera memoria de datos cuyo acceso era de un único flanco ascendente. Para aprovechar la utilización de la cache y asemejarlo a esta primera memoria de datos tendrían que ser programas de baja frecuencia de escrituras y muy alta frecuencia de lecturas pero aun así no podría igualar la eficiencia ya que por el hecho de tener que traer los datos de memoria ya conllevan ciclos adicionales.

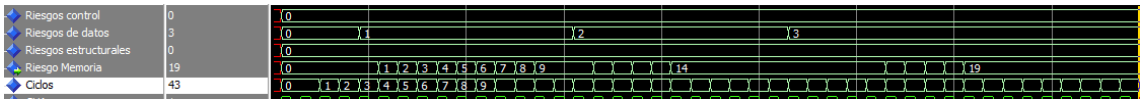


Figura 2: Valores de los contadores durante la ejecución de la prueba.

La ejecución de la prueba tarda 43 ciclos en completarse y hay 19 paradas por MD.

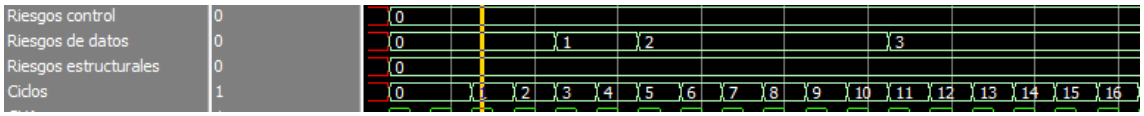


Figura 3: Valores de los contadores durante la ejecución de la prueba pero sin memoria cache.

La ejecución de la prueba con el proyecto 1 como es de esperar, es mucho mas rápida que la actual.

Si la MD tardará varios ciclos en enviar el dato, la memoria cache gracias a los Read-Hit podría enviar el dato mucho antes que sin memoria cache. Por falta de tiempo no hemos podido modificar el proyecto 2 para que no usara la memoria cache, en ese caso si que se vería un speedup favorable a nuestro proyecto con la memoria cache implementado.

# 6. Programas de prueba

Para la depuración y comprobación del MIPS, MC y sus modificaciones se ha realizado un banco de variadas pruebas.

## 6.1. Prueba 1

En esta primera prueba (prueba_op1.vhd) se comprueba el correcto funcionamiento del primer apartado opcional.

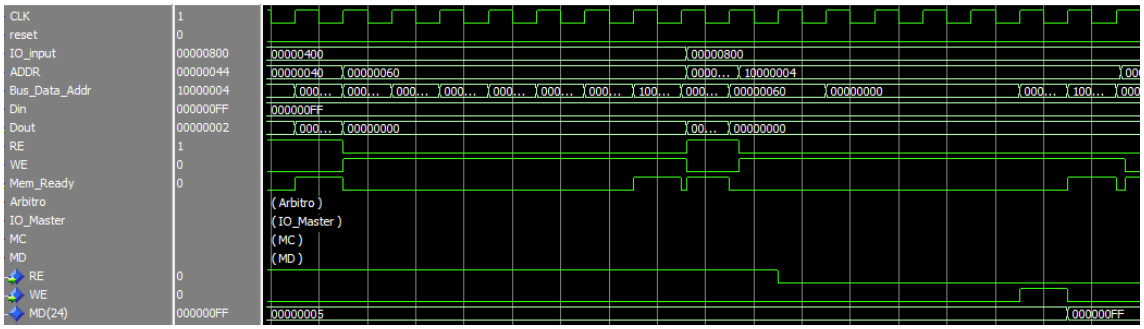


Figura 4: MC gestiona Read-Hit antes de que finalice el Write-Miss

La segunda instrucción de la prueba, tiene como addr(0x00000060) y WE(1) por lo que es un Write-Miss. La tercera instrucción definida con addr(0x00000044) y RE(1), y como la memoria cache tiene el conjunto al que quiere acceder, sera Read-Hit, con que puede realizar su acción aunque el Write-Miss no acabe. En cambio la cuarta instrucción tiene WE(1) por lo que no podrá avanzar hasta que el Write-Miss acabe. Como se ve en la captura cuando se escribe el valor en MD(24).

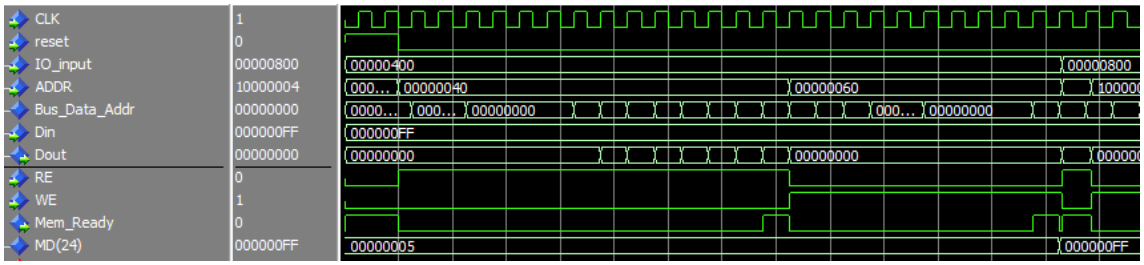


Figura 5: MC no gestiona Read-Hit antes de que finalice el Write-Miss

Hemos ejecutado la misma prueba con la versión del proyecto que no dispone de los apartados opcionales. Esto hará que la memoria cache no gestione el Read-Hit hasta que acabe la escritura. Y el Read-Hit tardara 10 ciclos en poder avanzar desde el comienzo de la escritura, en cambio en el proyecto con las optimizaciones solo tardara 7 ciclos.

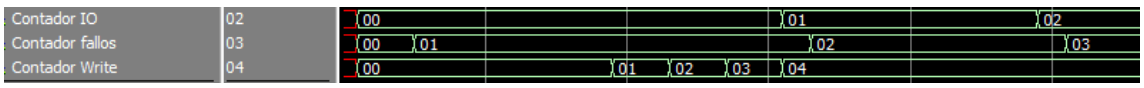


Figura 6: Resultado de los contadores con la optimización.

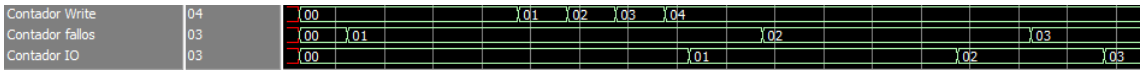


Figura 7: Resultado de los contadores sin la optimización.

Los resultados son prácticamente idénticos pero con las optimizaciones tardará bastantes menos ciclos en ejecutar el programa de pruebas que sin las optimizaciones.

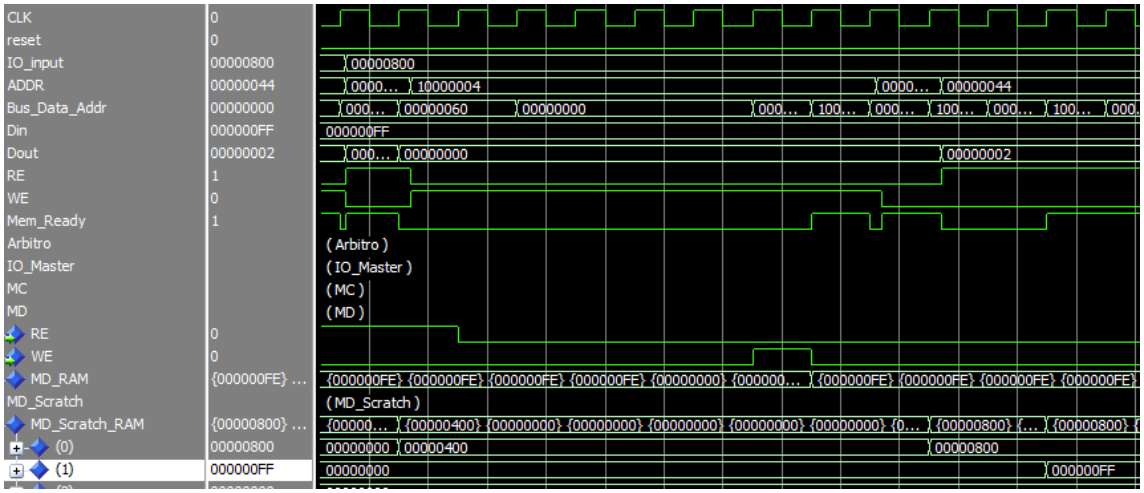


Figura 8: MC gestiona Read-Hit antes de que finalice el Write-Miss en MD_Scratch.

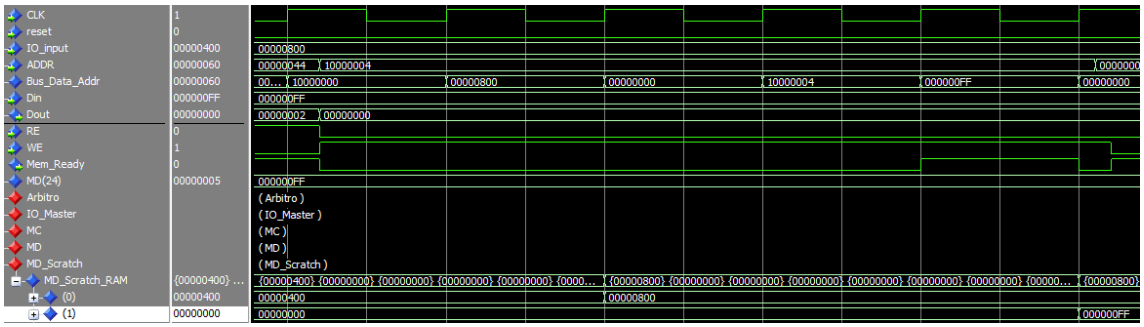


Figura 9: MC no gestiona Read-Hit antes de que finalice el Write-Miss en MD_Scratch.

En las 2 capturas anteriores se ven un caso casi idénticos al primero de esta prueba. Pero en este caso el write escribe en MD_Scratch, pero al seguir siendo un Write-Miss actuara igual que en ejemplo anterior.

## 6.2. Prueba 2

En la segunda prueba (prueba_op2.vhd) se comprueba la correcta ejecución del mecanismo de adelantar la palabra critica. Al realizar dicha optimización permites que las acciones con WE(0) y RE(0) avancen mientras la memoria cache termina de traer el bloque.

La primera instrucción de la prueba, tiene como addr(0x00000040) y RE(1) por lo que es un Read-Miss. La segunda instrucción definida con addr(0x00000000) y RE(0) y WE(0), en cuanto le llega

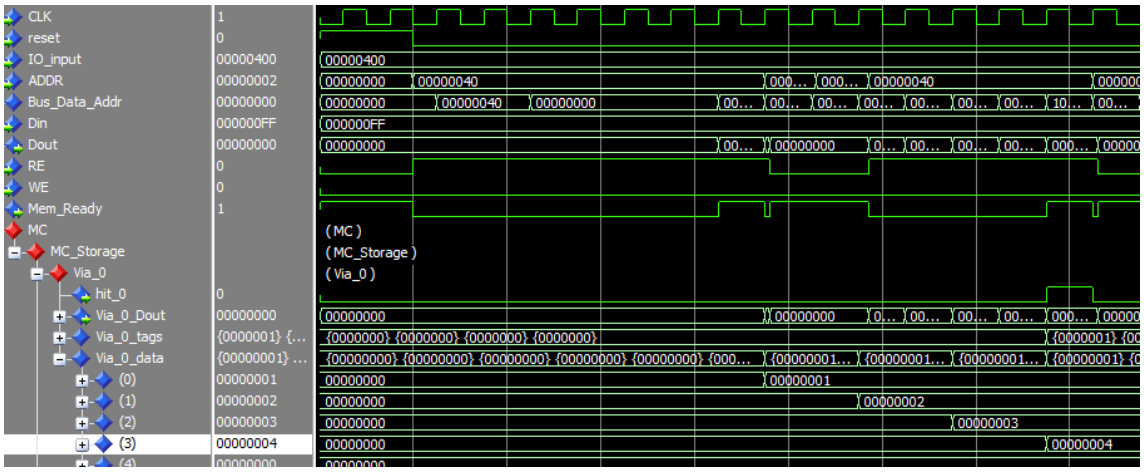


Figura 10: MC gestiona instrucciones que no usan Memoria antes de que finalice el Read-Miss

la palabra que el Read-Miss necesita, dejara pasar a la segunda instrucción que no usa memoria, también permite que pase la tercera instrucción que tampoco usa memoria. En cambio la cuarta que es otro Read-Miss no podrá avanzar hasta que el Read-Miss inicial traiga el bloque entero a memoria cache.

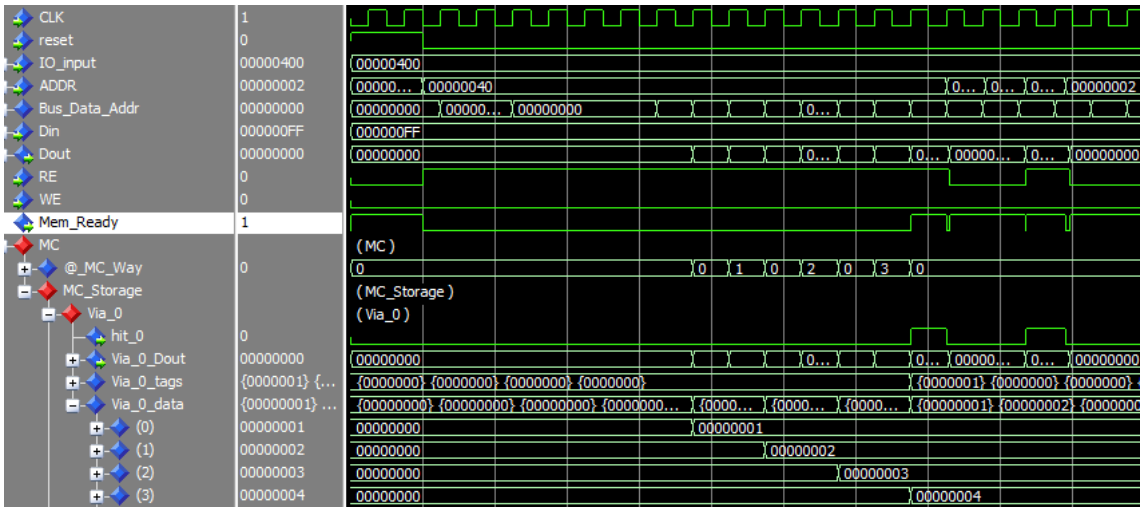


Figura 11: MC no gestiona instrucciones que no usan Memoria antes de que finalice el Read-Miss

Hemos ejecutado la misma prueba con la otra versión del proyecto que no dispone de los apartados opcionales. Entonces no dispondrá del mecanismo de adelantar la palabra critica entonces la CPU no podrá avanzar instrucciones hasta que el Read-Miss traiga el conjunto entero a la memoria cache.

## 6.3. Prueba 3

En la tercera prueba (prueba1.vhd) se comprueba la correcta ejecución de la memoria cache implementada en este proyecto por falta de tiempo no hemos podido documentarla con capturas y explicación pero si que la hemos comprobado y su ejecución es correcta.

## 6.4. Prueba 4

En la cuarta prueba (prueba1_Mips.vhd) se comprueba el correcto funcionamiento de la memoria cache implementándola al mips creado en el anterior proyecto, con unas cuantas modificaciones para que funcione correctamente.

```
.code
    LW R0,0(R0)   ; R0 = @[0] = 1;
    LW R1,0(R0)   ; R1 = @[1] = 1;
    ADD R1,R1,R1  ; R1 = R1 + R1 = 2;
    ADD R1,R1,R1  ; R1 = R1 + R1 = 4;
    SW R0,0(R1)   ; @[4] = 1;
    ADD R1,R1,R1  ; R1 = R1 + R1 = 8;
    LW R2,0(R0)   ; R2 = 1
    ADD R0,R2,R0  ; R0 = 2
    NOP
    SW R0,0(R1)   ; @[8] = 2
.end
```

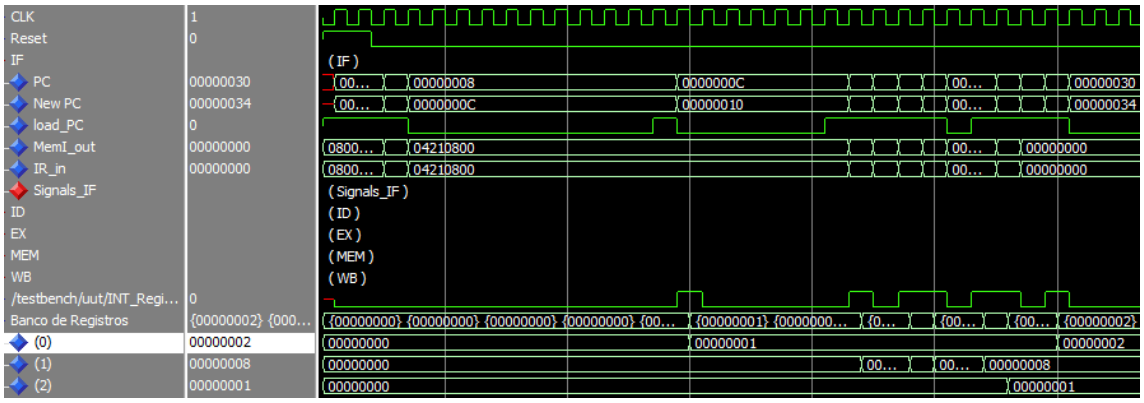


Figura 12: Valores de la MD durante la ejecución de la prueba.

En la captura se muestran los datos de los registros enteros durante la ejecución de la prueba(prueba1_mips.vhd). Como se ve en las instrucciones mostradas anteriormente son correctos.

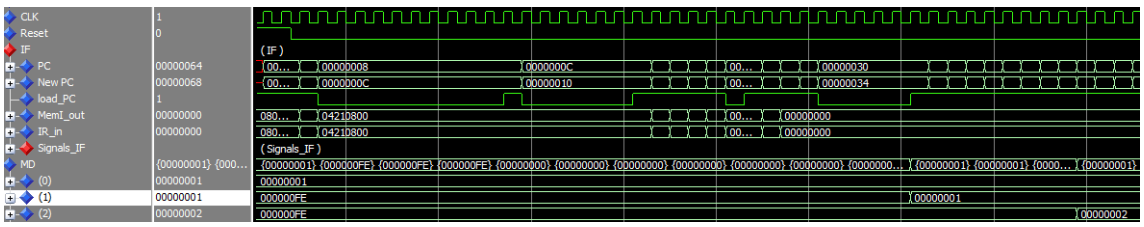


Figura 13: Valores de los registros Int durante la ejecución de la prueba.

En la captura se muestran los datos de la memoria MD durante la ejecución de la prueba(prueba1_mips.vhd). Como se ve en las instrucciones mostradas anteriormente son correctos.

La ejecución de la prueba tarda 43 ciclos en completarse y hay 19 ciclos perdidos por paradas de memoria.

La ejecución de la prueba con el proyecto 1 como es de esperar, es mucho mas rápida que la actual.

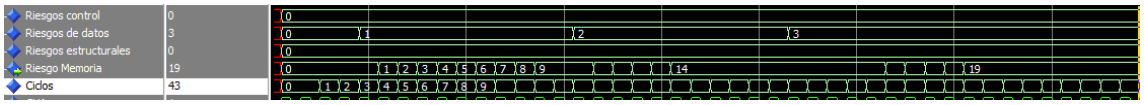


Figura 14: Valores de los contadores durante la ejecución de la prueba.

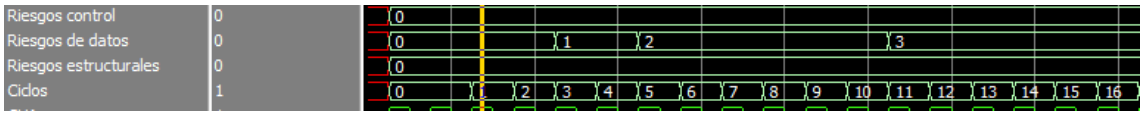


Figura 15: Valores de los contadores durante la ejecución de la prueba pero sin memoria cache.

### 6.5. Modificaciones opcionales y casos explicativos:

1. Incluir un buffer para las escrituras en Memoria de Datos: el rendimiento de esta modificación es notable si existe un gran número de instrucciones que sean lecturas acertadas cache o sean instrucciones que no hagan acceso a memoria, y si además las instrucciones de escritura en memoria, si hay, se encuentran lo suficientemente separadas entre sí, la ejecución del procesador se ve apenas afectada, ya que mientras la memoria cache se encuentra escribiendo el dato enviado por el procesador, este puede seguir ejecutando el resto de instrucciones y con la suficiente separación en ciclos, la siguiente escritura en memoria podrá realizarse sin espera. En otras condiciones, tales como abundancia de escrituras de memoria o lecturas fallidas apenas se ven afectadas por la mejoría.

2. Adelantar el envío de la palabra crítica: el rendimiento en esta modificación es notable si tras el envío de la palabra crítica las instrucciones que siguen a la actual son instrucciones que no realizan ningún acceso a memoria, pero si no es así, la mejoría será poco notable ya que inmediatamente de haber puesto en marcha el procesador deberá parar de nuevo debido a que el bus de la cache esta siendo utilizado y no esta preparado para múltiples operaciones simultaneas, con ciertas modificaciones de la unidad de control es posible mejorar el rendimiento y permitir que siguientes instrucciones con lecturas acertadas en otros bloques o en el propio bloque que esta siendo escrito a palabras ya escritas puedan obtener su dato y se escriba sin problemas el resto del bloque, algo que no ha sido implementado.

## 7. Control de esfuerzos

| Nombre | Rael Clariana | Devid Dokash |
|---|---|---|
| Estudio previo | 1 horas | 1 horas |
| UC | 8 horas | 10 horas |
| Depuración | 5 horas | 5 horas |
| Pruebas | 6 horas | 3 horas |
| Memoria | 6 horas | 8 horas |
| Total | 26 horas | 27 horas |

## 8. Conclusiones

Un proyecto suficiente y completo que junto a la práctica dedicada a memoria cache refuerza las ideas estudiadas en la segunda parte de la asignatura que cumple con su objetivo, el objetivo de que la persona que realice el proyecto, y no realice", si no que realmente ponga el esfuerzo

necesario para poder asimilar los diferentes conceptos, los asimile realmente, al igual que en el primer proyecto, además, de ya no una toma de contacto con el lenguaje VHDL si no que además de una profundización del lenguaje para comprender aspectos más internos junto a los programas de simulación.

La principal dificultad encontrada ha sido el diseño y evaluación de la unidad de control de la memoria cache junto a su depuración, fallos que han surgido y que aunque han sido estudiados y trabajados no han podido comprenderse el origen de los mismos, llevando bastante tiempo el corregirlos. Finalmente, tras este proyecto se puede concluir que los objetivos de la asignatura han sido completamente satisfechos con creces, y como añadido, hablando de cual sería realmente esta autoevaluación, no se puede aplicar a la perfección si es objetivamente hablando, y aunque nuestros esfuerzos han sido bastante altos tanto en diseño y documentación como en la implementación de modificaciones opcionales junto a la observación de errores e incongruencias en el código, también esta el hecho de las horas adicionales para poder concluir de manera correcta el proyecto y de ciertos errores que no hemos llegado a descubrir porque, llegamos a la conclusión de que es un esfuerzo que no puede ser cuantificado. Aunque un 10... siempre esta bien.

# 9. Anexo

A continuación, los diagramas de estados de la unidad de control de la memoria cache:

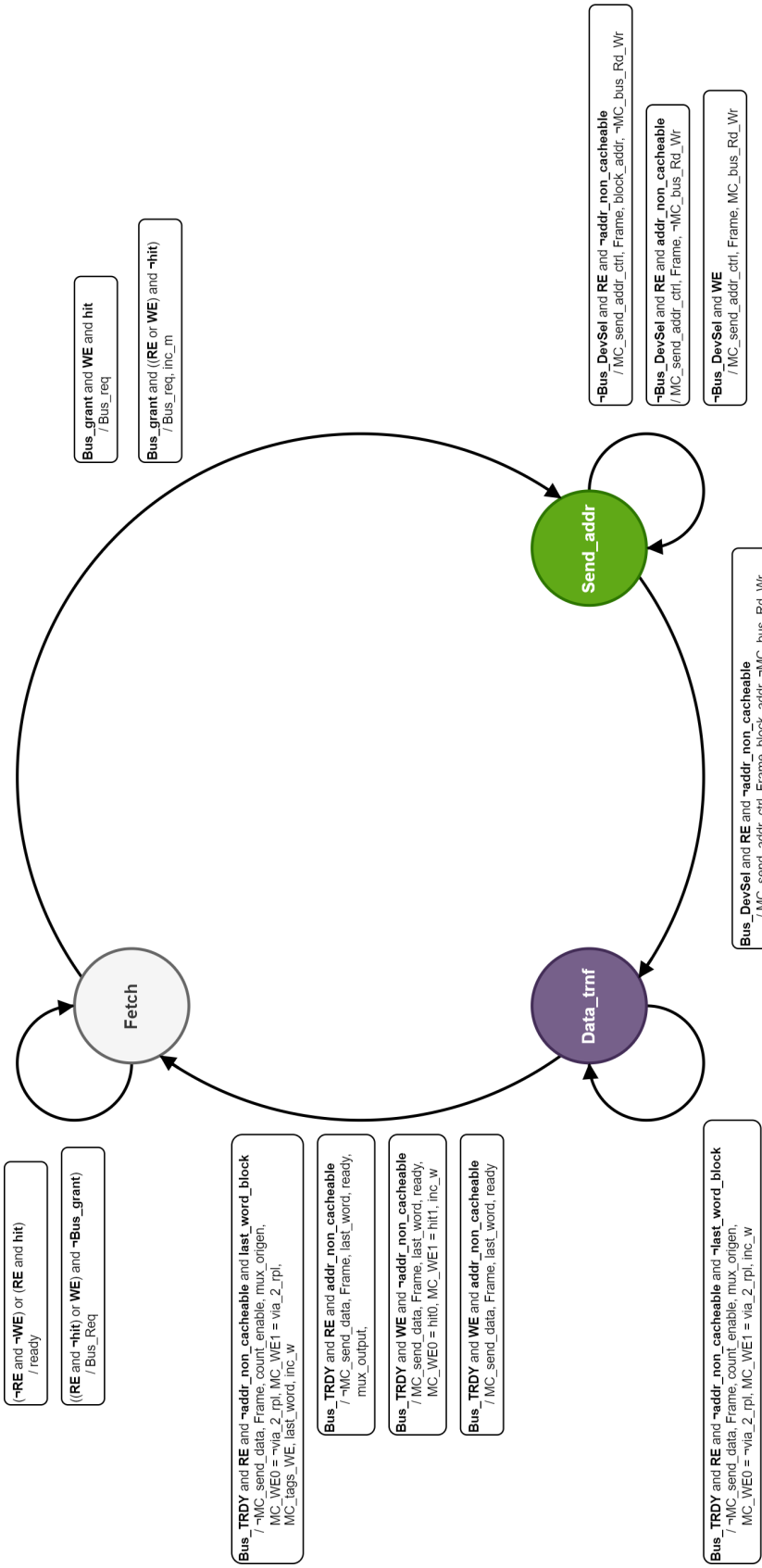


Figura 16: Diagrama de estados del proyecto base.

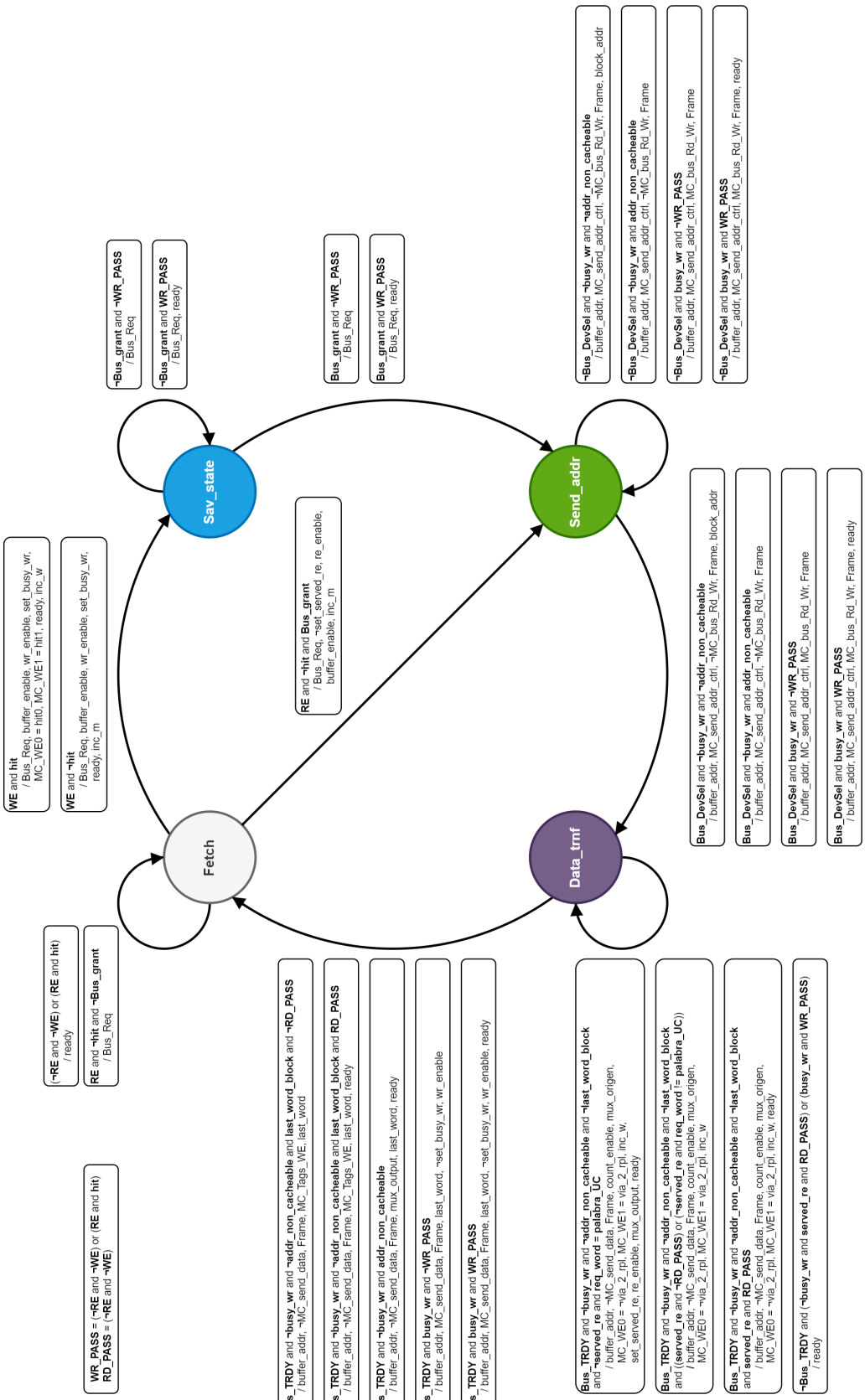


Figura 17: Diagrama de estados del proyecto extendido.